Improving the Compact Bit-Sliced Signature Index COBS for Large Scale Genomic Data

Master's thesis of

Daniel Ferizovic

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Reviewer:	Prof. Dr. Peter Sanders
Advisor:	Dr. Timo Bingmann

02. June 2019 – 02. December 2019

I hereby declare that I have written and developed the enclosed thesis completely by myself, and have not used other sources or means without acknowledging them in the text.

Karlsruhe, 29th November 2019

....

(Daniel Ferizovic)

Abstract

In this thesis we investigate the potential for improving the Compact Bit-Sliced Signature Index (COBS) [BBGI19] for large scale genomic data. COBS was developed by Bingmann et al. and is an inverted text index based on Bloom filters. It can be used to index k-mers of DNA samples or q-grams of plain text data and is queried using approximate pattern matching based on the k-mer (or q-gram) profile of a query. In their work Bingmann et al. demonstrated a couple of advantages COBS has over other state of the art approximate k-mer-based indices, some of which are extraordinary fast query and construction times, but as well as the fact that COBS can be constructed and queried even if the index does not fit into main memory. This is one of the reasons we decided to look more closely at some areas we could improve COBS. Our main goal is to make COBS more scalable. Scalability is a very important factor when it comes to handling DNA related data. This is because the amount of sequenced data stored in publicly available archives nearly doubles every year, making it difficult to handle even from the perspective of resources alone. We focus on two main areas in which we try to improve COBS. Those are index compression through clustering and distribution. The thesis presents our findings and improvements achieved in respect to those areas.

Zusammenfassung

In dieser Arbeit untersuchen wir verschiedene Möglichkeiten den kompakten bit-transponierten Signaturindex COBS [BBGI19] bei der Anwendung für Genomdatenbanken zu verbessern. COBS wurde von Bingmann et al. entwickelt und ist ein invertierter Textindex basierend auf Bloomfiltern. Der Hauptanwendungsbereich von COBS ist die Indizierung von Nukleotidfolgen die durch DNA-Sequenzierung gewonnen wurden. Er kann jedoch auch als allgemeiner Textindex benutzt werden. COBS beantwortet Anfragen durch einen approximativen Musterabgleich indem er sich die k-mer (oder q-gram) Profile der Anfragen zu nutze macht. Bingman et al. haben in ihrer Arbeit gezeigt das COBS einige Vorteile gegenüber Ansätzen hat, die dem Stand der Forschung im Bereich k-mer basiertes approximatives Textindizieren entsprechen. Einige von den Vorteilen sind sehr schnelle Konstruktion und Bearbeitung von Anfragen, sowohl die Möglichkeit den Index zu Konstruieren wenn dieser nicht im Hauptspeicher passt. Aus diesen Gründen, die sich zunächst vielversprechend anhören, haben wir uns entschlossen anzuschauen wie man COBS verbessern könnte. Das Hauptziel dieser Arbeit ist COBS skalierbar zu machen. Skalierbarkeit ist eine wichtige Eigenschaft eines Textindexes der mit großen Datenmengen umgehen muss. Die Größe von öffentlich verfügbaren Genomdatenbanken verdoppelt sich fast jedes Jahr. Dies ist insbesondere problematisch aus der Sicht von Resourcen die ein Rechner zu bieten hat, weil diese schnell an ihre Grenzen stoßen. In dieser Arbeit fokusieren wir uns auf zwei Aspekte die das Ziel haben COBS zu verbessern. Dies sind Indexkomprimierung durch Clustering und die Entwicklung einer verteilten COBS Variante.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

All living beings on our planet share a common ancestry, all the way back to the first cellular organisms that have roamed the Earth. Ever since then, it took evolution tens of millions of years to produce the vast spectrum of life that we are able to see today. The differences in capabilities and appearances between living organisms are encoded within their genetic material in form of DNA molecules. A single DNA string is encoded with 4 letters `{A, T, C, G}`, one for each nucleotide that can be present in the molecule. This information is extremely valuable and being able to extract and understand it is of great importance for us humans. The process of extracting this information from living samples is called **sequencing**. In sequencing a DNA molecule is broken up into many smaller pieces from which the encoding is independently inferred. Those pieces are called reads. In order to recreate the whole string a process called **assembly** is utilized, which tries to reconstruct the complete DNA string based on the inferred reads. The sequencing processes is by nature prone to errors, which makes the assembly problem even more difficult than it already is. Ever since the first DNA was sequenced in 1970, many advances have been made that lead to the development of faster, better and cheaper sequencing methods. The advent of next-generation sequencing methods in the recent years lead to an explosion of stored data in publicly available nucleotide archives, such as the European Nucleotide Archive (ENA). As of today ENA contains more than 2×10^9 microbial sequences, while the European Bioinformatics Institute reported to have reached 160 PB of storage in 2018 [CLS+18]. What makes matters worse is that these archives are subjected to exponential growth, doubling every 18 months.

Being able to search those archives for certain nucleotide patterns has a multitude of applications, including contact tracing of infectious diseases, detecting the spread of bacteria which are resistant to certain drugs, locating mutations responsible for known hereditary diseases and more. The most widely used tool for this purpose has been BLAST, which was developed in 1990 [AGM+90]. BLAST and its successors perform lookups by aligning a query sequence against an assembled genomic sequence. This is problematic because only a fraction of the sequenced data available in ENA is assembled. Their other drawback is that they scale poorly with the rapid increase of sequenced data.

Most of standard text indices from the information retrieval literature, such as the FM-Index, compact suffix array, or the Burrows-Wheeler transform, are also not well suited for this problem. They are not able to handle the huge amount of data with reasonable efficiency. Furthermore the approximate nature of the search problem poses additional difficulties to them. Since the sequencing experiments are prone to errors, when searching for a specific nucleotide sequence we are also interested in sequences that are close enough to our query sequence. Handling such approximate searches becomes extremely difficult for regular text indexes.

For this reason other approaches have been developed that can deal with the approximate nature of the pattern matching more efficiently. One widely used approach in bioinformatics is k-mer (q-gram) based approximate pattern matching [Ukk92]. The term k-mer is used to to indicate a substring of fixed length k. The pattern matching works by storing all k-mers of a given sequencing experiment in some kind of data structure S. A match for a query is reported if enough of its k-mers are contained in S. Some examples of such structures are SBT [SK15], SSBT [SK18], AllSomeSBT [SHCM17], HowDe [HM18], Othello [YLL+18], BIGSI [BdBR+19] and COBS [BBGI19]. They all differ in their approach of designing S as fast and compact as possible. In most cases, however, the building blocks for S are probabilistic structures such as the Bloom filter or quotient filter.

In this thesis we take a closer look at COBS and try to improve it in several aspects. COBS was developed by Bingmann et al. and is a direct successor of BIGSI. In its essence it is an inverted index based on Bloom filters. For each sequencing experiment that is indexed there is an corresponding Bloom filter storing its k-mers. Queries are delegated to each Bloom filter and the result is reported back to the user. Despite its simplistic design COBS has shown to be highly competitive with other state of the art approaches. Bingmann et al. report both faster query and construction times for their index on several data sets. They also demonstrate that only COBS is able to produce indexes exceeding the limitations posed by the main memory present on a machine. There are still some things which COBS is lacking however. Most importantly there is no distributed version of it, which is posing limitations on its use in real world scenarios. Making it more scalable is the main goal of this thesis. In the following sections we present our findings and improvements of COBS, which ultimately led to the development of **Distributed-COBS**.

1.2 Related Work

In biological sequence analysis k-mers have already been used in a variety of settings, some of which are sequence assembly [ZB08], identifying species in metagenomic samples [PB10], predicting bacterial antibiotic resistance, developing attenuated vaccines [ETOK18] and more. The problem of finding all the sequencing experiments containing a particular query sequence has been coined as the *Experiment Discovery Problem* over the past years. It has gained a lot of attraction in the last decade

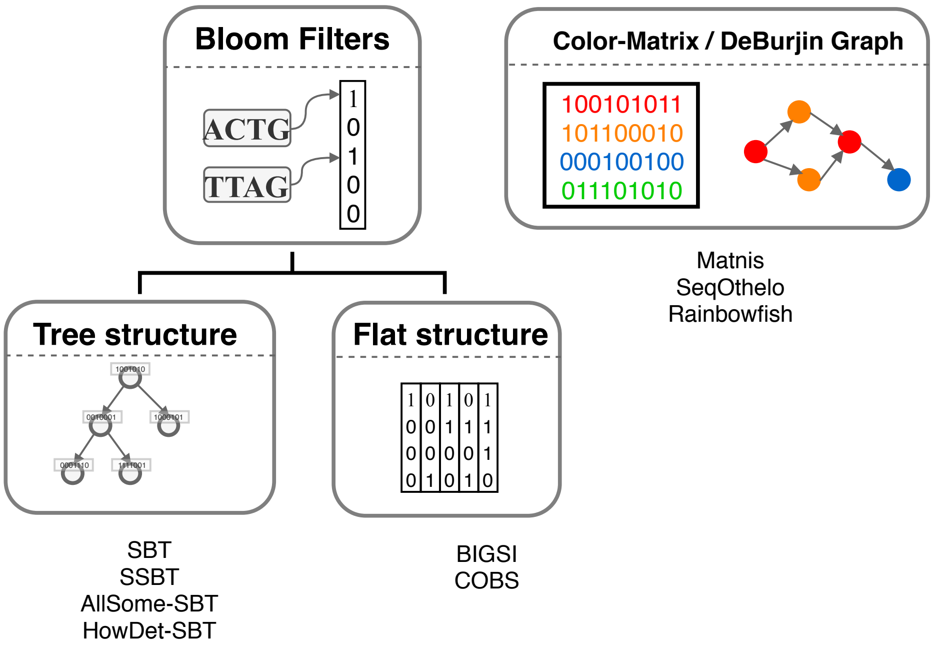


Figure 1.1: Available tools for k-mer-based approximate pattern matching.

and approaches using k-mer-based approximate pattern matching have shown to be especially attractive for solving this problem [CHM19] . They are simple, compact, fast and highly scalable. We present an overview of the available software tools in Figure 5.1

Bloom filter based search structures are widely used for the document discovery problem. **Bloofi** was the first work that proposed using them [CL15]. The authors suggest two different Bloofi variants, one of which is based on hierarchical trees (such as B-trees), while the other has a flat memory design. The Sequence Bloom Tree (**SBT**) by Solomon and Kingman [SK15] is the most prominent k-mer index in literature and is based on the former Bloofi variant. In SBT each sequencing experiment has its k-mer set represented by a Bloom filter stored in the leaves of a binary tree, while the inner nodes represent the union of the Bloom filters stored in their subtree. The tree is constructed using a hierarchical clustering algorithm, combining the most similar Bloom filters in each step. To make the index more compact the Bloom filters are RRR compressed [RRR02]. Queries are performed by a simple search on the tree, only descending into subtrees containing a sufficient amount of k-mers matching the query string. Other software tools using the same approach include the **Split-SBT** [SK18], which is the direct successor of SBT, **AllSome-SBT** [SHCM17] and **HowDet-SBT** [HM18] which have been developed by Medvedev et al. The key difference between them is in the way they encode the information in the inner nodes of the binary trees. In HowDet-SBT for example, there is not a single Bloom filter for representing the union at each inner node, rather there are two bit-vectors called *det* and *how*. *det* indicates which Bloom filter bits are determined to be the same in all Bloom filters in the subtree, while *how* indicates whether the bit is set to 1 or 0.

BIGSI [BdBR+19] and **COBS** [BBGI19] are k-mer indices which are based on the flat Bloofi design. In both cases the index is represented by a sequence of Bloom filters consecutively stored in memory. BIGSI uses a single Bloom filter size for all documents that are being indexed. This leads to a lot of space overhead, due to the varying sizes of sequencing experiments. This issue is improved by COBS, which is the direct successor of BIGSI. COBS provides the ability to construct the index in a compact way by presorting the documents by size and dividing them logically into chunks of fixed size. The Bloom filter size for each chunk is determined by the document of maximum size.

There are also k-mer indices which are not based on Bloom filters. One such example is **SeqOthelo** [YLL+18], which is using a hierarchy of minimal-perfect hash functions to map k-mers to various encodings of occurrence maps. The perfect hashing still permits false positives due to k-mers that are not present in the constructed index. Another software package is **Manits** [PAB+18], based on the k-mer counting tool Squeakr [PBJ+18], which is using counting quotient filters (CQF) in order to construct an occurrence map for each document. In contrast to Bloom filters they can be used in an exact mode - eliminating the one-sided error probability present in Bloom filters. Mantis uses a k-way merge algorithm on the CQFs to obtain a mapping from k-mers to color-IDs, as well as an mapping from color-IDs to color-classes. Each color-class can be interpreted as a signature file, which is a bit-vector indicating in which documents a k-mer is present. All color classes together form the so called color-matrix. To obtain a smaller index the bit-vectors are RRR compressed using the SDSL, and the color-class IDs are Huffman encoded. **Rainbowfish** [MBN+17] is another software tool using the color-matrix. Instead of using CQFs, Rainbowfish is making use of the Burrows Whealer Transform [BOSS12] to encode the k-mer mapping. Both Mantis and Rainbowfish can be seen as implicit colored De-Burjin graph representations of the sequencing experiments. Colored De-Burjin Graphs are a very useful tool in computational biology. They can be used for sequence assembly analysis as well as detection of genetic variations among sequenced samples. Representations of colored De-Burjin graphs often contain k-mer indexes as their underlying structure. Most of them however are not designed for the setting of approximate pattern matching. The most important ones are **Cortex** [ICT+12], **McCortex** [TGIM18] and **VARI** [Mug17].

1.3 Preliminaries

In this section we formalize the problem we are trying to solve and introduce some definitions that will be used throughout this thesis.

1.3.1 Problem Definition

The problem we are studying is k-mer-based approximate pattern matching. Let σ be an arbitrary but finite alphabet. For the purposes of this thesis, we can assume that $\sigma = \{A, C, T, G\}$, but note that we could have chosen any other alphabet as well. First we give a few basic definitions:

Definition 1.1 (**String**).
A string $S = s_1 s_2 ... s_l$ is an ordered and finite sequence over σ.

Definition 1.2 (**Document**).
A document $D = \{S_1, S_2, .., S_m\}$ is a finite set of strings.

Definition 1.3 (**Document Collection**).
A document collection $C = (D_1, D_2, .., D_n)$ is an ordered finite set of documents.

Strings in our setting represent the formal analog to DNA-reads while documents represent sequencing experiments. Next we introduce the notion of ***k*-mers** and ***k*-mer profiles**, which represent substrings of length k for a given string:

Definition 1.4 (***k*-mer of a String**).
A k-mer $w \in \sigma^k$ of a string $S = s_1 s_2 ... s_l$, is a substring of length k in S. In other words, $\exists i \in \{1, ..., l-k+1\} : s_i s_{i+1} ... s_{i+k-1} = w$

Definition 1.5 (***k*-mer Profile of Strings**).
The k-mer profile of a string S, is given as $M_k(S) = \{w \in \sigma^k \mid w \text{ is substring of } S\}$.

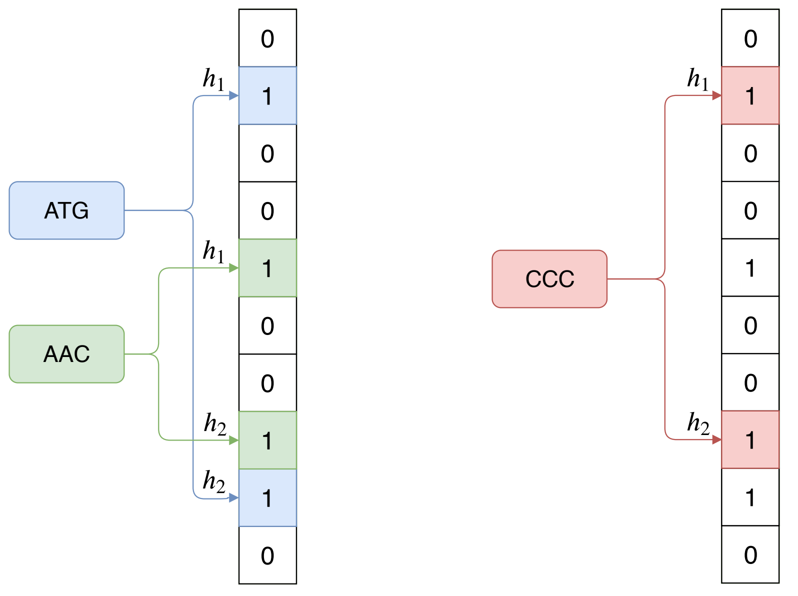
Definition 1.6 (***k*-mer Profile of Documents**).
The k-mer profile of a document D, is given as $M_k(D) = \bigcup_{S \in D} M_k(S)$.

We are now ready to give a formal definition of the problem. Given a document collection C and a query string q. We are interested in finding all documents $D \in C$ such that $|M_k(q) \bigcap M_k(D)| \geq \theta \cdot |M_k(q)|$, with $\theta \in (0, 1]$. The threshold parameter θ regulates how many of the query k-mers need to be present in a document in order for a match to be reported. One important thing to note is that our definition does not take frequencies of individual k-mers into account. This is due to the nature of Bloom filters used by COBS. Approaches that utilize this information may achieve more accurate results than it is the case with Bloom filters.

1.3.2 Bloom Filter

Bloom filters are the fundamental staple on which COBS and other k-mer indices are built on. For this reason, we will use this subsection to formally introduce them.

Definition 1.7 (**Bloom Filter**).
A Bloom filter $B = (A_m, h_1, ...h_r)$ is a compact probabilistic data structure used for representing sets of elements, where A_m is a bit-vector of length m and $h_1, ..., h_r$ are hash functions.



(a) Insertion (b) False positive lookup

Figure 1.2: (a) Insertion and (b) lookup of 3-mers into a Bloom filter with two hash functions. The lookup reports true, despite the fact that the 3-mer `CCC` was never inserted.

To keep things simple we will often use the notation B when referring to the bit-vector A_m, unless otherwise specified. Bloom filters support two main operations, insertion and lookup. At the beginning B is initialized to 0. Inserting an element x into B is equivalent to setting $B[h_i(x)] = 1$ for all $i \in \{1, ..., r\}$. To test whether $x \in B$ we check if $\wedge_{i=1}^{r} B[h_i(x)] = 1$ holds true. Both operations are shown in Figure 1.2. It also shows that the lookups may return false positives, that is, the Bloom filter may report positive membership even for elements $x \notin B$. The false positive rate for a Bloom filter holding n elements can be quantified with the given formula:

$$p = (1 - (1 - \frac{1}{m})^{nr})^r < (1 - e^{-rn/m})^r \tag{1.1}$$

The formula can be easily derived, which is left out here for purposes of simplicity. It is useful for us, because we can see how different parameters affect the false positive rate of the Bloom filter. And even more importantly, for a given false positive rate p and expected number of inserted elements n we can derive good approximations for m and k which leads to:

$$m = -\frac{n \cdot \ln p}{(\ln 2)^2} \quad \wedge \quad r = -\log_2 p \tag{1.2}$$

These approximations are used by COBS to calculate Bloom filter sizes for a given document collection while providing guarantees about the expected false positive rate of the index.

2 Compact Bit-Sliced Signature Index - COBS

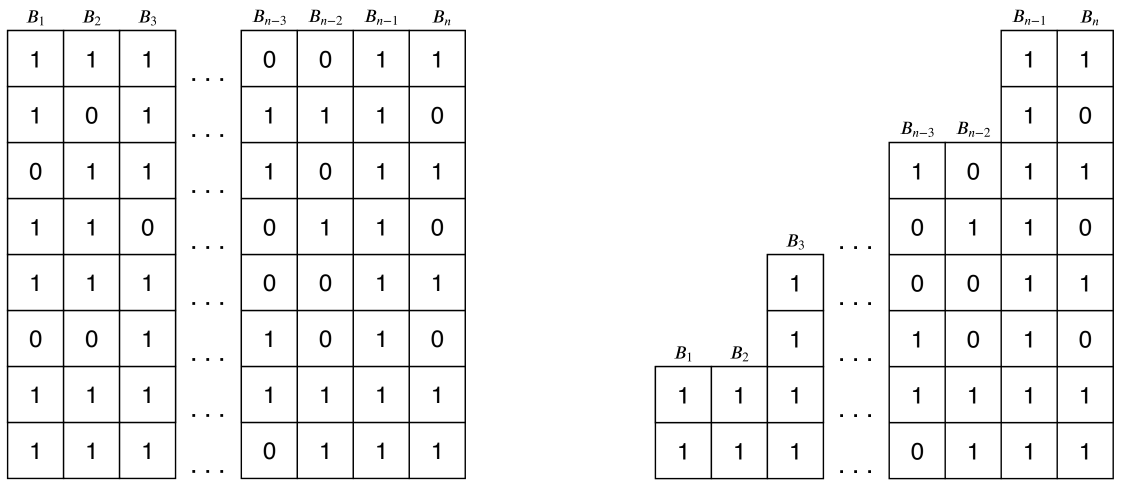
2.1 Overview

COBS is a k-mer index which is based on Bloom filters. For a given document collection $C = \{D_1, ..., D_n\}$ it creates a set of Bloom filters $B_1, ..., B_n$, where B_i stores all the k-mers from document D_i. Each Bloom filter is a signature file of its corresponding document and can be used to answer the membership problem for it. Due to the one sided error of Bloom filters we can only compute an approximation for $|M_k(q) \bigcap M_k(D_i)|$. We will quantify this error in the next section.

In COBS the Bloom filters are not stored separately. They are stored continuously in memory resembling a bit-matrix, as shown Figure 2.1. Each column of the matrix represents a Bloom filter, while each row represents a slice across all Bloom filters. Due to the fact that the documents can be of vastly different sizes, for a fixed false probability rate p the Bloom filter sizes $m_1, ..., m_n$ are also going to vary. To overcome this issue, a single Bloom filter size $m := \max\{m_1, ..., m_n\}$ is chosen across all Bloom filters.

This representation has several benefits to it. For one, accessing rows of the matrix will produce sequential disk reads leading to faster IO-performance due to caching. Another benefit is that one can exploit Single-Instruction-Multiple-Data (SIMD) instructions on modern CPU-s to achieve bit-level parallelism. And lastly, having the same signature size m for all Bloom filters allows us to use the same set of hash functions $h_1, ..., h_k$ across all Bloom filters. There is however one drawback in this representation and that is the memory overhead. Having all Bloom filter sizes being equal to that of the largest document allows COBS to have guarantees for the false positive rate at the cost of additional memory. This is especially bad if the data set consists of many small and a couple of large documents. COBS however comes with two modes of operation: **standard** and **compact**.

In compact mode the memory consumption of the index is reduced significantly. This is achieved by partitioning C into several chunks of documents $C_1, C_2, .., C_w$ and constructing a standard index for each chunk C_i. This alone leads to a smaller memory footprint, because only local maxima will be used for the Bloom filter size of the bit-sliced matrices. To achieve an even smaller discrepancy between the smallest



(a) Standard Bit-Sliced Matrix (b) Comapct Bit-Sliced Matrix

Figure 2.1: Index layout of COBS.

and largest document in a chunk, it uses a clever partitioning technique. First it sorts all documents by size and then it defines the C_i as consecutive chunks of fixed size. Though it is still possible to construct bad input sequences for this approach, it has been shown to provide near optimal Bloom filter sizes on real data sets.

For the chunk size COBS uses the page size of the disk, which is 4096 bytes. This allows the compact mode to combine the best of two worlds, having a smaller memory footprint while being able to exploit the additional benefits we talked about earlier.

2.2 Approximate Search

A query q is performed by counting how many of its k-mers are present in each document Bloom filter B_i. Let $u \in M_k(q)$ be an arbitrary k-mer of q. First all of its hash values get computed $h_1(u), ..., h_r(u)$, which will be used as indices for the rows of the bit-sliced matrix. The corresponding rows of the matrix are scanned sequentially from left to right computing $c_i = \wedge_{j=1}^{r} B_i[h_j(u)]$ for each Bloom filter. Each c_i gets added into a global score array R of size n. After processing all k-mers, $R[i]$ will contain an approximation to the number of k-mers present in D_i. This is shown more illustratively in Figure 2.2. The score array can be further used to rank the documents by relevance and to report only the most relevant documents for a given query.

One important question that arises is what the false positive rate for a single query actually looks like. We note that this is not the same as the false positive rate p of a single Bloom filter. A query q is reported to be contained in D_i if more than $\theta \cdot |M_k(q)|$ of its k-mers are found in B_i. The lookups of individual k-mers can be seen as independent Bernoulli trials, which lead us to the following theorem:

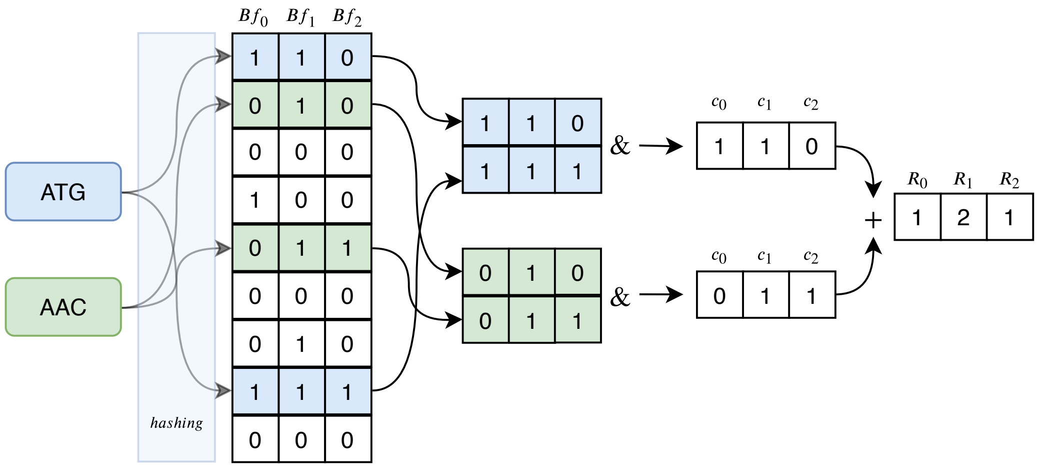


Figure 2.2: Query process of COBS.

Theorem 2.1 (**False Positive Rate of *k*-mer Sets**).
Let $w_0, w_1, ..., w_{t-1}$ be a sequence of t independent k-mers. Further let B be a Bloom filter with false positive rate p. Then the probability that more than $\lfloor \theta \cdot t \rfloor$ k-mers are false positives is $1 - \sum_{i=0}^{\lfloor \theta \cdot t \rfloor} \binom{t}{i} p^i (1-p)^{t-i}$

Using Chernoff bounds for the case $\theta > p$, Solomon and Kingsford [SK15] further bounded the probability that a query is falsely reported to be contained within a document.

Theorem 2.2 (**False Positive Rate of a Query**).
Let q be query with $l = |M_k(q)|$. Further let D be a document with a corresponding Bloom filter B and false positive rate p. The probability for q to be falsely reported as contained in D is bounded by $\leq e^{\frac{-l(\theta-p)^2}{2(1-p)}}$

Since the bound given by Theorem 2.2 is inversely proportional to the query length, it is possible to choose larger false positive rates for Bloom filters than it would be the case in conventional use. While for single membership queries Bloom filters often use values less than 0.01, In COBS the default false positive rate is set to $p = 0.3$, which also leads to the use of a single hash function $r = 1$. Using only one hash function also allows COBS to significantly reduce the number of cache faults as only one row needs to be accessed per k-mer lookup.

2.3 Implementation

COBS is implemented in `C++` and is available online under the following link `https://github.com/bingmann/cobs`.

Launching COBS opens a simple terminal client which allows the user to construct indices from documents and query those respectively. It supports various types of documents, including the most commonly used formats for storing sequencing data: `FASTA`, `FASTQ`, `Multi-FASTA`, `Multi-FASTQ`, `Cortex`, and simple `txt` files. It is also able to read compressed documents, but as of now only supports `gzip` based compression. The implementation also enables manual tweaking of configuration parameters. When building an index the user can specify the index type (*standard* or *compact*), the false positive rate p, the number of hash functions k and even the Bloom filter size. When performing queries, the query threshold θ and the number of documents in the result set can be specified as well. Both the index construction and queries are parallelized using `OpenMP`.

COBS supports out of memory construction which allows for index sizes not limited by the main memory. In the following we describe the *standard* index construction, but the note that the *compact* variant is implemented in a similar manner. COBS loads documents in blocks, one after the other, whose estimated index size fits in main memory. The documents of each block get processed in parallel to obtain a Bloom filter matrix as shown in Figure 2.1. After all indices have been serialized into persistent memory, they get streamed and simultaneously merged into a single index. This is done in parallel over all document blocks.

The query is performed by firstly computing all hash values of its k-mers. They are used to identify which rows of the bit-sliced matrix need to be loaded. In order to speed up disk reads COBS uses memory-mapped IO to access parts of the index. This is achieved with the `mmap` instruction. The rows are consequently logically partitioned and processed in parallel, where each thread is responsible for computing the query score for a subset of documents.

2.4 Contributions

We are now ready to give a full list our contributions towards improving COBS. Those include analyzing the clusterability of Bloom filters, support for batched queries, multi-index search, dynamic index updates and a distributed COBS variant which we call DCOBS. We briefly describe each of those points.

Cluster Analysis As we mentioned in the previous section, COBS does not provide any compression of its index. This was a conscious design decision, since compressing the index, with for example RRR, would add an additional decompression step during the query process. To guarantee fast query times, COBS was kept as simple as possible. Nonetheless, there are still other ways to compress the index without having to sacrifice the query speed as much. We tried to cluster the Bloom filters of the bit-sliced matrix together in order to achieve a more compact index representation. We describe this approach in more detail in Chapter 3.

Batched Queries Executing queries one after the other may not lead to optimal resource utilization. This is due to a multitude of reasons. Reading multiple rows of the bit-sliced matrix will inevitably lead to cache faults, this effect is magnified if the rows are read in an arbitrary fashion. Another drawback of sequential query execution comes into play if the rows that need to be read do not fit into main memory. This will cause each query to re-read the rows from the disk back into the memory, causing expensive disk reads. To overcome this issue we implemented batched queries, which basically executes multiple queries simultaneously. This is further described in Chapter 4.

Multi-Index Search There is no support in COBS for searching multiple indices simultaneously. This means that all documents are contained within a single bit-sliced matrix. We added support for multiple indices in the distributed version of COBS. As we will see later it plays a crucial role for achieving distribution. It allowed us to add support for dynamic index updates and a search filter based on the desired k-mer size of the query. Adding further search filters in the future is desirable, for example one would like to retrieve only for documents coming from a specific taxonomic group. Such filters will require splitting up the index along some attributes in order to support fast queries.

Dynamic Index Updates Dynamic index updates are crucial for any indexing structure. Building the index from scratch is very expensive and depending on the amount of data, this can take anywhere between days and weeks. COBS currently does not support adding new documents to an index, which means everything has to be built anew. This is highly undesirable, especially for the distributed variant where availability plays a crucial role. We implemented dynamic index updates and describe them further in Chapter 5.

Distribution The likely largest contribution of this thesis is the development of a distributed version of COBS. As we have already mentioned the amount of data stored in sequencing archives is overwhelmingly large. On top of that it grows exponentially by each year. Faced with these constraints a single machine quickly runs out of its computing resources, leading to the necessity of distributing COBS on multiple computing nodes. This will ultimately allow COBS to index complete sequencing archives under real time constraints. We develop and present our distributed COBS variant in Chapter 5.

3 Document Clustering

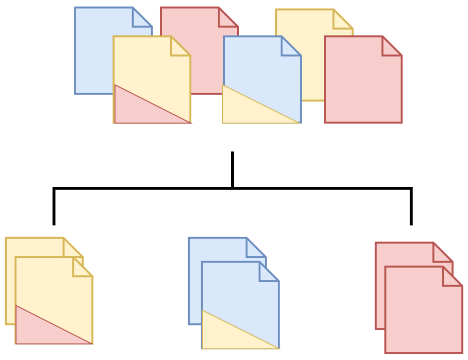


Figure 3.1: Document clustering illustration.

In this chapter we will talk about our efforts to achieve a more compact representation of the bit-sliced matrix in COBS. A straightforward thing to do would be to use a succinct representation for the columns of the bit-sliced matrix. Using RRR-indices to compress the columns would help us reduce the index size somewhat, but at cost of the query execution time. Reading multiple rows of the matrix would translate to a bunch of select queries for each column of the matrix. This would slow down our queries considerably for multiple reasons: overhead due to RRR structure, lower cache efficiency and no instruction parallelism. Because of this we opted to a different approach. The basic idea is to use a single Bloom filter to represent a group of documents and not just one. For this to work the documents within each group have to be reasonably similar. This is because Bloom filters will now store k-mers from multiple documents, leading to an increase in the false positive rate for every k-mer not shared among documents in the same group. To verify whether this approach could work for sequencing experiments we tried to cluster several data sets.

Document clustering has been an ongoing problem in cluster analysis for a long time [Wil88]. Most approaches fall either into hierarchical or partitioning clustering algorithms. But there are also mixtures of both approaches [SKK00]. Documents are usually represented and clustered using *frequency vectors*. The frequency vector of a document $D \in C$ is given as $f_D = (c_1, c_2, c_3...)$, where f_D represents an ordered vector of frequency counts over all words available in the document collection. In our setting, we are working with a slightly different document representation. The frequency vectors are replaced by Bloom filters, and instead of words, documents consist of k-mers. Since Bloom filters are bit-vectors they do not carry any frequency information within them. This makes them an inferior representation for the document clustering problem. We, however, are not directly interested in clustering documents, but rather the Bloom filters obtained from them. This is because one Bloom filter will replace several other Bloom filters in our compressed index. The increase in the false positive rate due to this is directly proportional to how similar the Bloom filters are and not the documents per se.

This reduces our clustering problem to that of clustering high-dimensional bit-vectors. This is difficult for two main reasons, firstly because of the discrete nature of bit-vectors, and secondly becauses of the high dimensionality. The size of Bloom filters obtained in our experiments are usually within the range of $10^8 - 10^9$ bits, making the vector space in which they fall extremely sparse. In the following section we describe in more detail how the clustering of Bloom filters can be exploited to construct a better index representation that uses less space and provides search prunning for queries. We note that we did not end up implementing the proposed structure as our clustering results were not good enough to justify the effort.

3.1 AND/OR Structure

The idea to cluster Bloom filters in the setting of k-mer indices is nothing new. All Bloom tree approaches use a hierarchical clustering algorithm to build the tree structure of their index. Doing so gives them two main advantages: index compression and search pruning. We decided to take is similar approach to that of split sequence Bloom trees. The leaves of their tree store the document Bloom filters, while each inner node consists of two Bloom filters, an *AND* and an *OR* Bloom filter.

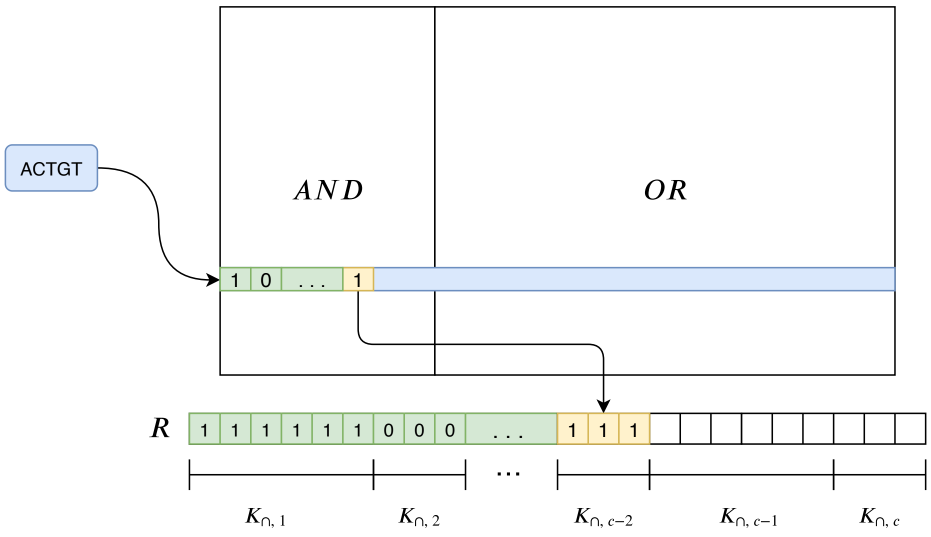
$$AND_T := \bigcap_{B \in T} B, \qquad OR_T := \bigcup_{B \in T} (B - AND_T) \tag{3.1}$$

The *AND* Bloom filter stores the intersection of all Bloom filters in its sub tree, while the *OR* Bloom filter stores the union of the same Bloom filters subtracted by the *AND* Bloom filter. Search pruning using this structure is relatively easy. If a k-mer is present in the *AND* Bloom filter of a node, it is also present in all documents of that sub tree. Thus, the search can be stopped at that point. Otherwise, one looks in the *OR* Bloom filter to decide whether the k-mer is present in any documents at all. Compression is achieved by RRR and improved by not storing bits in child nodes if they are present in the *AND* Bloom filter of the parent.

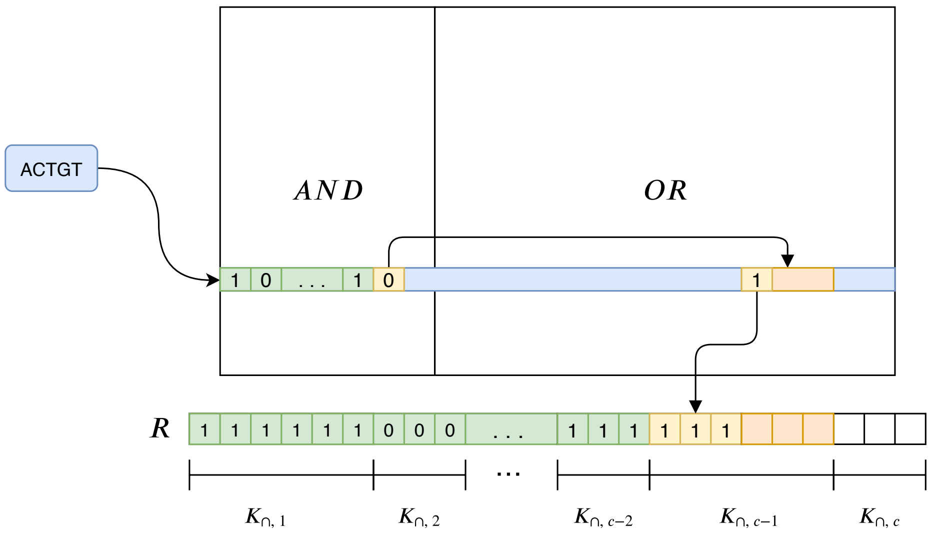
We have already argued against using *RRR* compression in the case of COBS. For the same reasons we do not want to build a tree structure, because it would nullify most of the benefits COBS has to offer. Having those considerations in mind we devised a similar approach to that of SSBT while trying to keep most advantages COBS has to offer. The structure is build in two phases: `AND` and `OR` phase.

`AND Phase` Starts by clustering all Bloom filters $B_1, ..., B_n$ into a fixed number of clusters $K_{\cap,1}, ..., K_{\cap,c}$. We will refer to such clusters as *intersection clusters*. For each of them an AND-Bloom filter is constructed, similar to that of SSBT. It is defined as: $AND_i := \bigcap_{B \in K_{\cap,i}} B$.

`OR Phase` Clusters Bloom filters within each $\widetilde{K}_{\cap,i} := \{B - AND_i \,|\, B \in K_{\cap,i}\}$, obtaining one or more new clusters $K_{\cup,i,j}, ..., K_{\cup,i,c_i}$. We will refer to those clusters as *union clusters*. For each of them an OR-Bloom filter is constructed, again similar to the one in SSBT. It is defined as: $OR_{i,j} := \bigcup_{B \in K_{\cup,i,j}} B$. The resulting AND/OR structure is represented by the concatenation of all AND and OR Bloom filters as illustrated in Fig 3.2.



(a) AND-Bloom filter match



(b) AND-Bloom filter missmatch

Figure 3.2: AND/OR bit-sliced matrix.

In Figure 3.2 we can see how a search would be performed on the new bit-sliced matrix. One important thing to note is that one can determine an arbitrary but fixed order of documents within the score array. The first entries will correspond to documents in $K_{\cap,1}$, then to documents in $K_{\cap,2}$ and so on. The entries for each intersection cluster will be further ordered by their correspondence to the union clusters, first comes $K_{\cup,i,1}$, then $K_{\cup,i,2}$ and so on. The AND and OR Bloom filters are ordered in a similar fashion. This allows for a purely sequential access of the score array. We do however have to sacrifice some cache performance due to pruning. Searching for a particular k-mer only scans the AND-part of the matrix directly. Whenever we hit a bit set to 1, we increase the score of all documents corresponding to that intersection cluster, Figure 3.2 (a). On the contrary, we have to scan all the union Bloom filters corresponding to that intersection cluster, Figure 3.2 (b). This leads to frequent jumps to the OR-part of the bit sliced matrix. Because of the document order proposed above those jumps are not completely random. The jumps will always access increasing parts of memory, mitigating the effects of cache trashing somewhat.

Whether this approach is sensible to use is highly dependant on how similar the Bloom filters are. Intersection clusters consisting of dissimilar Bloom filters will be extremely sparse and be barely of any use at all. Union clusters consisting of dissimilar Bloom filters will increase the false positive rate of their documents drastically. And depending on the number of clusters one finds, the AND/OR matrix may consume even more memory than before, but not more than by a factor of two. In the next sections we will explain the clustering algorithms and techniques we used to cluster our Bloom filters. At the end we give an experimental evaluation on clustering several sequencing data sets.

3.2 Distance Measures

Most clustering algorithms rely on the presence of an distance measure between the points that are being clustered. There are many different distance measures that have been used in the context of document clustering in the past [Hua08]. We have implemented and compared some of the most popular ones, which we introduce in the following.

3.2.1 Euclidean Distance

Let $x = (x_1, ..., x_d)$ and $y = (y_1, ..., y_d)$ be two d-dimensional vectors, x, $y \in \mathbb{R}^n$. The Euclidean distance between them is defined as:

$$dist_E(x, y) := \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{3.2}$$

3.2.2 Hamming Distance

Let $x = (x_1, ..., x_n)$ and $y = (y_1, ..., y_n)$ be two n-dimensional bit-vectors, x, $y \in \{0, 1\}^n$. The Hamming distance between them is defined as:

$$dist_H(x, y) := |x \oplus y| = \sum_{i=1}^{n} x_i \oplus y_i \tag{3.3}$$

where $\oplus$ is the binary `xor` operator. We note that for bit-vectors it holds that $dist_H(x, y) = dist_E(x, y)^2$.

3.2.3 Cosine Distance

Let $x = (x_1, ..., x_n)$ and $y = (y_1, ..., y_n)$ be two n-dimensional vectors, $x,\ y \in \mathbb{R}^n$. The cosine distance between them is defined as:

$$dist_C(x, y) := 1 - \frac{x \cdot y}{||x||\,||y||} \ \in \ [0, 1] \tag{3.4}$$

The later term is the cosine of the angle between the vectors x and y. Thus vectors that have the same orientation will have $dist_C = 0$, while orthogonal vectors will have $dist_C = 1$.

3.2.4 Jaccard distance

Let $X = \{x_0, ..., x_n\}$ and $Y = \{y_0, ..., y_m\}$ be two sets drawn from the same space. The Jaccard distance between them is definded as:

$$dist_J(X, Y) = 1 - \frac{|X \cap Y|}{|X \cup Y|} = 1 - \frac{|X \cap Y|}{|X \cap Y| + |X \triangle Y|} \tag{3.5}$$

We use $\triangle$ to denote the symmetrical difference between two sets. Bit-vectors can also be interpreted as finite sets, with a 1 indicating presence and a 0 indicating absence of an element. As such, we are able to define the Jaccard distance on bit-vectors $x, y \in \{0, 1\}^n$ as well:

$$dist_J(x, y) = 1 - \frac{|x \wedge y|}{|x \wedge y| + |x \oplus y|} \tag{3.6}$$

3.3 Dimensionality Reduction

Documents storing sequencing experiments often contain up to several gigabytes of data. This results in a proportional amount of k-mers extracted from each document. Since our Bloom filters have a fixed false positive rate, their size has to accommodate for the larger number of k-mers. In our experiments the Bloom filter sizes were between 10^8 and 10^9 bits.

This is problematic for two main reason. Firstly, clustering algorithms dealing with high dimensional data are likely to suffer from the *curse of dimensionality.* The space containing our data points is going to be extremely sparse due to the large number of dimensions, reducing the informative value we obtain from distance functions. Secondly, computing the distance between two vectors in a d-dimensional space usually takes $\mathcal{O}(d)$ time. Moreover, the number of dimensions has a multiplicative effect on the running time of our clustering algorithms, leading to a prohibitively large execution time.

To deal with these problems, one usually applies a technique called *dimensionality reduction*. It projects the input data onto a lower dimensional subspace, while ideally preserving important properties such as relative distances. The arguably most famous approach is Principle Component Analysis [WEG87]. It projects the input data onto a new space whose bases point along directions of maximal variability in the input data. Due to its time complexity of $\Omega(d^2)$ it is only feasible for moderately high dimensions. Slightly less sophisticated, but in practice good alternatives are Random Projections [BM01] and Min-Hashes [Bro97]. Random Projections however are still too expensive for our scenario, so we opted to an approach that is similar to Min-Hashes. Instead of choosing the lowest p hash values like in Min-Hash, our reduction consist of choosing a random segment of the Bloom filter and extracting all hash-values from it. Since our Bloom filters are relatively dense and the bits are randomly distributed, it is expected to perform as good as the Min-Hash approach.

3.4 Clustering Algorithms

In this section we give a short overview of the clustering algorithms we have evaluated in our experiments. How exactly we compute distances between clusters, and distances between Bloom filters and clusters, will be explained in the next section when we introduce the notion of centroids.

3.4.1 Hierarchical Clustering

Algorithm 1 Ward's algorithm — Time: $\mathcal{O}(n^2 \log n)$

```
1: procedure Ward($B_1, ..., B_n, c$)
2:     $C := \{C_1, ..., C_n\} \leftarrow B_1, ..., B_n$        ▷ Each Bloom filter is initially a cluster
3:     $heap \leftarrow \emptyset$
4:     for $(i, j) \in \{1, ..., n\}^2$ do
5:         $heap.insert(C_i, C_j)$        ▷ Ordered by $dist(C_i, C_j)$
6:     end for
7:
8:     while $n \geq c$ do
9:         $(C_a, C_b) \leftarrow heap.pop()$        ▷ Gets the closest cluster pair
10:        $C \leftarrow C \setminus \{C_a, C_b\}$
11:        for $C_k \in C$ do
12:            $heap.insert(C_k, C_a \cup C_b)$
13:        end for
14:        $C \leftarrow C \cup \{C_a \cup C_b\}$
15:        $n \leftarrow n - 1$
16:    end while
17:    return $C$
```

The main drawbacks of this algorithm are its running time and space consumption which are both in $\Omega(n^2)$. In order to speed up Algorithm 1 we tried not to insert all possible cluster pairs into the heap. Instead we pick $n' < n$ clusters at random and insert only those pairs into the heap.

3.4.2 K-Means and K-Means++

Algorithm 2 K-Means Time: $\mathcal{O}(n)$

```
1:  procedure KMeans($B_1, ..., B_n, c$)
2:      $C := \{C_1, ..., C_c\} \leftarrow Init(B_1, ..., B_n)$        ▷ Initialization as in [AV07]
3:      for $r \in 1, ..., Rounds$ do
4:          $C'_1, ..., C'_c \leftarrow \emptyset$
5:          for $i \in 1, ..., n$ do
6:              $k = \arg\min_{j \in \{1,...,c\}} dist(B_i, C_j)$
7:              $C'_k \leftarrow C'_k \cup \{B_i\}$
8:          end for
9:          $C \leftarrow \{C'_1, ..., C'_c\}$
10:     end for
11:     return $C$
```

Compared to `Ward's` algorithm `K-Means` has a lower time and space complexity. Its running time is however affected by a lot of hidden constants. There are also some pitfalls and difficulties when it comes to `K-Means`. It is not obvious how to initialize the clusters at the beginning. A poor choice may slow down or prevent the algorithm from finding a good clustering. Instead of choosing points at random, we implemented `K-Means++` [AV07] which tries to maximize the pairwise distances between the initial cluster points. The second problem we had to deal with are empty clusters. Some of the newly computed clusters in line 9 of Algorithm 2 can be empty. To avoid this we re-assign points with the greatest distance to their host cluster into empty clusters.

3.4.3 Bi-Sectional K-Means

This clustering algorithm tries to combine a hierarchical top-down approach with a `K-Means` based partitioning. It has shown to deliver promising results in the setting of document clustering [SKK00].

Algorithm 3 Bi-Sectional K-Means Time: $\mathcal{O}(n)$

```
1:  procedure Bi-Sectional-KMeans($B_1, ..., B_n, c$)
2:      $C := \{B_1, ..., B_n\}$
3:      $heap = \{C\}$
4:      while $heap.size() < c$ do
```

```
5:         $C_k \leftarrow heap.pop()$                    ▷ Cluster with worst evaluation
6:         $(C_a, C_b) \leftarrow KMeans(C_k.elements(), 2)$
7:         $heap.insert(C_a)$
8:         $heap.insert(C_b)$
9:     **end while**
10:    **return** $heap.elements()$
```

What we mean by *worst cluster* in line 5 of Algorithm 3 will become apparent when we introduce our evaluation function in the experimental section of this chapter. But an equally good heuristic would be splitting along the largest cluster.

3.5 Centroid Selection

One thing which was left out from the previous section was how to compute the distances to a cluster, that is $dist(B_i, C_j)$ and $dist(C_i, C_j)$. There are several possible ways to define this. One could for example pick a random document within the cluster and use it to compute the distance. One could also pick multiple random documents and average the sum over all distances to the cluster. We decided to take a *centroid* based approach. A *cluster centroid* is a point which minimizes the sum of distances to all other points in the cluster. The centroid does not have to be a member of the cluster. For example, the centroid of $\{v_1, v_2, ..., v_n\}$ in the d-dimensional Euclidean space is given as:

$$T_w = \frac{1}{n} \sum\nolimits_{i=1}^{n} v_i \tag{3.7}$$

We will denote T_w as the *weighted centroid.* The problem with the above equation is that $T_w \in \mathbb{R}^d$ is not necessarily a bit-vector. This would render some of our metrics useless, such as the Hamming- and Jaccard distance. There are two ways to adapt the centroid based approach for bit-vector metrics as well. One way is by defining a centroid which is also a bit-vector, the *modal centroid*:

$$T_m[i] = \begin{cases} 1, & \text{if } \sum\limits_{B \in C} B[i] \geq \frac{n}{2} \\ 0, & \text{otherwise} \end{cases}, \quad \forall i \in \{1, ..., d\} \tag{3.8}$$

Each coordinate is set to 1 or 0 based on whether the majority of the documents has a 1 or 0 set at that position. It is a natural extension of the *weighted centroid* to the bit-vector space. We also went the other way around and adapted the Hamming distance to work with the *weighted centroid.* This new metric will be denoted by *weighted Hamming distance* in our experiments. The extension to the Euclidean space makes this new metric equivalent to the Manhattan distance.

3.6 Experimental Results

The experiments we did in this section mostly revolve around analyzing whether the clustering idea is feasible in the first place. As it turned out clustering noisy high dimensional bit vectors is difficult. We did not manage to get any meaningful clusters out of our data, as shown the in sections to come. For this reason we did not implement the proposed AND/OR structure but took a hold after performing initial clustering experiments. In order to figure out why the data is difficult to cluster we performed two additional experiments. In the first we tried to cluster manually generated data by introducing random noise, and in the second we tried to investigate the effect that low-frequency k-mers have on the quality of the clustering.

3.6.1 Setup

In the following we give a brief introduction to the platform and sequencing data the experiments were performed on.

Platform We run our experiments on a 32-core machine with 4 x Intel Xeon E5-4640 (2.4 GHz, 8 cores), 512 GiB of DDR3-PC16000 RAM, and 2 TB of SATA hard drives. The machine was running the Linux distribution Ubuntu 18.04 and the code was compiled using `gcc` 7.4.0.

Datasets There are 3 main data sets that we will be using throughout this thesis. The first data set contains sequencing reads of genomic microbial data, which we will call **microbial**. The entire corpus of microbial data present in ENA, about 450,000 files, was indexed for the first time by Bradley et al. [BdBR+19]. The accession list of the data used can be found at http://ftp.ebi.ac.uk/pub/software/bigsi/nat_biotech_2018/ctx/. Depending on our experiments we will be using a subset of those documents. We also note that all our sequencing data is in raw format. That is, unless stated otherwise, we did not clean any low-frequency k-mers from the data in order to remove noise introduced by the sequencing process. Our second data set, which we call **sbt**, contains sequencing reads of trancriptomes obtained from various human tissues. This data set was obtained from Solomon and Kingsford and has been used in the evaluation of their SBT trees. The corresponding accession list can be found in http://ftp.ebi.ac.uk/pub/software/bigsi/nat_biotech_2018/ctx. The third and last data set contains sequencing reads from human gut microbiomes and was obtained from the BioProject `PRJEB2054`. This data set will be denoted as **human-gut**. All of our data was downloaded from the Sequence Read Archive (SRA).

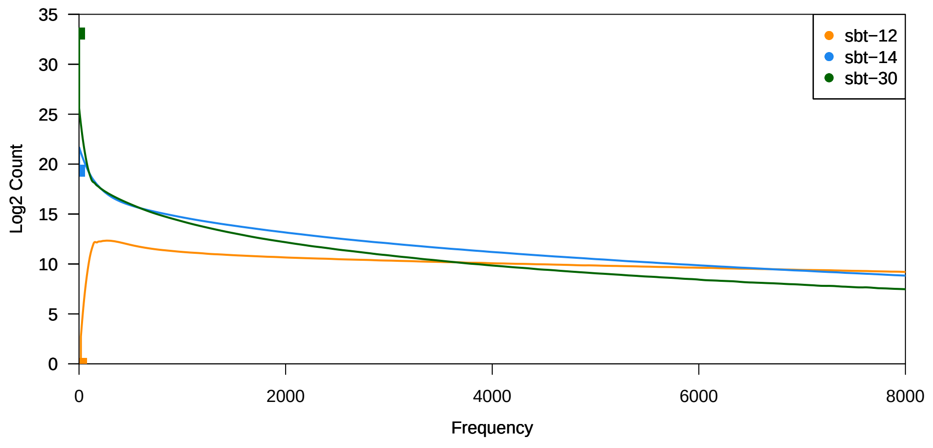
In order to test how good our clustering algorithms perform we used the following evaluation function, which we will denote by **coverage**:

$$f = \frac{\sum_1^c |AND_i| \cdot |K_{\cap,i}|}{\sum_1^n |B_i|} \tag{3.9}$$

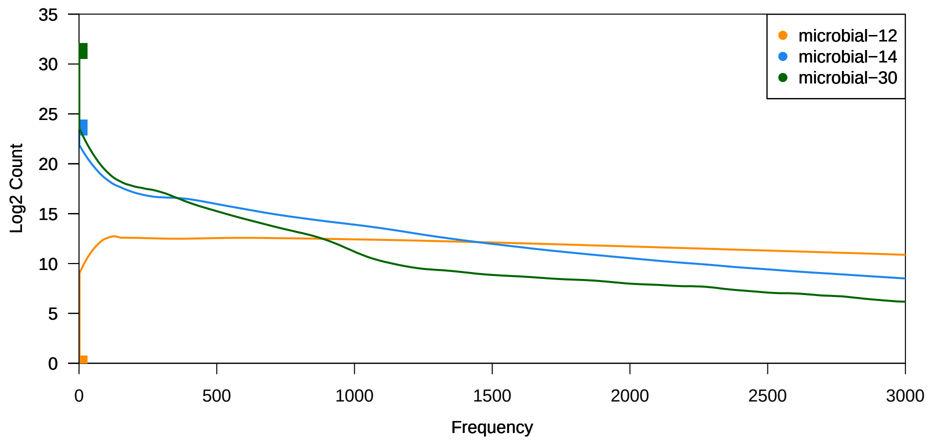
The coverage is used to evaluate the `AND`-phase of our clustering. First of all, it holds that $0 \leq f \leq 1$. Furthermore if the Bloom filters within each cluster are identical it follows that $f = 1$. And if the Bloom filters within the clusters do not share any bits it follows that $f = 0$. From these considerations one can see why the choice of f is meaningful for our problem. Consequently the coverage represents a very good indicator of how good the prunning quality of the AND/OR structure is going to be. We note that a very similar coverage function can be used for the `OR`-phase, which we omitted from the following experiments due to simplicity.

3.6.2 Frequency Distribution of k-mers

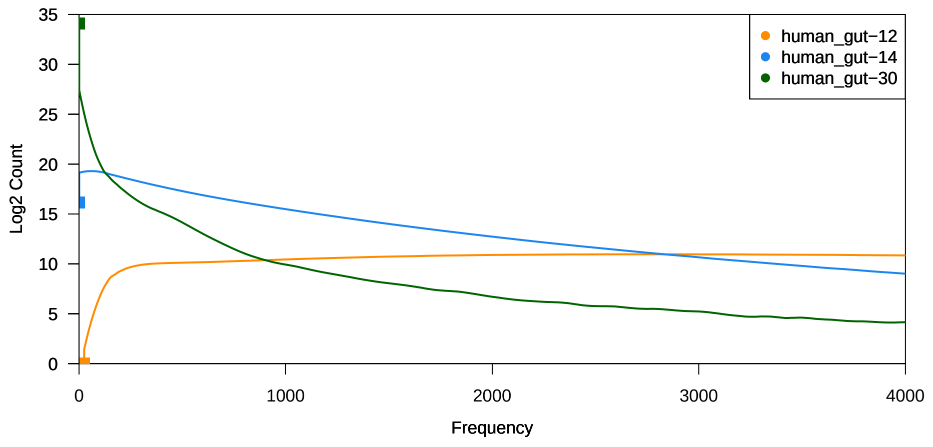
One thing that has been interesting to us while doing our experiments is the question about what the right k-mer size is. The original sources from which we obtained some of our data sets (**sbt**, **microbial**) did not provide a reasoning for their choice of the k-mer size. Neither did we find much useful information otherwise. There are two main factors that influence the choice of the k-mer size in our view. The first factor is inherit to the nature of approximate pattern matching. Choosing a k-mer size that is too small will lead to a drastic increase in the false positive rate of the queries. On the other hand if the k-mer size is too large we will not be able to have small queries. The second factor that can influence the choice of the k-mer size is the presence of noise in the data, which is due to errors in the sequencing process. The larger the k-mer size the more likely it is that some of its characters are erroneous. Having these considerations in mind we can see why it is not easy to choose a right k-mer size. A common rule of thumb seems to be to look for a k-mer size that produces a more or less uniform frequency distribution. This is a reasonable choice in so far that it prevents us from choosing a k-mer size that is too big or too small. Due to influence of noise the frequency distributions are likely to have a large spike in their low-frequency parts. In order to gain more insight, in Figure 3.3 we analyze the behavior of the k-mer frequency distribution for various choices of k. The frequency data was generated using the k-mer counting tool `jellyfish` [MK11]. Since there is a lot of variability in the frequency data we have smoothened it out by fitting the curves through the points using the `loess` function in `R` with a local windows size of $0.03 \cdot p$, where p is the total number of frequency points. We picked three representative k-mer sizes for each data set to illustrate the change in frequency distribution, $k \in \{12, 14, 30\}$.



(a) sbt



(b) microbial



(c) human_gut

Figure 3.3: k-mer frequency distributions for various choices of k.

We can see that for $k = 12$ the number of high-frequency k-mers is large relatively to the amount of low-frequency k-mers. This is not ideal because it leads to an increase in the false positive rate of our pattern matching and allows noise to blend in throughout the distribution. For $k > 12$ we can see a clear change. The low-frequency part shows a steep drop, which is most likely caused due to noise in the sequencing data. The high-frequency parts also start to experience a drop compared to $k \leq 12$. Which $k > 12$ is the best choice for which data is not really clear. It seems to be about finding the right balance between low-frequency k-mers falling into the noisy part of our distribution and the high-frequency part. Increasing that gap by using a larger k again leads to negative effects in our pattern matching due to larger influence of noise.

3.6.3 Clustering Sequencing Data

From each of our data sets we picked $n = 100$ random sequencing documents and clustered them into $c = 10$ clusters. In order to evaluate the quality of our clustering we used the coverage function f introduced above. For `KMeans` we performed $I_K = 20$ rounds while for `Ward` we initialized the heap by computing the distances between all points. We note that none of our clustering algorithms directly optimizes the criterion function f. In order to see whether a direct optimization of f leads to better clustering results, we have implemented a K-Means variant that tries to optimize f locally for each iteration of the algorithm. We denote this clustering approach by `OptKMeans`. For the **sbt** data set we used a k-mer size of $k = 20$, while for the other two data sets a k-mer size of $k = 31$ was used. In Table 3.1 we give an overview of the coverage, as well as the size of the smallest- and largest intersection cluster that were found in our experiments. In terms of coverage the best clusterings were obtained by `Ward`'s algorithm followed by `KMeans`. We have to note that a coverage of 30% may not be as good as it seems on the first sight. This can be seen from the size of the largest found intersection cluster, which in most cases contains almost all of our Bloom filters. The corresponding `AND` Bloom filter in such cases is empty, which means that almost all of our documents would not benefit from any prunning or potential memory savings. The relatively high coverage is contributed by relatively large Bloom filters that got clustered into singleton clusters – for which again we have no added value by introducing an `AND` Bloom filter. When it comes to execution time `Ward` was the fastest from our algorithms, leaving `BisectionalKMeans` by more than an order of an magnitude behind on the **sbt** data set, which can be seen in Table 3.2.

		Distance Measure									
Algorithm	**Dataset**	Euclidean		Hamming		W-Hamming		Cosine		Jaccard	
`Ward`	sbt	1/92	0.38	1/92	0.38	1/92	0.38	1/81	0.002	1/77	0.006
	microbial	1/94	0.34	1/94	0.34	1/94	0.34	1/93	0.013	1/37	0.001
	human-gut	1/93	0.18	1/93	0.18	1/93	0.18	1/88	0.0005	1/71	0.0002
`KMeans`	sbt	1/92	0.31	1/55	0.32	1/91	0.35	1/33	0.011	2/60	0.0022
	microbial	1/91	0.31	1/77	0.31	1/94	0.34	1/75	0.002	1/58	0.0001
	human-gut	1/93	0.16	1/93	0.17	1/82	0.1	1/65	0.0002	1/62	0.0006
`BisectionalKmeans`	sbt	1/81	0.19	1/45	0.11	1/90	0.32	1/68	0.018	1/81	0.0004
	microbial	1/75	0.21	1/73	0.17	1/89	0.29	1/36	0.018	1/54	0.014
	human-gut	1/93	0.15	1/93	0.15	1/58	0.08	1/71	0.0004	1/71	0.0005
`OptKmeans`	sbt	-	-	1/91	0.31	-	-	-	-	-	-
	microbial	-	-	1/65	0.26	-	-	-	-	-	-
	human-gut	-	-	1/93	0.17	-	-	-	-	-	-

Table 3.1: Cluster analysis of the `AND`-phase. The first column represents the fraction between the smallest and largest found intersection cluster. The second column represents the coverage.

Algorithm	**Dataset**	**Time[s]**	**Bloom Filter Size**
`Ward`	sbt	234.806	22405036765
	microbial	17.8093	3755021565
	human-gut	131.343	1732588147
`KMeans`	sbt	871.075	22405036765
	microbial	120.931	3755021565
	human-gut	592.96	1732588147
`OptKmeans`	sbt	1948.93	22405036765
	microbial	119.604	3755021565
	human-gut	716.016	1732588147
`BisectionalKmeans`	sbt	7031.16	22405036765
	microbial	1895.02	3755021565
	human-gut	5838.78	1732588147

Table 3.2: Execution time comparison of clustering algorithms.

3.6.4 Analyzing the Effects of Noise

The process of obtaining sequencing reads is not exact. Random noise gets introduced into the data due to the chemical process involved in generating the reads. A valid question that arises is to what extent does this noise influence the clusterability of our Bloom filters. To investigate this question we performed two experiments:

Generating Synthetic Data We generated $c = 5$ synthetic clusters with $n_c = 20$ documents per cluster. For each cluster we initially constructed a random

base string of length $l = 100000$ with the characters `{A, C, T, G}`. Each of the n_c documents within a cluster is a random mutation of this base string. The base string is mutated by changing each character with a probability of $m_r \in [0, 1]$. The characters are changed to a random nucleotide base letter. In Table 3.3 we illustrate the effect of the mutation rate m_r on the coverage of the obtained clusters. For the clustering of the data we used `K-Means` with Euclidean distance and a k-mer size of $k = 20$. We can see from the results how much impact noise can have in k-mer-based indexing. The problem is that a single faulty character introduces noise in up to k different k-mers. Having uniformly distributed noise is especially bad since it affects mostly distinct k-mers of a string. Unfortunately we do not have any information about how noisy our sequencing data actually is. This experiment rather shows us that even a small amount of constant noise can make the Bloom filters too noisy for our clustering application.

	$mr = 0.001$	$mr = 0.005$	$mr = 0.01$	$mr = 0.05$	$mr = 0.1$
coverage	0.7886	0.3370	0.1240	0.0068	0

Table 3.3: Effects of random mutation on the cluster coverage.

Removing Low-Frequency *k*-mers Low-frequency k-mers, especially those that occur only once, are likely introduced due to errors in the sequencing process. In the frequency plots introduced in this chapter, we can see that low-frequency k-mers are especially abundant. This means that the data we are trying to cluster is potentially extremely noisy. In order to investigate the influence of this noise we performed an experiment by deriving three new data sets from `sbt`. The data sets were obtained by filtering out all k-mers which occur with a frequency less than two, three and four respectively. The effects this had on the coverage quality of our clustering can be seen in Table 3.4. The results were obtained by using `K-Means` with Euclidean distance and a k-mer size of $k = 20$ If we compare these results to Table 3.1, where we did not filter out any k-mers from the data, we can see that our clustering performance initially drops. This would indicate that a large portion of the k-mers that occur only once are actually shared across the sequencing experiments. Filtering out increasingly more low-frequency k-mers shows to have a positive impact on our clustering coverage, overall confirming our assumption. It is unclear however, how much noise and how much actual data is removed by this process, which is something that requires further investigation.

	cutoff = 2	cutoff = 3	cutoff = 4
coverage	0.2238	0.3174	0.3540

Table 3.4: Effects of low-frequency k-mers on the cluster coverage.

4 Batched Queries

4.1 Overview

Queries in COBS are executed sequentially one after the other. This may be optimal for the execution time of a single query, but looking at $t \in \mathbb{N}$ queries $\{q_1, q_2, ..., q_t\}$ as a whole, one might do better. For each query that is executed sequentially a set of rows has to be read from the bit-sliced matrix:

$$R_i := \bigcup_{i=1}^{r} \{h_i(s) \bmod m \mid s \in M_k(q_i)\} \tag{4.1}$$

That is, we first read all rows indexed by R_1, then those by R_2 and so on. Depending on how many queries are executed and how large they are, the number of rows that are shared among the R_i increases. We can exploit that fact to our advantage. When batching, we compute the set of all rows:

$$R := \bigcup_{i=1}^{t} R_i \tag{4.2}$$

Instead of scanning the index multiple times, we only scan it once for all rows present in R. Since we are processing t queries simultaneously, we also need to update t different score arrays while reading the rows. That means, the expected speed up of this approach does not necessarily come from a lower number of operations executed.

Our batched queries are going to have a better cache performance and require less disk reads. The better cache performance comes from the fact that we are scanning through the bit-sliced matrix only once instead of t times. Even more importantly, in contrast to COBS we access the rows of the matrix in a sorted fashion. The decrease in disk reads only comes into play when the index does not fit into main memory. Let M be the size of the main memory in bytes, and B the block size of our disk. Further let the index size be $I = kM$, for $k \geq 2$. The sequential query execution will perform $\mathcal{O}(t \cdot \frac{I}{B})$ disk reads. In comparison, the batched approach will only require $\mathcal{O}(\frac{I}{B})$ disk reads. Batching does have a drawback in real time systems however. Depending on how large the batch size is, the query execution time will be delayed correspondingly. Thus designing a proper batching system is subjected to experimental evaluation.

4.2 Implementation Details

In this section we discuss a few implementation and design choices we had to face while modifying COBS to support batched queries:

Number of hash functions We restrict the number of hash functions used by COBS to $r = 1$. Since this is the default value which COBS uses anyway, it only represents a weak limitation. For $r = 1$ one can simply add up the row entries into the score arrays directly. While for $r > 1$, one has to compute the bit-wise `AND` from multiple row entries before the result can be added into the score array Figure 2.2. Which means that we need additional space to compute the bit-wise `AND` for each k-mer from all queries in the batch. The additional space needed for that would be in $\mathcal{O}(n \cdot \sum_{i=1}^{t} |q_i|)$. For moderately large batches this can get close to our index size. Due to this additional cost, we affirm that the choice of $r = 1$ is likely also the best when dealing with batched queries.

Sorted vs. Random access The order in which COBS reads the rows of the bit-sliced matrix is random. Accessing rows in a sorted fashion can be beneficial due to cache effects. This is especially true for smaller rows. In order to test whether we can get a performance gain through this, we have presorted the row indices before scanning the matrix. The sorting algorithm used for this is the In-Place Parallel Super Scalar Samplesort [AWFS17] by Axtmann et al.

SSE Instructions COBS uses the SSE2 instruction `_mm_add_epi16` in order to speed up the row processing. When computing the score array COBS is able to update 8 documents simultaneously this way. One drawback we noticed is that when two k-mers from a query hash into the same row, COBS would just read that row twice, and update the score array accordingly. Instead we store for each row a list of pairs containing the query id and the number of k-mers hashed into that row $\{(id_1, f_1), ..., (id_w, f_w)\}$. The score array of each query gets incremented as before with `_mm_add_epi16`. In addition we use `_mm_mullo_epi16` to premultiply the operand with the k-mer frequency.

4.3 Experimental Results

4.3.1 Setup

The same platform and data sets were used as in the previous Chapter 3.6.1.

4.3.2 Evaluation

We did a direct comparison between executing queries sequentially and in a batched fashion for $n = 10^4$ random sequencing experiments from the **microbial** data set. In Table 4.1 we show the execution times of both variants, while varying the number of queries performed. The size of the queries was equal to the k-mer size used for indexing the documents $k = 31$. Half of them were obtained by sampling the actual document set, while the other half was generated at random. We can see a performance increase of more than a factor of 10 which is mostly due to the better utilization of CPU and disk cache that comes from processing the rows in a sorted manner. The accessed data rows fit into RAM regardless of whether we used the compact or classic variant of COBS. Since designing an experiment where the rows do not fit into RAM is very artificial and unlikely to occur in real world scenarios, we choose not to conduct such an experiment.

	$queries = 10^3$	$queries = 10^4$	$queries = 10^5$	$queries = 10^6$
sequential	0.331595	3.18695	31.6946	325.397
batched	0.0417439	0.244372	2.18177	22.4698

Table 4.1: Execution times (in seconds) for sequential and batched querying.

5 Distribution

5.1 Motivation

Querying sequenced genomic data for particular occurrences becomes computationally more and more expensive as the amount of sequenced data nearly doubles every year. To handle the massive increase in computational and storage requirements, indexing such data efficiently requires a distributed approach. In the following we introduce our distributed version of COBS, which we will call for short by DCOBS.

5.2 Architecture

DCOBS uses a **master-slave** architecture. As such we distinguish between two types of compute nodes in our system: **query** (master) and **index** (slave) node. Ideally we would like to have a $n : m$ relation between the roles. For reasons of simplicity the current implementation only supports a $1 : m$ relation, but it should be easily extendable to support the former case. In the following we give a short description of the different roles in the system:

Query Node Responsible for orchestrating all operations on the distributed index. Accepts incoming user queries and distributes them among the index nodes for processing. Once the query results from all index nodes are received and merged together, it sends the result back to the user. It maintains a consistent view on the index and deals with fault tolerance.

Index Node Is essentially a wrapper around a COBS implementation. The index node accepts query requests from the query node and sends the computed result back to it. Each index node builds an index from a subset of the entire document collection. The documents a node is responsible for are stored in persistent memory on the node itself.

DCOBS Client A terminal client which connects to the query node. It can be used to dispatch queries to the query node and view the corresponding result. The terminal client will block until the result from the query node is received or a timeout has been reached.

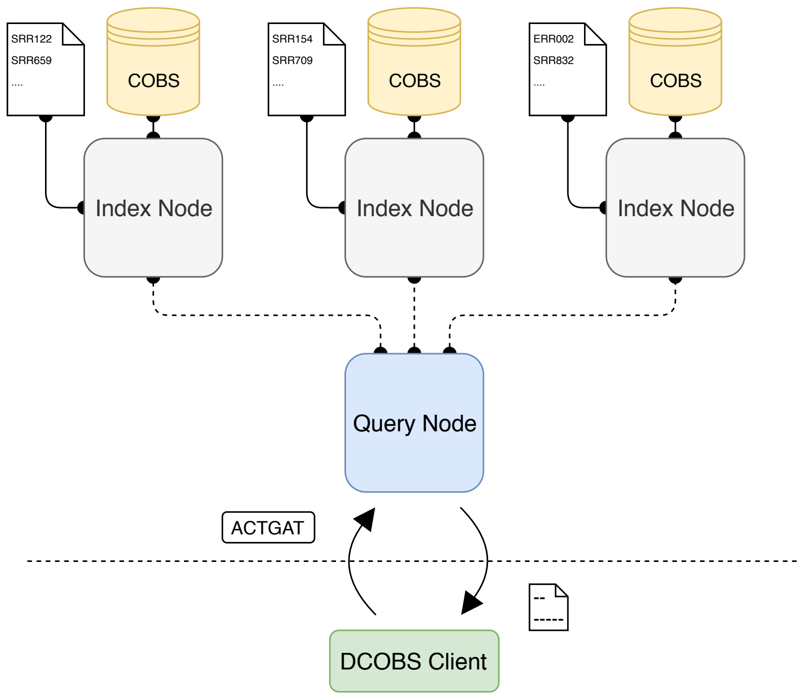


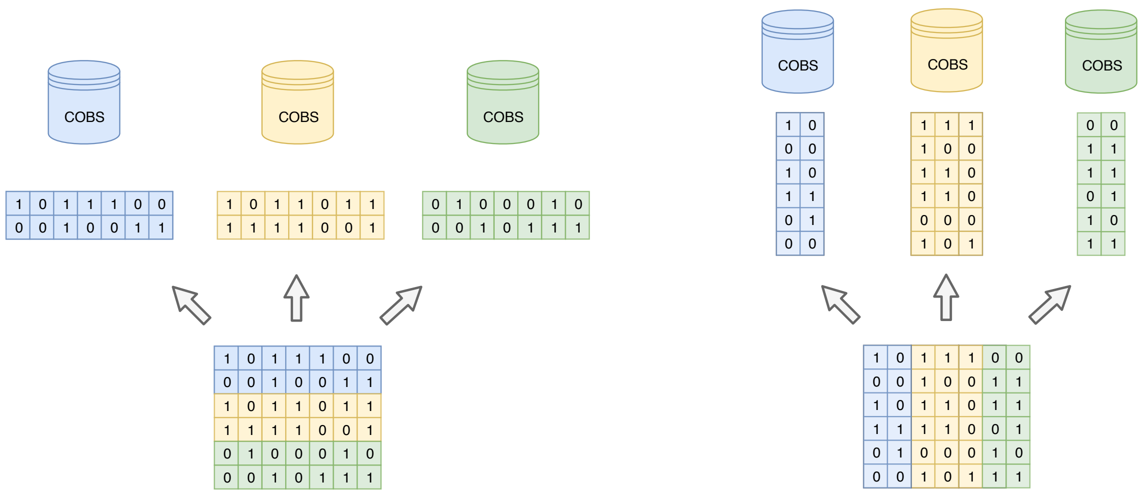
Figure 5.1: DCOBS architecture.

5.3 Implementation Details

The implementation was done in `C++` using the `Boost ASIO` library:
https://think-async.com/Asio.
In order to increase the portability of our code we decided to use the standalone version of the library, rather than the one integrated in `Boost`. However, we could not get rid of `Boost` completely. We did not find an easy way around using its serialization library, which we require in order to serialize the messages between different hosts. `ASIO` enabled us to implement the communications in a completely asynchronous manner. This means that no thread in the query or index node will be wasting computing resources waiting for a response. Both nodes are also multi-threaded and concurrent safe. As such they are able to deal with multiple incoming query requests simultaneously.

The distributed index is constructed without any interaction between index nodes. Each node has its own accession list of sequencing documents. The nodes read those lists and download the documents required to build their local index. This is done by using a custom `FTP`-client written in `C++`. After the downloads are complete, each node invokes COBS in order to build their index, and upon successful construction, they connect to the query node signaling their readiness to process queries.

A user connects to DCOBS by using a terminal client. It allows the user to query the system by either providing a raw query string or a text file. When the query node receives a query, it gets delegated to all connected index nodes. The query node



(a) Sharding by hash (b) Sharding by document

Figure 5.2: Sharding techniques.

needs to be able to later identify which incoming results from index nodes belong to the same query. To achieve this each request gets associated with a unique transaction id. The individual results from different index nodes get stored into a concurrent hash table. When the results from all index nodes are received they get merged using a `k-way-merge` algorithm in $\mathcal{O}(n \log k)$ time. Here n is the total size of the result set and k is the number of index nodes. The merging order is provided by the score array just like in COBS, which also gets sent back to the query node as part of the result set.

In addition to the query, users can provide three optional parameters: the k-mer size k, a threshold value θ and the number of documents in the result set t. This is the same as in the implementation of COBS, so the parameters are just delegated directly to it. The `k-way-merge` is adapted accordingly to stop after having found t documents, reducing the running time to $\mathcal{O}(t \log k)$.

5.4 Index Sharding

Index sharding refers to the logical partitioning of a single COBS index into multiple smaller indices, which in turn get distributed across the index nodes. The sharding should strive to achieve fast query times and robustness against system failure. There are two main ways to shard the COBS index across multiple nodes: by hash-space and document-space. In the following we address the advantages and drawbacks of them. Since sharding by document-space turned out to be more advantageous, it is the technique we employed in our implementation.

Sharding by hash-space Represents a horizontal partitioning of the bit-sliced matrix. Each index node is assigned one or more partitions. An incoming query to the query node is not forwarded directly to index nodes. Instead a Bloom filter containing all of its k-mers has to be computed first, which consequently gets partitioned along the same boundaries as the COBS matrix. The individual index nodes then receive corresponding parts of that Bloom filter, which they use to perform a local query. Although it provides good load balancing, because the entries in Bloom filters are randomly distributed, this approach has several drawbacks. Firstly, the same Bloom filter size has to be used across all documents, increasing the overall memory footprint of the index. Secondly, this approach has a poor fault tolerance. If one index node goes down, it eliminates at least one complete horizontal partition from our index. If we want to recover this lost data, we have to re-build the complete index using all documents from scratch, which is computationally very expensive.

Sharding by document-space Represents a vertical partitioning of the bit-sliced matrix. In other words, each index node is responsible for a subset of the entire document collection. This allows for queries to be directly forwarded to index nodes, without any prior processing by the query node. And because of this, index nodes are able to choose Bloom filter sizes to their liking, reducing the amount of wasted memory overall. The last advantage is that the failure of a single node is not as costly as when sharding by hash-space. We only have to re-build an index out of the documents that have been lost. Thus, the computational cost is proportional to the lost data and not the entire data collection in the distributed system.

5.5 Dynamic Index Updates

Once the distributed index is initially constructed, it should be possible to insert and delete documents from it. We will call such operations *dynamic index updates*. As we have already mentioned before, each index node in our system has its own list of unique document keys it is responsible for. Furthermore, we implemented an asynchronous event in our index node that periodically checks whether this list has been modified. If so, the node will start fetching new documents via `FTP` and trigger a dynamic index update.

The most trivial way to implement dynamic index updates is to re-construct the local index completely (with or without the documents in question). This however can get very expensive as the index size grows. We should not have to re-compute Bloom filters of documents that are not part of the update operation. We present two different ways to achieve this: by adapting compact `COBS` and by designing an alternative index representation.

5.5.1 COBS Compaction

The problem that arises with compact `COBS` lies in how the bit-sliced matrix is constructed. It gets build by pre-sorting the document collection by size and partitioning the resulting sequence into chunks $C_1, C_2, ..., C_p$ of fixed size W. For each chunk we build a bit-sliced matrix by choosing the Bloom filter size according to the largest document in the chunk, as shown in Figure 5.3.

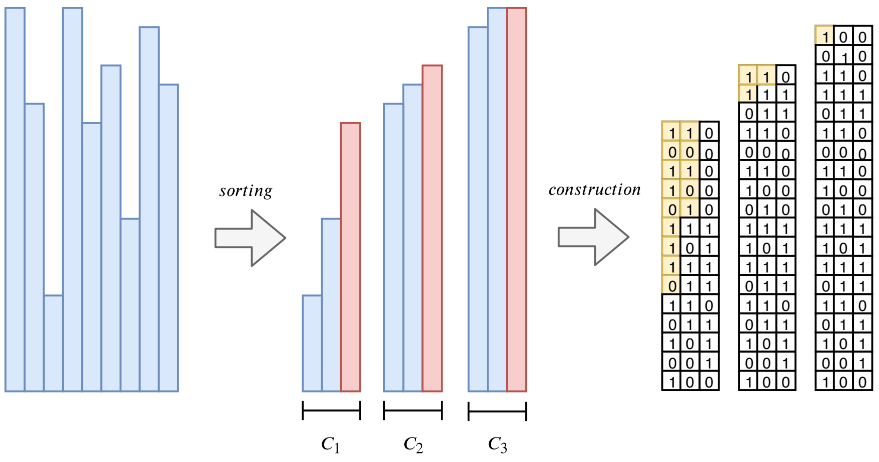


Figure 5.3: Compaction in COBS. Yellow areas of the index illustrate the wasted space by choosing the Bloom filter size according to the largest document.

In order to remove a document from the index we need to know in which chunk it resides. This can be answered in $\mathcal{O}(1)$ time by simple pre-computation. For a new document that is to be inserted in the index we have to choose into which chunk it should go. To maintain the sorted order of documents, we have to find the chunk whose largest document is as small as possible while still being larger than the new document. This can be done with binary search in $\mathcal{O}(\log n)$ time, where n is the number of documents in the index. Adding and removing documents will inevitably produce an in-balance in the index, because chunks will start to vary in size. A chunk with too many elements will start to look more and more like the classic COBS variant, wasting unnecessary space on large Bloom filters. Chunks that become to small will negatively effect the cache performance of the reads. To avoid this one can use a balanced binary search tree for each chunk to store the document sizes. Once there are more than $2 \cdot W$ documents in a chunk it gets split in two. And once there are less than $\frac{W}{2}$ documents in a chunk, we can join it with the previous chunk (following a potential split). Joining and splitting will require the re-computation of Bloom filters for all documents affected. In all other cases an insertion and deletion will only do a scan/copy of the index. Split and join on balanced binary search trees can be done in $\mathcal{O}(\log n)$ time [BFS16].

5.5.2 Fixed Interval Compaction

An alternative way to implement compaction is to use fixed-value chunks. In our implementation we define chunks in terms of powers of two. For each $i \in \mathbb{N}$ the chunk with index i will store all documents whose sizes fall in the range $(2^{i-1}, 2^i]$. The Bloom filter size for chunk i is determined by the largest possible document in that chunk, which is 2^i. This makes insertion and deletion especially easy, because Bloom filter sizes within a chunk never change. For this reason we also never have to recompute Bloom filters of documents that are not directly affected by an dynamic operation. This is important because in order to re-compute the Bloom filters one has to have access to the raw documents. Storing all of them permanently may not be a viable option in some scenarios due to large amount of data. Finding out in which chunk a document belongs to can be easily computed in $\mathcal{O}(\log n)$ time.

Because of its simplicity and the earlier mentioned advantages we opted to partition our index locally based on intervals of powers of two, rather than a fully sorted arrangement like in compact COBS. Depending on the distribution one compaction may deliver better results than the other. But overall the COBS compaction is expected to deliver better space savings due to its smaller chunks sizes. We can however give a bound on the worst case of our approach. If M is the space consumption of all Bloom filters, individually computed on a document basis, then the index partitioned in terms of powers of two will require at most $2 \cdot M$ space.

5.6 Experimental Results

5.6.1 Setup

We conducted our experiments on the high performance computing cluster `ForHLR II`, which is operated by the Steinbuch Centre for Computing (SCC) in Karlsruhe. The computing node consists of two Deca-core Intel Xeon processors E5-2660 v3 (Haswell) with hyper-threading enabled. Each core runs at a clock speed of 2.1 GHz and has 10×256 KB of level 2 cache and 25 MB level 3 cache. Each node has 64 GB of main memory and 480 GB of local SSDs.

All nodes in the cluster are connected through InfiniBand 4X EDR interconnect, which is characterized by a very low latency of 1 microsecond and a point to point bandwidth between two nodes of more than 6000 MB/s. We store our sequencing data on the distributed file system which is operated by Lustre. It allows for a read/write performance of 2GB/s per computing node and a total read/write performance of 50 GB/s. The computed indexes were stored on the local SSDs of each computing node.

5.6.2 Evaluation

We evaluated DCOBS by spinning up $N + 2$ computing nodes, which includes N index nodes, one query node and one client node. The data we used to construct our index was the same as in the previous chapter - $n = 10^4$ random sequencing experiments from the **microbial** data set. The data set was split up into N equal parts and each part was assigned to a corresponding index node. After construction the client node would generate $q = 10^5$ queries and dispatch them sequentially to the query node. The size of each query was equal to the k-mer size of $k = 31$, while half of them were obtained by sampling the sequencing documents, and the other half was generated at random. In Table 5.1 we report construction and query times for different numbers of N. The reported construction time is the maximum over all index nodes, while the reported query time is measured from when the first query is dispatched utill all the results were obtained from the query node. We can see that DCOBS scales very well, although we are not sure what caused the drastic increase in query performance from $N = 4$ to $N = 8$. One thing to note is that the communication overhead is also considerable. We start getting slightly better query performance than a single node for $N \geq 8$. Which is not bad given that the query performance of COBS is already super fast. At last we also want to mention the space consumption of the fixed interval compaction in comparison to the standard and compact variants of COBS. Even though the fixed interval compaction is considerably simpler than the compact variant of COBS, we can see in Table 5.2 that it was able to construct a bit-sliced matrix that is almost by factor of 2 smaller than the matrix produced by compact COBS.

	$nodes = 1$	$nodes = 2$	$nodes = 4$	$nodes = 8$	$nodes = 16$
build	10707.1	5070.3	2743.24	1691.57	1131.49
query	819.368	590.762	401.762	23.859	16.3288

Table 5.1: Distributed construction and query times (in seconds) for varying numbers of index nodes.

	standard `COBS`	compact `COBS`	fixed interval `DOBS`
Index Size in GB	451	34	18

Table 5.2: Comparison of index sizes for different compaction methods.

6 Conclusion and Future Work

We conclude this thesis by looking back at some of the outcomes and goals we have set at the beginning. Unfortunately we did not get good results by clustering Bloom filters obtained from sequencing experiments. Thus we did not manage to provide a justification for using the proposed AND/OR structure for prunning and compaction. The clustering problem was surrounded by many challenging aspects, from noisy and high dimensional data to properly parametrizing the clustering algorithms. Whether the clusterability changes with a larger amount of data or having a more flexible parametrization is left as an open question. Furthermore it is unclear how much of noise is present in the data, making it difficult to derive conclusions from it. Another different approach would be to look into whether the rows of the bit-sliced matrix share similarities and exploit those to reduce the size of the index - similar to the color-matrix used by De Brujin graphs. The Bloom filters in COBS also have the disadvantage of not being able to store the multiplicity of k-mers. Whether different encoding techniques could be used to modify the Bloom filters or perhaps a different structure altogether is left as an open question for the future as well. Things that ended up delivering promising results are the query batching and index distribution. Our batching mechanism managed to improve the query execution time by a factor of 10, while DCOBS showed to be very scalable - quickly overturning the query execution times of COBS. One thing that we did not have time to implement is a batching mechanism for distributed queries. Given our results it might be worth while to look into integrating this feature into DCOBS at some point. And lastly, another important characteristic that COBS is lacking is the ability to provide $query\ filters$. At the current state it is not possible to express queries to only consider sequencing experiments fulfilling certain characteristics, i.e. having the same taxonomic identifier, coming from the same species, sampling project and similar. Looking into how to build this into COBS is another interesting question open for future considerations.

Bibliography

[AGM$^+$90] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local aligment search tool," *Journal of molecular biology*, vol. 215, pp. 403–10, 11 1990.

[AV07] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '07. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035.

[AWFS17] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders, "In-Place Parallel Super Scalar Samplesort (IPSSSSo)," in *25th Annual European Symposium on Algorithms (ESA 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), K. Pruhs and C. Sohler, Eds., vol. 87. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 9:1–9:14.

[BBGI19] T. Bingmann, P. Bradley, F. Gauger, and Z. Iqbal, "Cobs: A compact bit-sliced signature index," in *String Processing and Information Retrieval*, N. R. Brisaboa and S. J. Puglisi, Eds. Springer International Publishing, 2019, pp. 285–303.

[BdBR$^+$19] P. Bradley, H. C. den Bakker, E. P. C. Rocha, G. McVean, and Z. Iqbal, "Ultrafast search of all deposited bacterial and viral genomic data," *Nature Biotechnology*, vol. 37, p. 152–159, 02 2019.

[BFS16] G. E. Blelloch, D. Ferizovic, and Y. Sun, "Just join for parallel ordered sets," in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '16. New York, NY, USA: ACM, 2016, pp. 253–264.

[BM01] E. Bingham and H. Mannila, "Random projection in dimensionality reduction: Applications to image and text data," in *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '01. New York, NY, USA: ACM, 2001, pp. 245–250.

[BOSS12] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya, "Succinct de bruijn graphs," in *Proceedings of the 12th International Conference on Algorithms in Bioinformatics*, ser. WABI'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 225–235.

[Bro97] A. Broder, "On the resemblance and containment of documents," in *Proceedings of the Compression and Complexity of Sequences 1997*, ser. SEQUENCES '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 21–.

[CHM19] R. Chikhi, J. Holub, and P. Medvedev, "Data structures to represent sets of k-long dna sequences," 2019.

[CL15] A. Crainiceanu and D. Lemire, "Bloofi: Multidimensional bloom filters," *Information Systems*, vol. 54, 01 2015.

[CLS$^+$18] C. E. Cook, R. Lopez, O. Stroe, G. Cochrane, C. Brooksbank, E. Birney, and R. Apweiler, "The european bioinformatics institute in 2018: tools, infrastructure and training," *Nucleic Acids Research*, vol. 47, no. D1, pp. D15–D22, 11 2018.

[ETOK18] K. Eschke, J. Trimpert, N. Osterrieder, and D. Kunec, "Attenuation of a very virulent marek's disease herpesvirus (mdv) by codon pair bias deoptimization," *PLOS Pathogens*, vol. 14, p. e1006857, 01 2018.

[HM18] R. Harris and P. Medvedev, "Improved representation of sequence bloom trees," 12 2018.

[Hua08] A. Huang, "Similarity measures for text document clustering," *Proceedings of the 6th New Zealand Computer Science Research Student Conference*, 01 2008.

[ICT$^+$12] Z. Iqbal, M. Caccamo, I. Turner, P. Flicek, and G. McVean, "De novo assembly and genotyping of variants using colored de bruijn graphs," *Nature genetics*, vol. 44, pp. 226–32, 02 2012.

[MBN$^+$17] M. Muggli, A. Bowe, N. Noyes, P. Morley, K. Belk, R. Raymond, T. Gagie, S. Puglisi, and C. Boucher, "Succinct colored de bruijn graphs," *Bioinformatics*, vol. 33, 02 2017.

[MK11] G. Marcais and C. Kingsford, "A fast, lock-free approach for efficient parallel counting of occurrences of -mers," *Bioinformatics (Oxford, England)*, vol. 27, pp. 764–70, 03 2011.

[Mug17] M. D. Muggli, "Vari," 2 2017. [Online]. Available: https://github.com/cosmo-team/cosmo/tree/VARI

[PAB$^+$18] P. Pandey, F. Almodaresi, M. Bender, M. Ferdman, R. Johnson, and R. Patro, “Mantis: A fast, small, and exact large-scale sequence-search index,” *Cell Systems*, vol. 7, p. 201–207, 06 2018.

[PB10] S. Perry and R. Beiko, “Distinguishing microbial genome fragments based on their composition: Evolutionary and comparative genomic perspectives,” *Genome biology and evolution*, vol. 2, pp. 117–31, 05 2010.

[PBJ$^+$18] P. Pandey, M. Bender, R. Johnson, R. Patro, and B. Berger, “Squeakr: An exact and approximate k -mer counting system,” *Bioinformatics (Oxford, England)*, vol. 34, pp. 568–575, 02 2018.

[RRR02] R. Raman, V. Raman, and S. S. Rao, “Succinct indexable dictionaries with applications to encoding k-ary trees and multisets,” in *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA ’02. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002, pp. 233–242.

[SHCM17] C. Sun, R. S. Harris, R. Chikhi, and P. Medvedev, “Allsome sequence bloom trees,” in *Research in Computational Molecular Biology*, S. C. Sahinalp, Ed. Springer International Publishing, 2017, pp. 272–286.

[SK15] B. Solomon and C. L. Kingsford, “Large-scale search of transcriptomic read sets with sequence bloom trees,” 2015.

[SK18] B. Solomon and C. Kingsford, “Improved search of large transcriptomic sequencing databases using split sequence bloom trees,” *Journal of Computational Biology*, vol. 25, 03 2018.

[SKK00] M. Steinbach, G. Karypis, and V. Kumar, “A comparison of document clustering techniques,” *Proceedings of the International KDD Workshop on Text Mining*, 06 2000.

[TGIM18] I. Turner, K. Garimella, Z. Iqbal, and G. McVean, “Integrating long-range connectivity information into de bruijn graphs,” *Bioinformatics (Oxford, England)*, vol. 34, 03 2018.

[Ukk92] E. Ukkonen, “Approximate string-matching with q-grams and maximal matches,” *Theor. Comput. Sci.*, vol. 92, no. 1, pp. 191–211, Jan. 1992.

[WEG87] S. Wold, K. H. Esbensen, and P. Geladi, “Principal component analysis,” 1987.

[Wil88] P. Willett, “Trends in... a critical review recent trends in hierarchic document clustering: A critical review,” 1988.

[YLL$^+$18] Y. Yu, J. Liu, X. Liu, Y. Zhang, E. Magner, E. Lehnert, C. Qian, and J. Liu, "Seqothello: querying rna-seq experiments at scale," *Genome Biology*, vol. 19, 12 2018.

[ZB08] D. Zerbino and E. Birney, "Velvet: Algorithms for de novo short read assembly using de bruijn graphs," *Genome research*, vol. 18, pp. 821–9, 06 2008.