Karlsruhe Institute of Technology

# Engineering a Compact Bit-Sliced Signature Index for Approximate Search on Genomic Data

Master's Thesis of

Florian Gauger

14 February 2018


| | |
|---|---|
| Supervisors: | Prof. Dr. rer. nat. Peter Sanders |
| | Dipl.-Inform. Dipl.-Math. Timo Bingmann |
| | Dr. rer. nat. Simon Gog |
| Advisors: | Dr. Zamin Iqbal |
| | Phelim Bradley |

Department of Informatics
Karlsruhe Institute of Technology

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

**Karlsruhe, 14 February 2018**

.....

(**Florian Gauger**)

**Abstract**

In recent years there has been an exponential increase in publicly available DNA data. This data encodes a huge amount of information on the ancestry of organisms, their evolution, and their abilities and properties. Until recently, it was not possible to search this data in its entirety. Providing online search facilities would enable research into epidemiology, basic science of disease, and antimicrobial resistance. Such search would be as enabling for microbial genomics as natural text search engines are for many everyday tasks.

Recently the Bitsliced Genomic Signature Index (`BIGSI`) [BdBR+17] was introduced allowing the search on all viral and bacterial genomic sequence reads in the European Nucleotide Archive (ENA) [TAA+17]. As of December 2016, the data set consists of 447,833 genomes using 170 TiB of space. However, longer queries can take up to half an hour on a high performance machine.

In this thesis we introduce the Compact Bit-Sliced Signature Index (`COBS`), a new index variant with significantly improved space and time requirements. In our empirical evaluation we show that execution time is improved by two orders of magnitude while space requirements are cut in half. For longer queries the execution time can be reduced from half an hour to under five seconds. Additionally, we enrich the query results with quality information which can be used for ranking the result set.

Several techniques are introduced which make these improvements possible. A compact data structure layout is used to achieve large decreases in index size with only a marginal increase in execution time. Additionally, algorithm engineering techniques are employed to show why the choice of index parameters made by `BIGSI` was not optimal and how optimal parameters can be chosen. This insight is used to significantly reduce execution time. Next, a new set of equations is introduced to enrich the result with quality information which gives the user more insight into the composition of the result and could be used to increase effectiveness in the future. Lastly, we add several low-level optimizations, exploiting modern hardware, which further lower execution time.

To evaluate the techniques, several experiments are conducted which give insight into index construction, execution performance, space-time trade-offs, false positive distributions and the impact of the engineering optimizations. Multiple areas where future work can be conducted are introduced. Finally, the steps required for the creation of a dynamic online index which supports sharding on multiple machines are described.

While `COBS` was constructed to address the challenges of searching large genomic collections, the main findings can be applied to a wide range of problems.

`COBS` is written in `C++17` and is available at `https://github.com/devgg/cobs`.

## Zusammenfassung

Die Menge an öffentlich zugänglichen DNA Daten stieg in den letzten Jahren exponentiell. Diese Daten enthalten Informationen zur Abstammung von Organismen und deren Evolution, Fähigkeiten und Eigenschaften. Bis vor kurzem konnten diese Datenmengen nicht in ihrer Gesamtheit durchsucht werden. Auf dem Gebiet der Mikrobiellen Genomforschung würden entsprechende Suchoptionen der Forschung neue Möglichkeiten bieten, z.B. im Bereich der Epidemiologie, bei der Erforschung von Resistenzen gegen Antibiotika und der Grundlagenforschung.

2017 wurde der Bitsliced Genomic Signature Index (`BIGSI`) [BdBR+17] vorgestellt, der es ermöglicht, alle bakteriellen und viralen Gensequenzen (447.833 Genome, 170 TiB) des European Nucleotide Archive [TAA+17] zu indizieren. Allerdings haben längere Anfragen Antwortzeiten von bis zu einer halben Stunde.

In dieser Arbeit stellen wir den Compact Bit-Sliced Signature Index (`COBS`) vor. In unser empirischen Analyse zeigen wir, dass `COBS` die Ausführungszeit um zwei Größenordnungen verringert und der Platzbedarf um die Hälfte reduziert wird. Dies bedeutet, dass längere Anfragen weniger als fünf Sekunden benötigen. Zusätzlich ergänzen wir die Ergebnisse der Anfragen mit Qualtiätsinformationen, die zur Bestimmung der Reihenfolge der Ergebnisse benutzt werden können.

Diese Verbesserungen werden durch mehrere Techniken ermöglicht. Um den Index kompakt zu speichern, wird ein neues Datenstruktur Layout vorgestellt. Dieses Layout wird benutzt, um die Index Größe zu verringern, ohne dabei viel Ausführungszeit einzubüßen. Wir wenden Algorithm Engineering Techniken an um zu zeigen, welchen Einfluss die Index-Parameter auf Größe und Performance haben. Diese Erkenntnisse werden verwendet, um die Ausfürungszeit zu verringern. Es werden Formeln vorgestellt, mit deren Hilfe die Ergebnisse der Anfragen verbessert werden können. Zusätzlich werden mehrere hardwarenahe Optimierungen eingesetzt, um die Ausfürungszeit weiter zu senken.

Um Methoden und Optimierungen zu evaluieren wurden mehrere Experimente durchgeführt. Unter anderem wurden Index Konstruktion, Ausführungszeit, verschiedenen Tradeoffs und Optimierungen der Implementierung evaluiert. Zusätzlich werden mehrere Bereiche, in denen weiter Forschung notwendig ist, aufgezeigt, sowie die nötigen Schritte für die Erstellung eines dynamischen Index dargelegt.

Obwohl `COBS` entwickelt wurde, um die Herausforderungen der unscharfen Suche auf genomischen Daten zu lösen, können die Erkenntnisse und der Index in vielen Bereichen der Wissenschaft genutzt werden.

`COBS` wurde in `C++17` geschrieben und ist unter `https://github.com/devgg/cobs` verfügbar.

# Contents

# 1. Introduction

The ability to process and understand large collections of data is essential for many fields of science. Example applications are large-scale physics experiments and autonomous driving. Recent advances in genome sequencing technology have caused an exponential increase in raw genomic sequence read data [CAA+12]. Projections expect the amount of data contained in public sequencing databases such as the European Nucleotide Archive (ENA) to double every 12 to 18 months [SLF+15]. This data encodes a huge amount of information on the ancestry of organisms, their evolution, and their abilities and properties. Providing online search facilities for bacteria and viruses would enable research into epidemiology, basic science of disease, and tracking of "mobile elements" that jump between genomes and species. For example, the problem of antimicrobial resistance, where bacteria develop the ability not to be killed by specific drugs, is a global threat to public health. This occurs either through random mutations or by passing entire genes between different bacteria on long mobile vectors called plasmids. It is therefore of huge value to be able to search for these mutations, genes and plasmids. Indeed such search would be as enabling for microbial genomics as natural text search engines are for many everyday tasks.

Nearly all current approaches (`SBT`, `SSBT`, `AllSomeSBT`, `Mantis`) are focussed on queries on highly repetitive collections (such as human transcriptomes), which occupy a microscopic corner of the billions of years of evolution (and therefore diversity) in the tree of life [HBA+16]. However, a large portion of the data available in the public genomic archives is viral and bacterial data that, due to higher mutation rates, shorter generation times, and billions of years of evolution, does have a significantly lower similarity. Genomic data can be stored in two formats. In its raw format, it is stored in many small sequences generated by an error-prone experimental procedure that oversamples from the genomes that are in the physical sample under test. In its assembled format, the underlying genome is inferred from these sequences - this data varies wildly in accuracy, and is lossy when raw input data comes from more than one genome. Since assembly is a very costly process, most publicly available data is unassembled. The ENA contains 170 TiB of raw viral and bacterial data as of December 2016.

One of the most widely-used tools in all of science is the Basic Local Alignment Search Tool (`BLAST`) [AGM+90]. It first enabled sequence-level searches over genomic databases. Many improvements have been made to the basic concept, resulting in numerous `BLAST` variants that improve speed and sensitivity. While `BLAST` and its variants have been widely used, they rely on the genomic data to be assembled. Additionally, `BLAST` and its extensions do not scale well as the amount of data increases. Hence, the amount of data currently present in public archives cannot be accessed using the traditional methods. As a result, new tools had to be created to cope with the exponential influx of raw genomic sequence reads.

Many of the existing information retrieval data structures do not function well due to the nature of the dataset and the queries. Full-text indexing data structures such as the Burrows-Wheeler transform [BW94] or the FM-index [FM05] currently are not able to handle data of this scale. Indices known from web search such as the inverted index are not suitable for the approximate nature of the queries and the lack of "words" in the data. Approximate queries are a necessity since the raw genomic reads can contain sequencing errors and since some applications require the ability to search for similar sequences. Furthermore, queries of tens of thousands of base pairs in length should be supported. These challenges led to the creation of several novel tools.

In 2016 Solomon and Kingsford introduced the Sequence Bloom Tree (`SBT`) [SK16]. This data structure and the associated algorithms solved the challenges and were used to enable the search on 2652 human transcriptomic sequences. To allow for approximate pattern matching a $q$-gram (also known as $n$-gram or $k$-mer) based approach was employed. This also helps in reducing the original data by removing duplication generated by the sequencing technique. Additionally, a hierarchy of compressed Bloom filters was used to reduce the size of the data structure. Later the Split Sequence Bloom Tree (`SSBT`) [SK17] and the AllSome Sequence Bloom Tree (`AllSomeSBT`) [SHCM17] were introduced, improving on the `SBT`.

`Mantis` is another large-scale sequence-search index [PAB+17]. It follows a fundamentally different approach than `SBT` or its variants. `Mantis` decreases the index construction time, index size, and execution time. It uses a counting quotient filter [PBJP17] to store a mapping from $q$-gram to a document vector, resulting in various improvements over the Sequence Bloom Tree and its variants. However, one challenge introduced by the nature of viral and bacterial data is not addressed by any of these tools. The relatively low similarity in the genomic sequences of viruses and bacteria results in rapid increases in index size when new data is added. This is not the case for human sequence data since these tools compress the index by utilizing similarities within the data set. Since both problem domains, human genomes and viral and bacterial genomes, are of vital importance to many research fields the creation of specialized tools is of value. Due to the rapid growth of the data in both domains, this is especially important.

Recently, bit-sliced signatures were rediscovered as a viable approach to web search. `BitFunnel` [GHL+17] is an index that addresses several of the challenges traditionally associated with signature-based indices. Simultaneously, bit-sliced signatures were also

used in the Bitsliced Genomic Signature Index (BIGSI) [BdBR+17]. This index was created for raw genomic data and does not rely on similarity within the data. BIGSI was used to index the raw viral and bacterial genomic data available in the ENA. As such, BIGSI was the first index that was viable for data with the properties described previously.

## 1.1 Contributions

We introduce the Compact Bit-Sliced Signature Index (COBS) which outperforms BIGSI in build time, index size and query performance. We show how the index size can be reduced by sacrificing query performance and how the result can be enriched with quality information.

COBS solves the challenges of approximate pattern matching on large collections of genomic data with relatively low similarity. Its implementation outperforms the implementation of BIGSI by up to two orders of magnitude. Additionally, the space required to index all the viral and bacterial raw sequence reads of the ENA is reduced by 58% (849 GiB). COBS enables approximate search on millions of raw genomic sequences with queries of tens of thousands of characters. Queries of these lengths have various applications such as searching for a plasmid. Also, the contributions of this publication are applied to genomic data specifically, they build upon the foundations laid by BitFunnel and can be used in other scientific fields as well.

There are five main contributions. The index size is reduced by introducing a compact data structure. It is shown how the variation of index parameters affects index size and query performance. The result of a query is enriched and several practical engineering improvements are presented. Lastly, several ideas are proposed to show how the index can scale dynamically to multiple machines.

**Compact Index** BIGSI grows linearly with the largest document contained in the index when new documents are added. This leads to a lot of wasted space when the size of the documents in the dataset varies. COBS introduces a new data structure which scales linearly with the *mean document size*. As a result, for very large collections each document is stored in a compact way. This comes at a cost of performance. However, the trade-off between index size and query performance can be made gradually. A large portion of the index size reduction can be achieved with only a minor increase in query time.

**Algorithm Engineering** In recent publications the choice of index parameters was made to optimize for space usage [BdBR+17]. However, the performance impact that these parameter choices have was not made clear. We show how the choice of parameters influences both index size and query performance. Afterwards, we employ these findings to construct the index on the ENA dataset.

**Result Enrichement** To decrease the index size BIGSI uses a set membership data structure. COBS uses the same underlying structure which leads to result sets that can contain false positives. We provide a formula which can be used to mitigate the influence of these false positives. By using this method the result can be enriched

with quality information that could be used to increase the effectiveness of the index. This formula can also be used to improve the results of `BIGSI`.

**Optimizations** The reference implementation of `BIGSI` is written in `Python`. We provide a `C++` implementation of `COBS` at `https://github.com/devgg/cobs`. This leads to a performance increase over `BIGSI` of up to two orders of magnitude. Multiple additional performance optimizations have been made. The disk is accessed with raw direct I/O, bypassing kernel space buffering. Additionally, Single Instruction Multiple Data (`SIMD`) and Open Multi-Processing (`OpenMP`) are used to speed up query processing.

**Dynamic Index** We propose multiple ideas on how to make the index dynamic. That is to say, how to enable insertion, deletion, and modification. Additionally, we show how the index can be efficiently sharded onto multiple machines. Further research is needed to evaluate and build upon these ideas.

# 2. Fundamentals

In this chapter, the fundamentals needed to understand the Compact Bit-Sliced Signature Index are established. The notation used in this thesis is introduced in Section 2.1. Next, the problem of approximate pattern matching is examined in Section 2.2. Section 2.3 outlines related work in the areas of Information Retrieval (IR) and bioinformatics. We finish this chapter by formulating the main problem which is solved in the remainder of the thesis (Section 2.4).

## 2.1 Notation

The notation used in the thesis is the common notation used in the field of Stringology, which differs from notations used in bioinformatics.

A *string* $S$ is a sequence of characters. The finite ordered set of all characters $\Sigma$ is called *alphabet*. Since we will be focusing on genomic data our most used alphabet is defined as the set of all base pairs $\Sigma = \{\text{A, C, G, T}\}$. A *q-gram* (less commonly *n-gram* or *k-mer*) is a substring of length $q$. Let $G_q(S)$ be the *q-gram profile* of $S$ [Ukk92]. That is to say, $G_q(S)$ is defined as the vector of all $q$-grams contained in $S$. Equation 2.1 shows the relation between $|G_q(S)|$ and $S$.

$$|G_q(S)| = |S| - q + 1 \tag{2.1}$$

A *document* $d$ is a vector of strings such as the contents of a website or the reads of a genome. Multiple documents form a *collection* $D$. To formulate the problem further notation is introduced in Section 2.4.

## 2.2 Approximate Pattern Matching on Genomic Datasets

Understanding the properties of the dataset is the first step in deciding which approximate pattern matching approach to take. Our goal is to index all viral and bacterial sequence

reads available in the European Nucleotide Archive (ENA) [TAA+17]. This dataset was first used in [BdBR+17] and contains 447,833 documents consisting of sequence reads of lengths between 35 and 1000 characters. The total dataset has a size of over 170 TiB. In contrast to the human pan-genome, the pan-genome of bacteria is much larger [LG09]. Additionally, the dataset consists of genomes of different species. As a consequence, the similarity between genomes in the dataset is comparatively lower.

The raw data consists of genomic sequence reads. Therefore, each document contains a great deal of duplication (generally substrings are oversampled at least 30 times). In addition the sequences have an error rate of approximately $0.5\% - 3\%$ depending on the sequencing technique [RRC+13]. Assembling the genomes is not optimal for this application as the input data is not guaranteed to be from one genome, and assembly programs are heuristic, error-prone and slow. To clean the data, the built-in error-cleaning algorithm of `McCortex` [ICT+12, ITM12] was used.

The problem of approximate pattern matching has been the topic of many publications [Kru16]. For large datasets a pre-calculated index structure is necessary. Due to the properties of the dataset, we decided to create an index based on a $q$-gram vocabulary. This is advantageous for several reasons. As discussed previously, the dataset contains a lot of duplication which is eliminated by the use of $q$-grams. In other words, duplicate $q$-grams can be discarded since it is very unlikely that the same $q$-gram is contained multiple times in a genome (for sufficiently large $q$). Furthermore, the queries that the index will need to handle might be several tens of thousands of characters long. As a consequence, more complex approximate string matching measures like the edit distance are not computationally feasible and, as was shown in [BdBR+17], not necessary for real-world applications.

For the above reasons, our problem is transformed from approximate pattern matching to the problem of finding set intersections. That is to say, for each document we want to find the number of $q$-grams both contained in the document and the query. For this reason, we transformed each of the 447,833 documents containing raw sequence reads into a set of $q$-grams. This reduced the space requirements for our initial data from 170 TiB to 10 TiB. To reduce redundancy, each $q$-gram was transformed into its canonical form. In other words, we use the lexicographical smaller of the forward and the reverse complemented $q$-gram sequence as described in [OTM+16].

To choose the $q$-gram size $q$, a trade-off must be made between *uniqueness* and *sensitivity*. Longer $q$-grams are more likely to uniquely define a position in the genome and can be used to identify a specific mutation. At the same time, identifying a single-base change using very long $q$-grams is likely to fail due to other nearby mutations. We choose $q = 31$ which should result in sufficient uniqueness while still providing enough sensitivity for real-world applications. Iqbal et al. [ICT+12] provide further insight into this trade-off.

## 2.3 Other Approaches

The problem of finding documents relevant to a query is important in many different areas of scientific research. Traditionally documents were scanned sequentially. Knuth, Morris,

and Pratt [KMP77] and Boyer and Moore [BM77] introduced algorithms to efficiently scan text. Both these algorithms scale at least linearly with the length of the documents. For larger datasets, this quickly becomes prohibitive in terms of time. As a result, precomputed index structures were introduced [FO98]. In this section we give an overview of existing approaches related to the $q$-gram-based pattern matching described in Section 2.2. First, we examine the approaches already present in the field of bioinformatics. Afterwards, we look at the indices classically used for Information Retrieval.

### 2.3.1 Genomic Sequence-Search Indices

Since the introduction of the Basic Local Alignment Search Tool (`BLAST`) in 1990 [AGM+90] a wide range of tools has been created to manage and organize genomic data. `BLAST` was the first tool that allowed efficient comparison of biologic sequence information. It is used to search for a query sequence in one or multiple target sequences. In a first step, a heuristic is used to find similar sub-sequences. This step is called seeding. Afterwards, the similar sub-sequences created during seeding are extended in both directions. During the extension a score is computed for each alignment. If this score becomes too low the alignment is excluded from the result. Several improvements have been made to both speed and sensitivity by tools such as `DIAMOND` [BXH15] or the BLAST-like Alignment Tool (`BLAT`) [Ken02]. However all these tools require the collection to only contain assembled genomic sequences. Since over 80% of sequences available in archives such as the ENA are unassembled, new tools had to be created to make this data accessible.

In 2016 the Sequence Bloom Tree (`SBT`) was introduced by Solomon and Kingsford [SK16]. This data structure uses a binary tree of compressed *Bloom filters* to efficiently store the $q$-grams of a collection. A Bloom filter is a set membership data structure which can be used to store a set of terms. When querying, the filter might produce a false positive but not a false negative. In other words, the filter might report that a term that was not inserted is present but might not report that an element that was inserted is absent.

In the `SBT` each document is represented as one Bloom filter at a leaf of the binary tree. An inner node contains all $q$-grams present in its subtree. When querying this index the tree can be traversed from the root to find all documents containing at least a certain percentage of the $q$-grams. A subtree can be excluded when this threshold of $q$-grams is not present in the Bloom filter at its root. Due to its construction algorithm, where nodes are combined from the leaves down until the root node is created, all Bloom filters have to have the same size. This size depends on the number of unique $q$-grams present in the collection.

For a small number of documents with high similarity the number of unique $q$-grams is relatively small. However, for millions of documents containing viral and bacterial data with comparatively low $q$-gram sharing, the size of the index would become prohibitive [BdBR+17]. The more recent Split Sequence Bloom Tree (`SSBT`) [SK17] and AllSome Sequence Bloom Tree (`AllSomeSBT`) [SHCM17] share the same limitation. These indices improve on the `SBT` with a similar high-level approach. The main idea is to use two Bloom filters for the internal nodes of the tree. The first Bloom filter only stores the $q$-grams

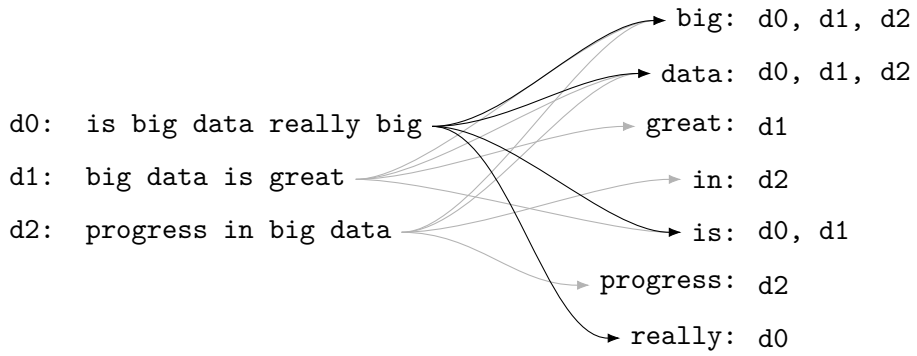


Figure 2.1: The relation between the collection $D = \{\texttt{d0}, \texttt{d1}, \texttt{d2}\}$ and an inverted index built on this collection.

present in all children of the node. These $q$-grams do not need to be stored in any of the children, thereby reducing redundancy. The second Bloom filter stores the rest of the $q$-grams that only appear in some of the children. To reduce the total space requirements both `SSBT` and `AllSomeSBT` cluster similar documents into the same subtree. Although this technique reduces the space requirements when compared to the `SBT`, the index size during construction would still remain prohibitive [BdBR+17].

`Mantis` is another large-scale sequence-search index [PAB+17]. It uses a counting quotient filter [PBJP17] to store a mapping from $q$-gram to a document vector, resulting in various improvements over the Sequence Bloom Tree and its variants. A counting quotient filter is a multiset membership data structure. `Mantis` uses this structure to store the locations of compressed bit vectors. To query the index the location to the bit vector associated with each $q$-gram is extracted from the counting quotient filter. Multiple $q$-grams can point to same bit vector which stores the documents that contain these $q$-grams. This reduces both index size and execution time. However, similar to the other approaches `Mantis` relies on the high $q$-gram sharing of the human genome and scales superlinearly with the number of unique terms. Therefore it is not suitable for the properties of the ENA dataset. To our knowledge, the only index that can be built on a similar dataset and run on consumer grade hardware is `BIGSI` [BdBR+17]. An introduction to `BIGSI` is provided in Section 2.3.3.

### 2.3.2 Inverted Index

Inverted indices are one of the main tools for Information Retrieval in web search [FBY92, BCC16]. An inverted index is a mapping from every term in a collection to a list of documents containing it. This allows for quick lookups of documents containing specific words. Today there exist highly engineered implementations with techniques such as early termination and top-$K$ retrieval [WMB99, ZM06].

A collection $D$ can be seen as a mapping from documents to words. Given the document, it is straightforward to get the list of words contained in it. To allow for the reverse operation, the mapping needs to be inverted. That is to say, to find all documents which contain some words one needs to have a mapping from word to document. This is the definition of an inverted index and is illustrated in Figure 2.1.

On the left side a collection $D = \{\texttt{d0}, \texttt{d1}, \texttt{d2}\}$ is shown. This mapping from document to words is inverted and the resulting inverted index is shown on the right. An inverted index can be used to efficiently find the documents containing a set of terms. For example, the documents ($\{\texttt{d0}, \texttt{d1}\}$) containing both the terms `big` and `is` can be found quickly by looking at the relevant rows in the inverted index. However, the properties of our dataset make the inverted index a suboptimal choice.

The inverted index grows linearly with the number of unique words contained in the documents of the collection. This is fine for web search where the number of terms is relatively small compared to the number of documents. For example, $10^{12}$ documents containing $10^8$ unique terms for Google's index [BdBR+17, BP12]. This is not the case for documents containing $q$-grams based on viral and bacterial sequence reads where the pan-genome might be of essentially infinite size [LG09]. There, the number of possible $q$-grams is $4^n$ which for $q = 31$ is on the order of $10^{18}$. For the ENA dataset with less than $10^7$ documents there are $10^{12}$ unique $q$-grams. Therefore we consider other approaches that do not scale linearly with the number of unique terms.

### 2.3.3 Signature Files

In the context of Information Retrieval a signature is a way to represent a document $d$, typically by a bit pattern that can be created in various ways [FC84]. The most well-known approach to implement signatures is via Bloom filters. Bloom filters were introduced by Burton H. Bloom in 1970 [Blo70] and have found application in many areas of computer science since then. A Bloom filter is a set membership data structure. We will introduce Bloom filter-based signatures more thoroughly in Section 3.1.1. Tarkoma et al. provide a good overview of current Bloom filter variants in their survey from 2012 [TRL12].

Signature files were first introduced in 1984 by Faloutsos et al. [FC84]. Signature files can be seen as an index containing signatures for each document in a collection. Signature files have long been known as a method of Information Retrieval in large collections [FC84]. Zobel et al. deemed signature files inferior to inverted index based approaches in almost every category measured [ZMR98].

Originally, signature files stored the signatures of the documents sequentially. One early improvement was to use signatures of the same size for each of the documents and to transpose the resulting matrix [Won85]. The resulting index is called Bit-Sliced Signature Files (`BSSF`) [FC88] but can be seen as an inverted index based on signatures. This index, which we call Bit-Sliced Signature Index, is described in Section 3.1.2.

Recently the interest in signature files was reinvigorated by `BitFunnel` [GHL+17]. In this publication, a set of techniques was introduced to address the challenges classically associated with signature files. The first technique is called *frequency-conscious signatures*. It adjusts the number of hash functions for each term individually. The number of hash functions is based on the frequency of the term's occurrence in the collection. Terms that appear less frequently are assigned more hash functions. This ensures that the number of false positives is more consistent and also implies a smaller memory footprint.

*Higher rank* rows are the second technique proposed. It builds upon the idea of blocked signature files [SDKR87, KSDR90, ZMR98] to reduce execution time. Each of the terms is hashed into multiple signatures of different sizes. This allows for early pruning when one of the higher (shorter) signatures reports a negative result. To automatically configure the number of hash functions and the number and kind of blocks for each term, `BitFunnel` uses a performance model. This model is used to perform a constrained optimization.

As described earlier, the signatures of all documents need to have the same size in the Bit-Sliced Signature Index. Depending on the dataset, this can result in a lot of wasted space and large differences in false positive rates. Solutions for this problem have been proposed [ZMR98]. However, each of them comes with several drawbacks making them not generally applicable. The third technique introduced in the `BitFunnel` paper is *sharding* by document length. Each machine stores one index with documents of similar sizes. Therefore, little space is wasted and the false positive rate is similar between any two documents. When working on the scale of web search, sharding by document length is an improvement at no additional cost since the index needs to be sharded on many machines anyway. In Section 3.1.4 we will introduce a solution that addresses this challenge on a single machine, thereby creating a compact index.

Bradley et al. introduced `BIGSI` [BdBR$^+$17], a `BSSF`-based search index enabling the indexing of millions of viral and bacterial genome sequence reads. This index was the first that applied the concept of bit-sliced signatures in the context of large-scale genomic sequence-search.

## 2.4 Problem Definition

As discussed in Section 2.2, the problem of approximate string matching can be solved in multiple different ways [Kru16]. When taking into account the specific properties of viral and bacterial genomic data, a $q$-gram based approach is the most viable. Note that we ignore the frequency of $q$-grams in the original documents since $q$ was chosen in a way that $q$-grams should be unique within documents. Therefore, the problem of approximate sequence-search in genomic data is transformed to the problem of counting the number of $q$-grams of the query that appear in a document. As a result, the original query string is divided into its $q$-gram profile. Let $Q = \{t_1, ..., t_{|Q|}\}$ be a bag of terms called a *query*, and let $m = |Q|$ be the number of terms in a query. Then the $q$-gram genomic search problem is defined in Problem 1.

**Problem 1.** Return the number of terms of the query $Q$ that are also contained in document $d$, the result may contain false positives. Let $r = R(d, Q)$ be the result, $r_{\text{tp}}$ be the number of true positives and $r_{\text{fp}}$ be the number of false positives. Then:

$$r = r_{\text{tp}} + r_{\text{fp}} = \big|\{t \in Q : t \in d\}\big| + r_{\text{fp}}$$

$R(d, Q) = r$ and $R(D, Q, \epsilon) = \{(i,\ R(d_i, Q)) \mid d_i \in D \wedge R(d_i, Q) \geq \epsilon m\}$ are the results of a query on a document and a collection respectively. The boolean conjunction represented by $\epsilon$ is an optional term. The current index implementation does not contain optimizations based on $\epsilon$. Therefore, we assume $\epsilon = 0$ for the rest of this work. The Bitsliced Genomic

Signature Index solves Problem 1. We introduce `COBS` which provides vast improvements in build time, space requirements and query time.

# 3. Compact Bit-Sliced Signature Index

This chapter gives an in-depth explanation of the Compact Bit-Sliced Signature Index (`COBS`). Section 3.1 gives a thorough explanation of the index and the main scientific improvements made in this thesis. The engineered implementation details are outlined in Section 3.2. Lastly, Section 3.3 shows how the index can scale to multiple machines to accommodate the rapid increase in genomic data caused by the advances in genome sequencing.

## 3.1 Design

Although `COBS` was designed to enable large-scale interactive searching of viral and bacterial data, the index excels in all scenarios where data with similar properties is used. For this reason and for better comprehensibility, the introduction of the data structure is written in terms of web search. We first give an overview of Bloom filter-based signatures. Afterwards, we describe the index and its parameters. Finally, we introduce equations to help the user understand the quality of the results.

### 3.1.1 Signatures

Inverted indices scale poorly when there are a lot more unique terms than documents. This is the case for the ENA dataset since the length of the $q$-grams is chosen such that $q$-grams are unique. Signatures [FC84] are a way to remedy this problem. As discussed in Section 2.3.3, we use Bloom filters to implement signatures. A Bloom filter-based signature of a document is a bitstring of $w$ bits. Each of the words of the document is hashed with $k$ hash functions and the corresponding bits are set to 1. Figure 3.1 illustrates this process. Note that $v$ represents the number of unique terms in a document, as inserting the same element multiple times does not change the signature.

Each of the $v = 4$ terms is hashed $k = 3$ times and the corresponding bits in the signature are set. Notably the number of set bits of a signature does only depend on the signature size $w$, the number of hash functions $k$, and the number of unique terms $v$. To check

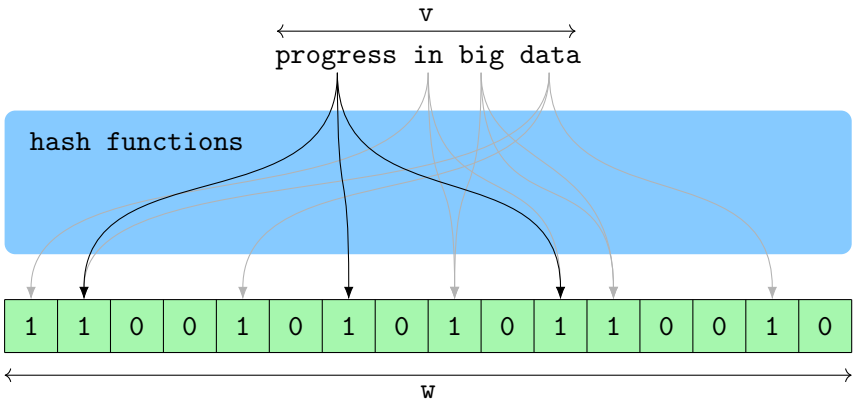


Figure 3.1: Creation of a Bloom filter-based signature with $k = 3$ and $w = 16$ of the document $d = \{\texttt{progess}, \texttt{in}, \texttt{big}, \texttt{data}\}$.

whether a word is contained in the signature, the hashing process is repeated and the corresponding bits are checked. The membership query returns true if all the bits are set and false otherwise. It is easy to see that this can result in false positives but not in false negatives.

Figure 3.2 illustrates the different results of a membership query. The signature is the same as in Figure 3.1. The first query checks whether `progress` is contained in the signature and therefore the document. Since all the corresponding bits are set, a positive result is reported. This is a true positive since the term was contained in the original document. The second membership test illustrates a negative result. It also shows that multiple hash functions can point to the same element. The third example illustrates a false positive result. The word `great` is not part of the document but the membership test result is positive.

### 3.1.2 Bit-Sliced Signature Index

An index can be built on the collection $D$ by creating one signature for each of the documents. These signatures have to have the same size so that a query always accesses the same bits in each document. This property can be used to minimize the number of random accesses by storing the arrays transposed (bit sliced) [Won85]. The index described in this section is very similar to `BIGSI`. To query the index, $km$ rows need to be read. The complete query process can be seen in 3.3.

The process starts off by hashing each of the terms in $Q = \{\texttt{data}, \texttt{is}, \texttt{great}\}$ with $k = 3$ hash functions. Afterwards, the respective rows are read from disk. Then the rows corresponding to each of the query terms are combined using the logical `AND` operation. In the resulting $3 \times 3$ matrix each row resembles one of the query terms and each column one of the documents. A set bit signals that the term is contained in the document. The result $R(D, Q) = \{2, 3, 1\}$ is created by summing up the rows. Finally, we sort the result so we can return a list of (document name, result)-pairs ordered by the number of matched terms.

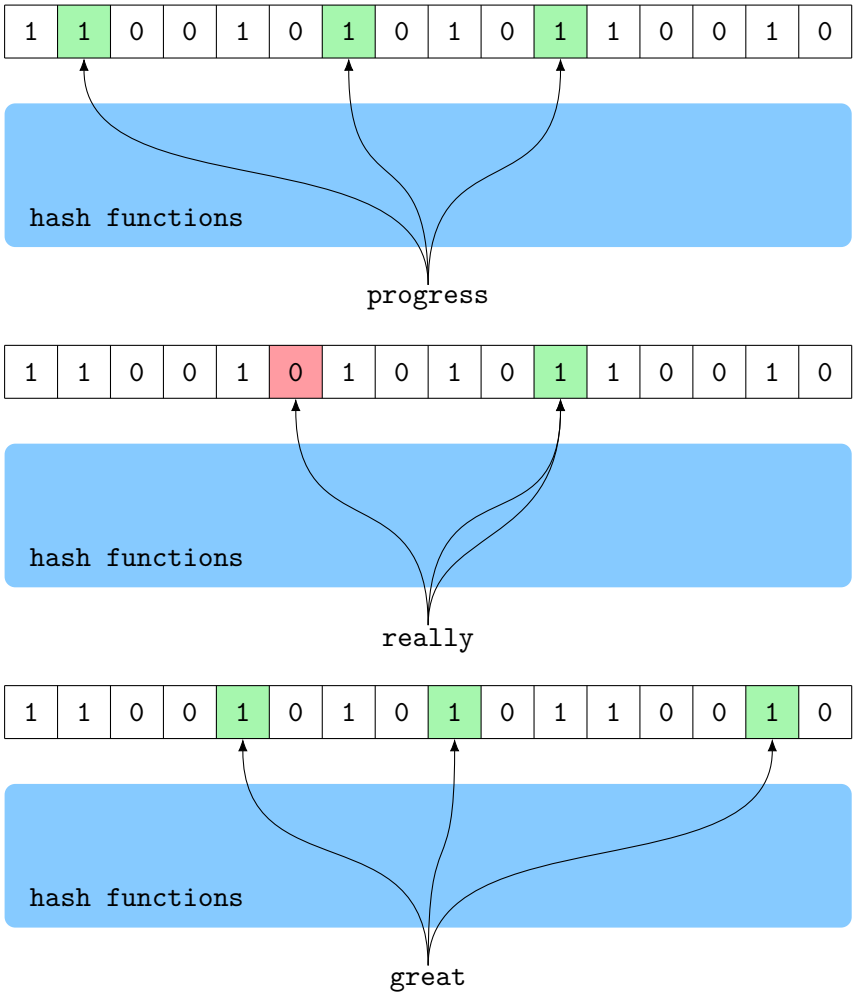


Figure 3.2: Three membership queries on the Bloom filter-based signature created in Figure 3.1. The first query yields a true positive, the second a (true) negative and the third a false positive.

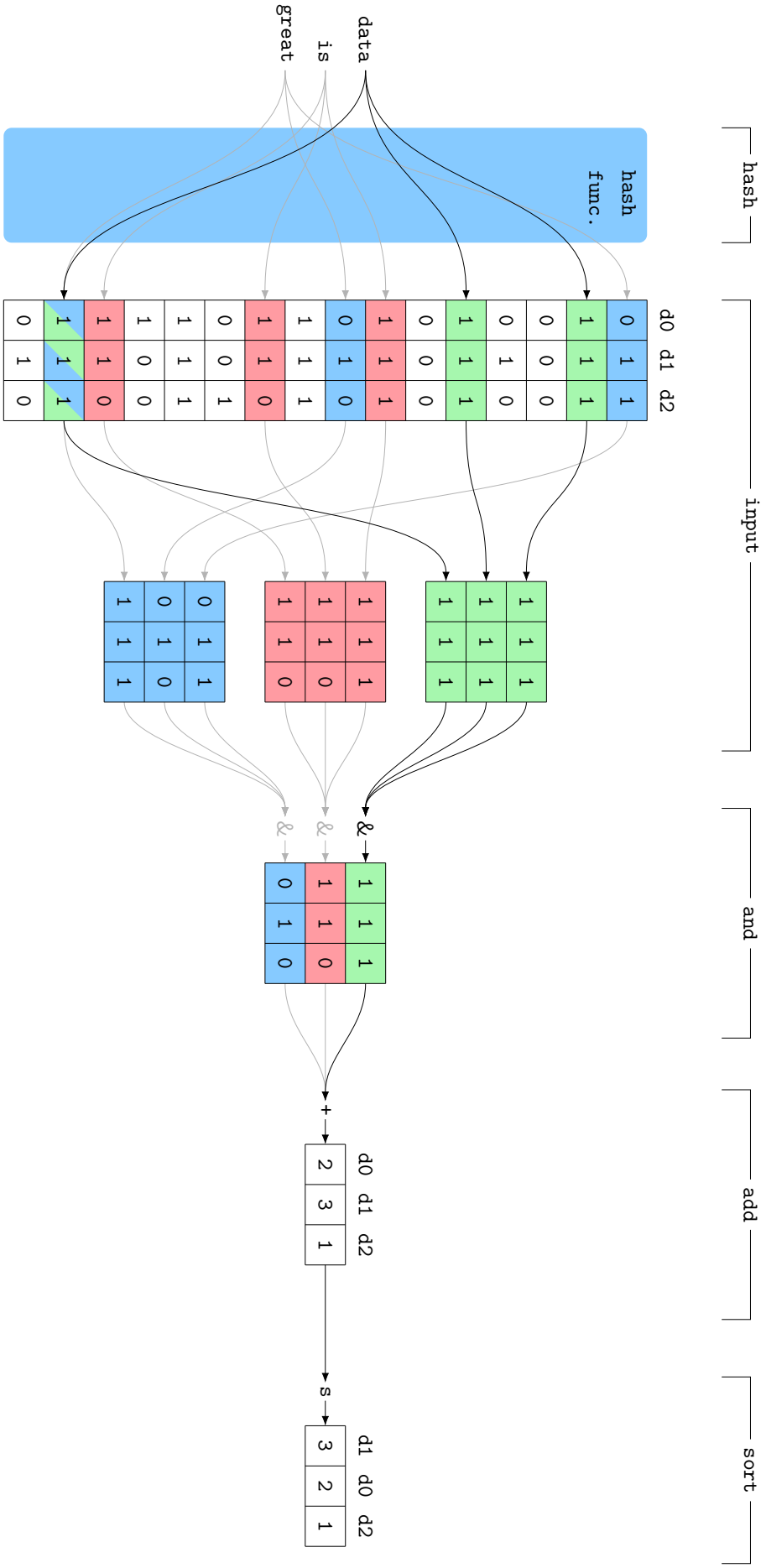


Figure 3.3: Query process of the Bit-Sliced Signature Index.

### 3.1.3 Signature Parameters

The false positive rate $p$ of a signature is determined by the number of (unique) inserted elements $v$, the number of hash functions $k$ and the size of the signature $w$ [ZMR98]. After $v$ elements are inserted into the signature, the probability that a specific bit is not set is $(1 - \frac{1}{w})^{kv}$. The equations to calculate the false positive rate and the ratio of $w$ to $v$ can be seen in Equation 3.1 [TRL12].

$$\begin{aligned} & p = \left(1 - \left[1 - \frac{1}{w}\right]^{kv}\right)^k \\ \Rightarrow \quad & p \approx \left(1 - e^{-kv/w}\right)^k \\ \Leftrightarrow \quad & 1 - p^{1/k} \approx e^{-kv/w} \\ \Leftrightarrow \quad & \ln(1 - p^{1/k}) \approx -\frac{kv}{w} \\ \Leftrightarrow \quad & \frac{w}{v} \approx -\frac{k}{\ln(1 - p^{1/k})} \end{aligned} \tag{3.1}$$

The choice of signature parameters influences false positive rate, index size, and access time. In previous work [BdBR+17], the "optimal" signature parameters were calculated with Equation 3.2 [BM04].

$$\begin{aligned} w &= \frac{v \ln p}{(\ln 2)^2} \\ k &= -\frac{\ln p}{\ln 2} \end{aligned} \tag{3.2}$$

This approach optimizes the index size for a given false positive rate. However, the access time is not taken into account. Figure 3.4 shows the relation between the false positive rate and the ratio of $w$ to $v$. The x-axis corresponds to the index size and the y-axis to the false positive rate. The number of hash functions correlates to the access time. For false positive rates greater than 0.39, using only one hash function is optimal in both query time and space. For lower false positive rates, a trade-off needs to be made between the ratio of $w$ to $v$ and the number of hash functions. The first four operations in Figure 3.3 scale linearly with the number of hash functions and therefore the total access time scales linearly with $k$.

For the false positive rate, we follow the logic in [BdBR+17], stating that 0.3 should be sufficient for indexing all ENA datasets. However, since we do not use Equation 3.2 but instead consider the trade-off that is illustrated in Figure 3.4, we choose $k = 1$, thereby using 11% more space but cutting access time in half.

### 3.1.4 Compact Index

The Bit-Sliced Signature Index described previously relies on signatures of equal size for all documents. This ensures that the input step only reads $km$ rows, thereby minimizing

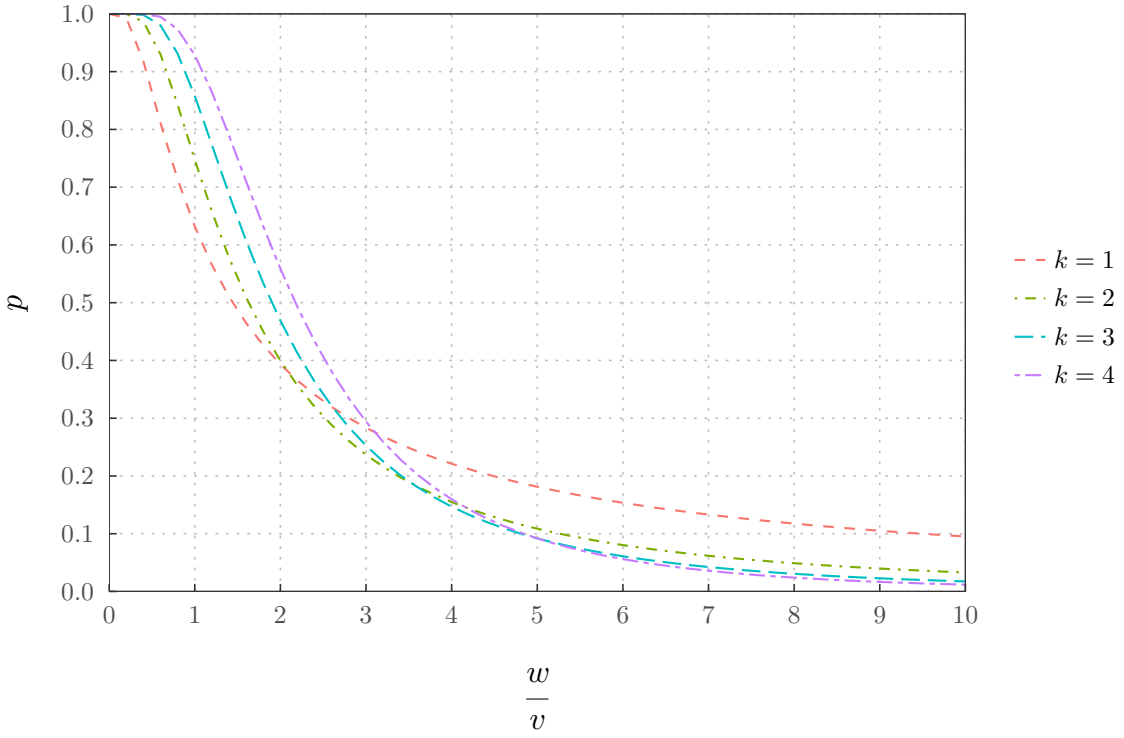


Figure 3.4: The false positive rate $p$ for different ratios of the signature size $w$ to the number of inserted elements $v$. Increasing the number of hash functions $k$ raises execution.

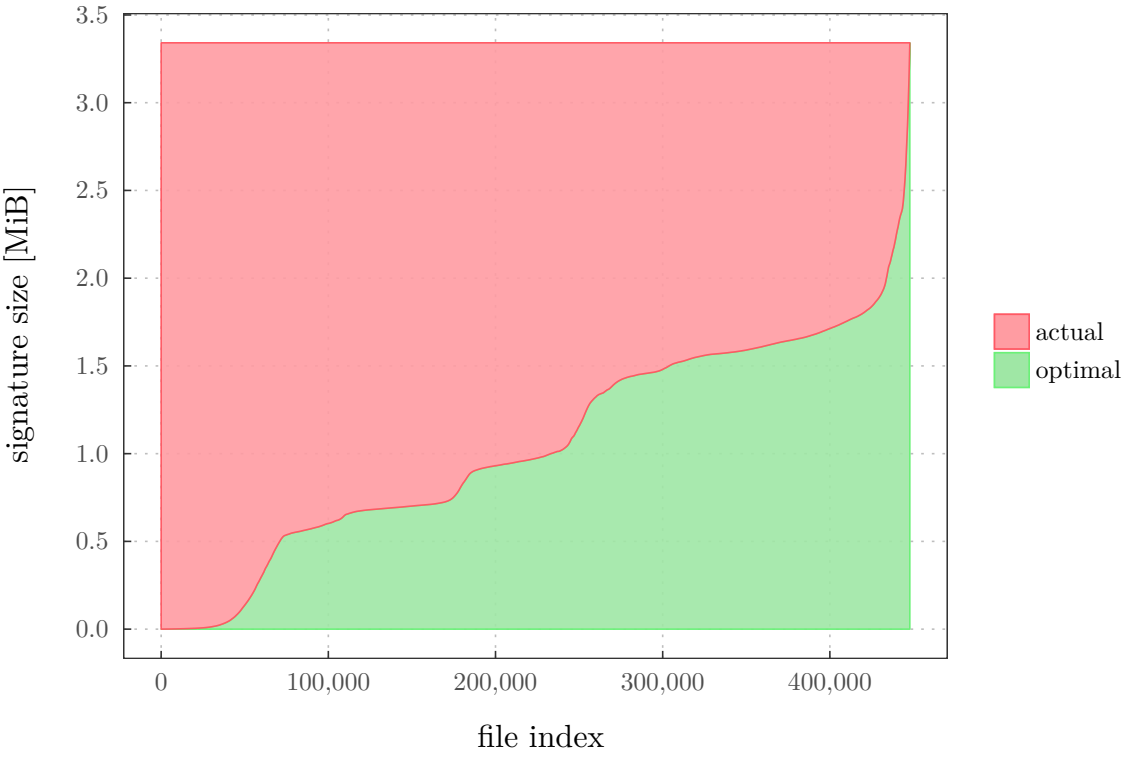


Figure 3.5: Comparison between the actual and the optimal signature size for each of the 447,833 documents in the Bit-Sliced Signature Index.

the number of random accesses. However, depending on the dataset, this also creates vast differences in false positive rate for the documents. Figure 3.5 shows the optimal and the actual signature size for each of the 447,833 documents in the Bit-Sliced Signature Index. The optimal size is the size that corresponds to a false positive rate of 0.3 per document.

For the ENA dataset, this discrepancy results in the Bit-Sliced Signature Index index using 218% more space (1461 GiB) than optimal (459 GiB). To minimize the space required we propose to use multiple subindices with different signature sizes instead of one index with one signature size. The resulting signature sizes for 14 subindices can be seen in Figure 3.6. Notably, the actual signature size is very close to the optimal signature size for most of the documents.

Let $h$ be the number of subindices. For $h = |D|$ each document has a signature of optimal size so no additional space is needed (ignoring space needed to index the subindices). As a result, the query $Q$ would produce $km|D|$ random disk accesses reading one bit each, which is not feasible. By using two subindices and therefore doubling the number of random accesses, the space overhead can already be reduced to 105%. For hard disk drives (HDD), choosing the number of subindices is a trade-off between space required and disk-seek overhead. The disk access pattern for $h = 1$ and $h = 3$ can be seen in Figure 3.7. When only one subindex is used with $k = 1$ each query term produces one long disk read. By using three subindices, the total index size can be reduced but the number of disk reads triples. Furthermore, the length of the reads is reduced by two thirds. That is to say, by

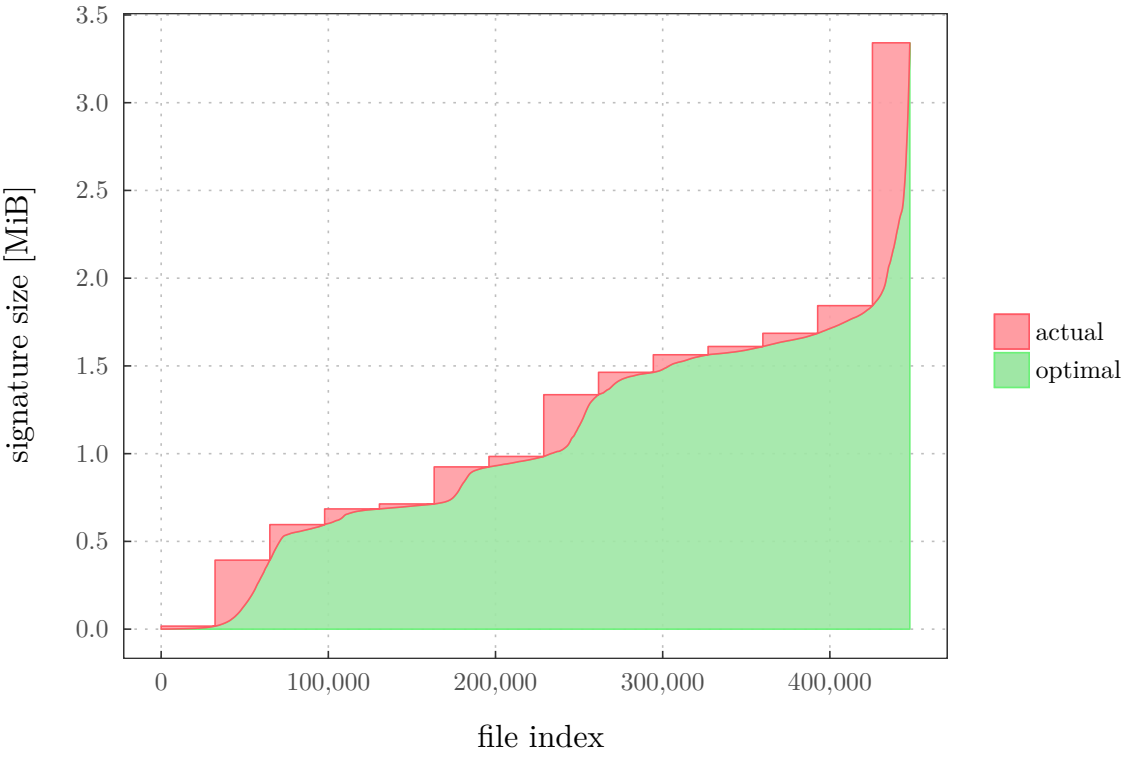


Figure 3.6: Comparison between the actual and the optimal signature size for each of the 447,833 documents in the Compact Bit-Sliced Signature Index.

using more subindices, the number of random reads increases and the length of these reads decreases. Overall, the same amount of data is read.

We call this index with multiple subindices Compact Bit-Sliced Signature Index (`COBS`). The index is compact since the number of subindices ($h$) increases as new documents are added to the index. Therefore, by adding new elements, the percentage of the space that is wasted is reduced. In contrast, the Bit-Sliced Signature Index uses the same amount of space for each signature, regardless of the length of the document. As a result, the required space does not depend on the size of the documents but only on the size of the largest document supported.

In 2009 Chen et al. [CKZ09] showed that SSDs exhibit 31 times greater bandwidth for small random reads (4 KiB) than HDDs. Two years later Chen et al. [CLZ11] demonstrated that for sufficient queue depth the bandwidth of 4 KiB random reads is more than half of the bandwidth of 256 KiB sequential reads. Based on these findings and our measurements in Section 4.6, we propose a compact index optimized for solid-state drives (SSDs). We use a subindex size of 8 KiB as a trade-off between index space and execution performance. On the ENA dataset the space overhead is reduced from 1002 GiB to 153 GiB (85%) when compared to the Bit-Sliced Signature Index. This comes with a 38% drop in performance. Alternatively, a subindex size of 26 KiB can be used resulting in a 518 GiB (52%) reduction in space overhead and only in a 3% drop in performance. Section 4.7 provides more insight into the trade-offs of different subindex sizes.


Figure 3.7: Access pattern for indices with one and three subindices and a single hash function.

Section 3.2 describes how direct I/O is used to access the raw block device of the disk. This requires the data to be read to be aligned at the SSD page boundaries. A page is the smallest unit of an SSD and is typically 4 KiB in size. As a result, the subindex size needs to be a multiple of the page size. Additionally, the last subindex which might not be completely filled needs to be padded to the next page boundary. In Section 3.3 this padding is used to grow the index dynamically.

### 3.1.5 Result Enrichment

The result of a query on one document is the sum of the number of query terms that are contained in the document (true positives) and the false positives introduced by the Bloom filter-based signature. As a result, the user has no intuition as to the quality of the returned result (i.e. how many of the reported $q$-gram matches were actually contained in the document and how many were added due to the Bloom filter-based signature). In this section, a set of equations is introduced to calculate the probability distribution for the number of true positives actually contained in the original document. This greatly increases the quality of the results.

We show how the probability distribution can be calculated for a document $d$ when the corresponding signature has a false positive rate of $p$, $k$ hash functions are used, and the length of the query is $m$. Since we have no prior knowledge of the data and the queries, we take a frequentist approach and assume that there is no randomness describing the number of true positives $r_{\text{tp}}$. We are trying to find the maximum likelihood for $r_{\text{tp}}$, given $p$, $k$, $m$, and the result of a query $r$. To make the quality of the returned results visible, we introduce Problem 2.

**Problem 2.** Return the discrete probability distribution $f(r_{\text{tp}}; r, m, p, k)$ for the number of true positives $r_{\text{tp}}$ contained in the result $r$ for a query of length $m$, a false positive rate $p$, and $k$ hash functions.

The query contains $m$ terms, $r_{\text{tp}}$ of which are contained in the original document. The remaining $m - r_{\text{tp}}$ terms are not contained in the document and may result in a negative

or in a false positive. For a term to create a false positive, all $k$ hash values need to point to set bits in the signature. The most likely value for $r_{\text{tp}}$ can be calculated by examining the ratio of (false) positive to negative membership test results of the remaining $m - r_{\text{tp}}$ terms.

Each of the membership tests of the terms not contained in the original document is an independent Bernoulli trial, for which the probability of success (a false positive) is $p^k$. Let $X_i = \{0, 1\}$ be the random variable describing the $i$th of these trials. Then the set $\{X_i \mid 1 \leq i \leq m - r_{\text{tp}}\}$ is mutually independent. Universal hashing ensures that two $q$-grams that are close to each other in the query or the original document do not map to the same hash value more frequently. Let $X = \sum_{i=1}^{m-r_{\text{tp}}} X_i$ be the random variable for the total number of false positives.

Since $X$ is the number of successes of independent Bernoulli trials the probability distribution for $X$ is a binomial distribution. For a given $r$ we can calculate the probability distribution by using one value of the binomial distribution for each possible value of $r_{\text{tp}}$.

Let $B(z_1, z_2, z_3)$ be the probability mass function of the binomial distribution, where $z_1$ is the number of successes, $z_2$ is the number of trials, and $z_3$ is the success probability. Then the formula for calculating the probability distribution for the number of true positives can be seen in Equation 3.3.

$$\begin{aligned} f(r_{\text{tp}}; r, m, p, k) &\propto \mathcal{L}(r_{\text{tp}} \mid X = r_{\text{fp}}) \\ &= B(r_{\text{fp}},\ m - r_{\text{tp}},\ p^k) \\ &\overset{\text{def}}{=} \binom{m - r_{\text{tp}}}{r_{\text{fp}}} (p^k)^{r_{\text{fp}}} (1 - p^k)^{m-r} \end{aligned} \tag{3.3}$$

The probability distribution for $r_{\text{tp}}$ is proportional to the likelihood that $r_{\text{tp}}$ has a certain value given a fixed value for $X$. This likelihood can be described with the binomial distribution. Figure 3.8 illustrates the intuition behind this equation. The illustration shows the possible results for a query on one document with $m = 8$ and $r = 3$. There are four possible values for $r_{\text{tp}}$. In the third row the likelihood of $r_{\text{tp}} = 1$ is displayed. For this value to be true there must have been two false positives and five negatives. This result is the most probable. Note that the values $(0.17, 0.30, 0.32, 025)$ are not normalized but are proportional to each other. In other words, $r_{\text{tp}} = 1$ is almost twice as plausible as $r_{\text{tp}} = 3$.

The normalized likelihood is the probability distribution of Problem 2. The distribution is illustrated in Figure 3.4.

$$f(r_{\text{tp}}; r, m, p, k) \overset{(3.3)}{=} \frac{B(r_{\text{fp}},\ m - r_{\text{tp}},\ p^k)}{\sum_{i=0}^{m} B(r - i,\ m - i,\ p^k)} \tag{3.4}$$

Figure 3.9 shows the true positive distributions for $p = 0.3$, $k = 1$, $m = 1000$, and multiple values of $r$. Note that only very few documents in the index have the same false positive probability. Nonetheless, we can still calculate the true positive probability for every result since we can store the false positive rate for each document individually. The 95% and 99%

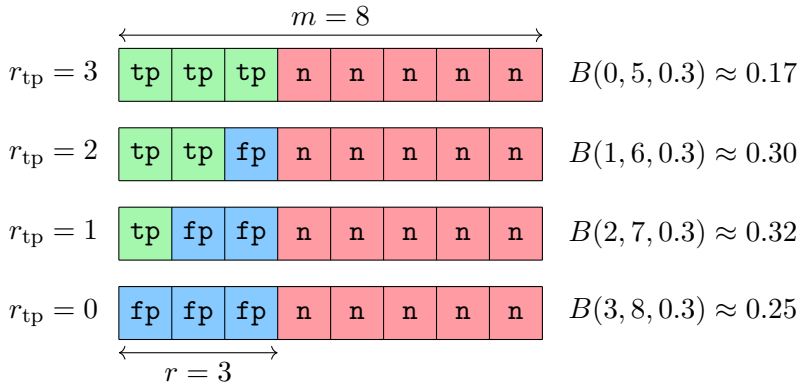


Figure 3.8: Illustration of the intuition behind the probability distribution of $r_{tp}$. For $m = 8$ and $r = 3$ there are four possible values for $r_{tp}$. The most likely value, in this case 1, is determined by the binomial distribution. In other words, the most likely value for $r_{tp}$ given $m = 8$, $r = 3$, $k = 1$ and $p = 0.3$ is 1.

| $r$ | 300 | 450 | 575 | 700 | 825 | 950 |
|---|---|---|---|---|---|---|
| $r_{tp}$ - mean | 2 | 216 | 394 | 575 | 752 | 930 |
| $r_{tp}$ - 95% CI | [1, 41] | [180, 252] | [362, 426] | [548, 602] | [732, 772] | [919, 941] |
| $r_{tp}$ - 99% CI | [1, 52] | [168, 264] | [352, 436] | [540, 610] | [725, 779] | [915, 945] |

Table 3.1: Confidence intervals for the number of true positives contained in a query result for $p = 0.3$, $k = 1$, $m = 1000$ and $r \in \{300, 450, 575, 700, 825, 950\}$.

confidence intervals are depicted in Table 3.1. As expected, the higher $r$ is, the smaller the confidence interval.

### 3.1.6 Computational Complexity

Let $Q$ be a query of length $m$. Then the time and I/O complexity of the index query process can be determined by analysing the individual steps. Each of the query terms is processed $k$ times in the `hash`-step. During the `input`-step each of the hash values is converted into $h$ I/O requests of length $|D|/h$ ($|D|/h$ will never be smaller than the disks block size). The `and`-step aggregates the data that has been read from disk. The aggregation is only needed when $k > 1$. Afterwards, each of the resulting columns is summed once during the `add` step. Finally, the `sort`-step reorders the result set. The number of I/Os can be derived by analysing the `input`-step. Let $B$ be the disk block size then the number of I/Os can be seen below.

$$mk \frac{|D|}{B}$$

Each of the $m$ terms is hashed $k$ times. For each of these hashes $|D|/B$ disk blocks are read. The computational complexity can be deduced by summing up the computational complexity of each of the steps of the query process. The `and`-step is not executed for $k = 1$.

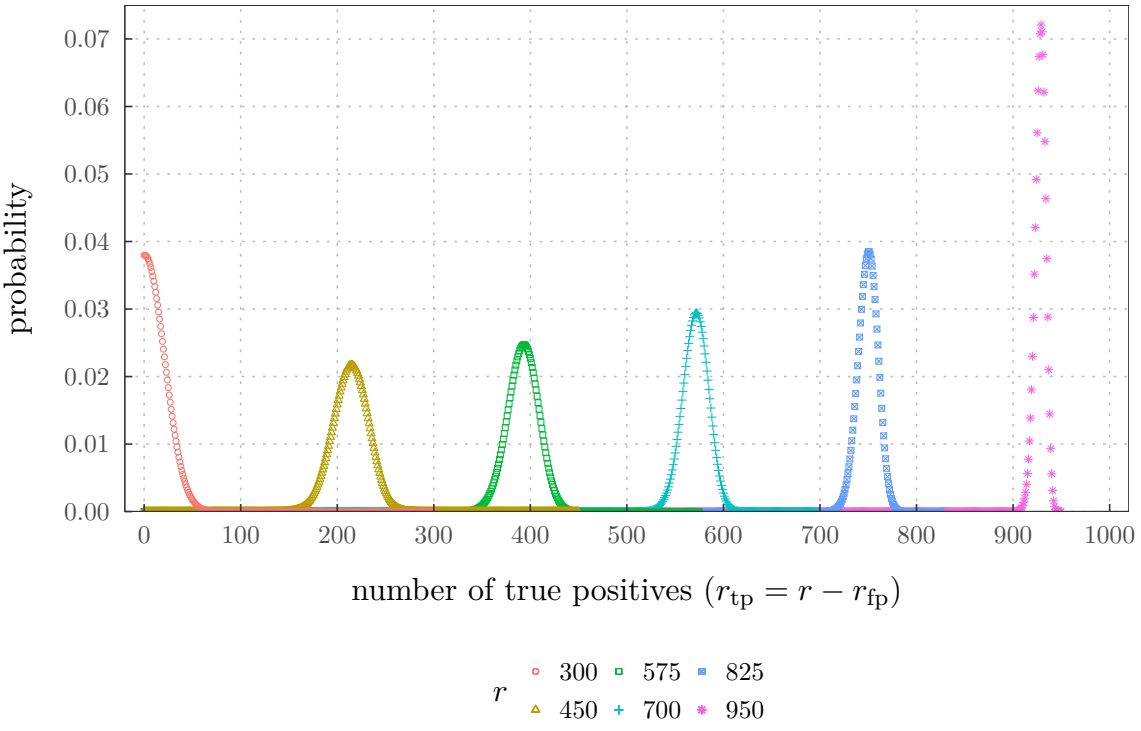


Figure 3.9: Discrete probability distribution of the number of true positives contained in a query result for fixed values of the false positive rate ($p = 0.3$), the number of hash functions ($k = 1$), the query length ($m = 1000$) and the result ($r \in \{300, 450, 575, 700, 825, 950\}$).

$$\begin{aligned}\overset{\texttt{hash}}{\mathcal{O}(mk)} + \overset{\texttt{input}}{\mathcal{O}(mk|D|)} + \overset{\texttt{and}}{\mathcal{O}(m(k-1)|D|)} + \overset{\texttt{add}}{\mathcal{O}(m|D|)} + \overset{\texttt{sort}}{\mathcal{O}(|D|\log|D|)} \\ = \mathcal{O}(mk|D| + |D|\log|D|)\end{aligned}$$

In Section 4.8 we show that for small queries the execution time is dominated by the sorting term. As expected, for larger queries the other terms dominate the query time. The `sort` step currently sorts the whole result set. To enable faster processing of small queries the reference implementation provides the option to use partial sorting. This drastically reduces the execution time of smaller queries.

## 3.2 Engineering Implementation Details

In this section, we outline some of the optimizations that result in a performance increase of two orders of magnitude over `BIGSI`. These were used to create a highly optimized `C++` implementation that is available at https://github.com/devgg/cobs. The optimizations can be selectively disabled during compile time.

### 3.2.1 Linux AIO

Chen et al. [CLZ11] showed that exploiting internal parallelism greatly increases the bandwidth of SSDs. For example, the bandwidth for 4 KiB random reads could be increased 7.2-fold by increasing the queue depth from 1 to 32. Since a query requires $hkm$ random disk accesses increasing the queue depth significantly improves I/O performance. For example, for $h = 14$, $k = 1$, and $Q = 65,536$, which resembles real-world parameters, 917,504 random reads are necessary.

To achieve optimal I/O performance we use Linux AIO (Asynchronous Input/Output). Additionally, direct I/O is used to access the raw block device and bypass kernel space buffering. To maximize queue depth all I/O requests are passed to the kernel with one call to `io_submit`.

### 3.2.2 SIMD

In the `add` step of the query process (Figure 3.3), 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 bitpacked. That is to say, each cell is represented by one bit. In the output of the `add` step, however, each document is represented by a 16-bit 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 mapping to expand 8 bits output by the `and` step to the 128 bits needed by the `add` step. This expansion can be seen in Figure 3.10.

The two-dimensional array on the left represents the bitpacked rows that are output by the `and` step. One byte, which corresponds to 8 documents, is used to access the expansion array. This array has 255 entries with a length of 128 bit each. That is to say, each of the

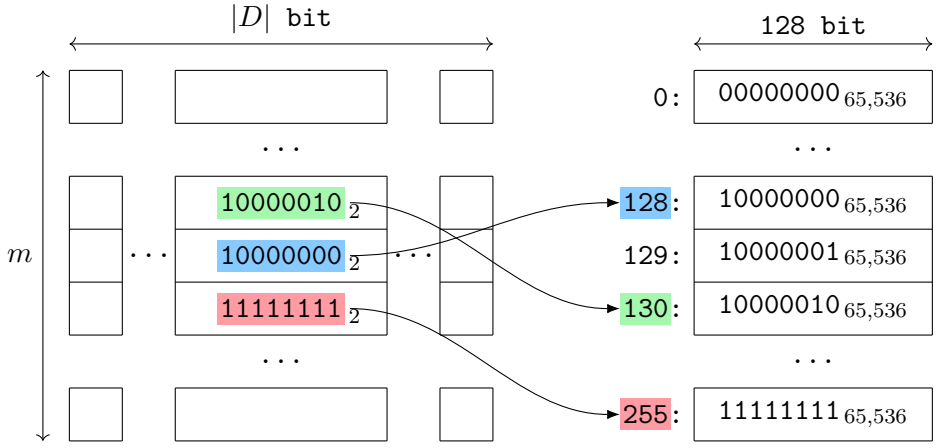


Figure 3.10: The expansion of the bitpacked rows during the `add` step.

documents is mapped to 16 bits. This is represented in the figure by an eight digit number with base 65,536. With these 16 bits, the final result can now be calculated by summing up the expanded values for each document. This is done, for 8 documents at once, with a single `SIMD` instruction to reduce the processing time of this expansion by almost 50%.

### 3.2.3 OpenMP

`OpenMP` is used in all steps of the query process to make use of the available processing cores. Most parts of the query algorithm can be trivially parallelized. `OpenMP` automatically parallelizes these steps at compile time. The main loop of the `hash` step is parallelized by splitting the $q$-grams equally among the threads. During the following `input` step only the creation of the I/O control block (`IOCB`) structures is parallelized. Again the creation of the $km$ `IOCB` structures is equally distributed among threads.

The selected rows returned by the `input` step are processed in parallel by both the `and` as well as the `add` step. For each of the $m$ $q$-grams, the `and` step combines $k$ rows. These $m$ tasks are again parallelized. Similarly, the `add` step is parallelized by not summing up all the expanded rows at once but rather creating as many sub sums as there are threads and then combining these sums. In other words, the `add` step requires a reduction to produce the final sum. The `sort` step also uses parallelization for the creation of the result pairs.

As can be seen in Section 4.9 not all of these optimizations lead to the expected performance benefit. Further research is needed to determine the reasons for this behaviour.

## 3.3 Dynamic Scaling

Most of today's collections are constantly changing. New web pages are published and genomes sequenced. As a result, an index needs to be able to dynamically insert, modify and remove documents. Furthermore, most indices need to support sharding on multiple machines to accommodate the exponential growth of modern collections. For example, the raw sequencing data in the European Nucleotide Archive doubles every 10 months [CAA+12].

To support insertion, modification, and deletion, the padding discussed in Section 3.1.4 is used. Additionally, the rest of the available disk space is utilized by allocating multiple empty subindices of different sizes. This allows new documents to be inserted into signatures of smaller size. One drawback of this approach is that each of these subindices needs to be accessed during the query process. Currently there is no approach that allows the index to grow dynamically without an increase in space overhead. Therefore the index should be rebuilt periodically. By modifying the mapping from index column to document identifier deletion can be supported. A document can be modified by combining deletion and insertion.

To accommodate for the exponential growth in genomic data and to reduce hardware cost, the index can be sharded to multiple machines. Naturally, the index is sharded by distributing its subindices. To evenly distribute both space and computational requirements, each machine should have an equal amount of subindices with a similar total size. In other words, the number of subindices determines the execution time.

# 4. Evaluation

Currently there are very few indices which operate on a similar dataset. Most existing indices were built for sequence data with high $q$-gram sharing across documents. As such, the existing benchmarks rely on such datasets. Probably the most popular benchmark for sequence-search was introduced by Solomon and Kingsford [SK16]. In this experiment 2,652 human RNA-sequence reads are indexed and then searched on. This experiment has been used for performance comparison by [SK17], [SHCM17], [PAB+17] and [BdBR+17]. While `COBS` is optimized for millions of relatively small (10 million $q$-grams) documents with little $q$-gram sharing, `SBT`, `SSBT` and `Mantis` perform best on documents with high $q$-gram similarity. This is demonstrated in [BdBR+17] by simulating the storage requirements of `SBT` with a growing number of documents. For documents with low $q$-gram sharing the size of an `SBT` with 250,000 documents would exceed one pebibyte.

`COBS` does not rely on $q$-gram similarity and therefore can be efficiently used for viral and bacterial data. `SBT`, `SSBT` and `Mantis` are meant to be used for genomes with high $q$-gram sharing. As a result, they scale super-linearly with the number of documents for datasets where $q$-gram sharing is low. Therefore a comparison between `COBS` and these tools would not be of much value. Since `BIGSI` is meant to operate under similar conditions, we will compare ourselves to it. `BIGSI` can be seen as `COBS` with one subindex (also we add padding for faster disk access). Therefore we evaluate the relation between subindex size and index construction time, false positive rate and execution time.

## 4.1 Procedure

We performed our experiments on an Amazon Web Service i3.8xlarge instance with 32 vCPUs, 244 GiB of memory and 1.9 TB non-volatile SSD-backed instance storage. Despite these beefy specifications the index can easily be constructed and queried on commodity hardware, the only requirement being an SSD drive to store the index. Our experiments are based on all the bacterial and viral genome sequences available in the European Nucleotide Archive (ENA) as of December 2016. This microbial dataset was first used in [BdBR+17]. The 447,833 sequence files were first converted from the raw read format to the intermediate

`McCortex` format [ICT+12]. This format was used to remove duplication by converting the reads to 31-grams. Additionally, the built-in cleaning feature was used to remove erroneous sequences. Lastly, the `McCortex` files were converted to a custom format containing just the unique 31-grams of each document. Since the alphabet of the 31-grams is made up of the four characters introduced in Section 2.1, 8 bytes are needed to store one 31-gram. In other words, each character can be stored in 2 bits and as a result 7 bytes and 6 bits are required to store one 31-gram.

We build multiple indices to compare the performance implications of different subindex sizes. The subindex sizes are 2, 4, 8, 12, 16, 28 and 56 kibibytes, respectively. The biggest index contains all 447,833 documents of the ENA corpus in one subindex. Therefore, it can be seen as a reimplementation of `BIGSI` with added padding. Only documents with less than 10,000,000 unique $q$-grams were included in the indices.

## 4.2 Index Construction

The indices were constructed on the Research Computing Core of the Wellcome Trust Centre for Human Genetics at the University of Oxford. Each of the cluster nodes has 16 GiB of memory and a single CPU. However, the index can be built on a single consumer grade machine with less memory. All the indices have been constructed on 447,833 documents containing all bacterial and viral genomic sequences available in the European Nucleotide Archive as of December 2016. The total size of these documents containing only the unique $q$-grams is 10.25 TiB. These files were created from the cleaned `McCortex` files in under 170 hours. Each index is constructed in 4 recursive steps, each of which combines 32 documents/indices ($\log^*_{32} 447,833 = 4$, where $\log^*$ is the iterated logarithm). Index size, build time and build I/O can be seen in Figure 4.1. Fluctuations in cluster load explain the inconsistent build times. As expected smaller subindex sizes improve all measured metrics.

## 4.3 Real versus Synthetic Query

Both synthetic and real queries were used to gauge performance. 2,147 DNA sequences from the Comprehensive Antibiotic Resistance Database (CARD) [MWN+13] were used to measure real-world performance. The mean length of these queries is 937 base pairs. For this measurement caching was enabled to better reflect the real-world performance differences. We speculate that there is no significant difference in execution time between the two query types.

To confirm this hypothesis we compared the performance of the 2,147 CARD queries with 2,147 synthetic queries of equal length. We repeated the experiment three times and the results can be seen in Table 4.1. The execution time is split into the different processing steps introduced in Section 3.1.2.

There are consistent differences in the `input` and `sort` step. CARD queries exhibit much higher similarity, therefore the input step should be faster due to caching effects. For the `sort` step we suspect that the different distributions produced by the CARD queries take

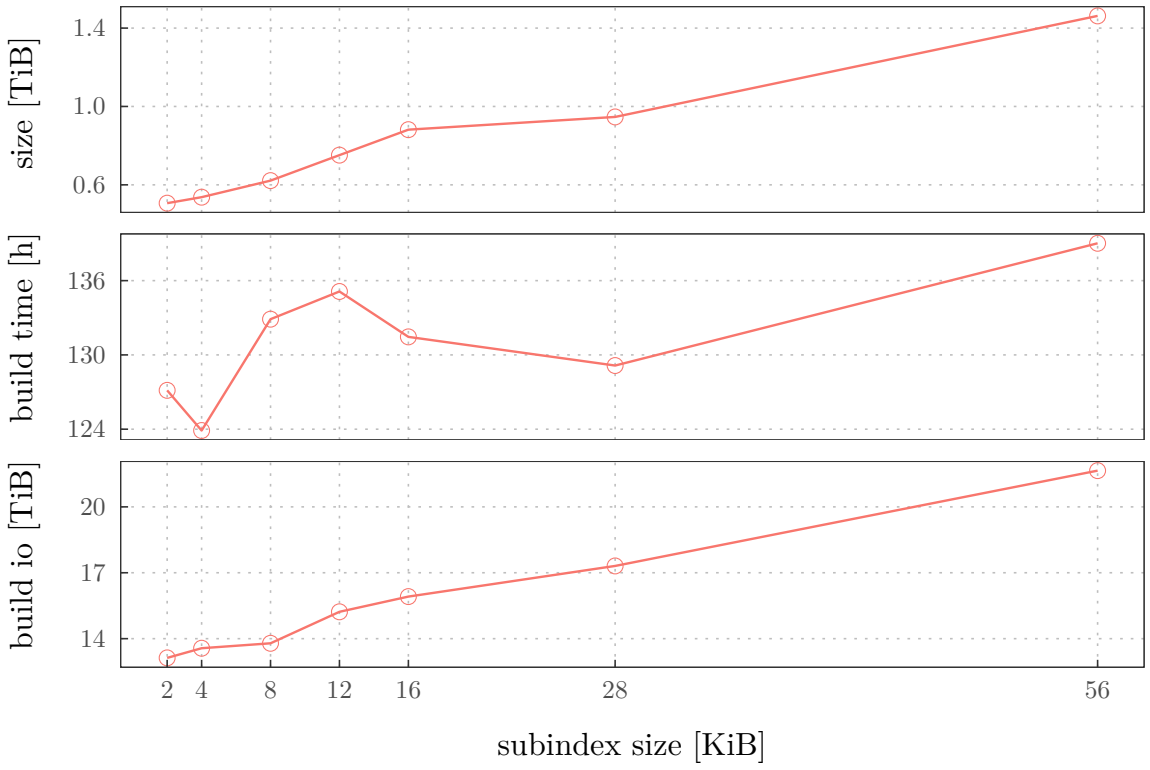


Figure 4.1: Index size, build time and build disk I/O for different subindex sizes

| | `hash` | `input` | `add` | `sort` | Σ |
|---|---|---|---|---|---|
| CARD 1 [s] | 61.37 | 132.20 | 32.26 | 164.76 | 391.99 |
| CARD 2 [s] | 61.51 | 132.31 | 32.22 | 164.94 | 392.26 |
| CARD 3 [s] | 61.41 | 132.79 | 32.20 | 163.79 | 391.47 |
| synthetic 1 [s] | 60.87 | 137.55 | 32.27 | 155.12 | 386.82 |
| synthetic 2 [s] | 60.90 | 136.56 | 32.25 | 153.59 | 384.16 |
| synthetic 3 [s] | 61.23 | 136.83 | 32.22 | 157.06 | 388.34 |
| CARD Σ [s] | 184.29 | 397.31 | 96.69 | 493.49 | 1175.72 |
| synthetic Σ [s] | 183.00 | 410.95 | 96.73 | 465.77 | 1159.31 |
| difference [s] | -1.29 | 13.64 | 0.05 | -27.72 | -16.41 |
| difference [%] | -0.70 | 3.32 | 0.05 | -5.62 | -1.40 |

Table 4.1: Performance difference between CARD and synthetic queries.

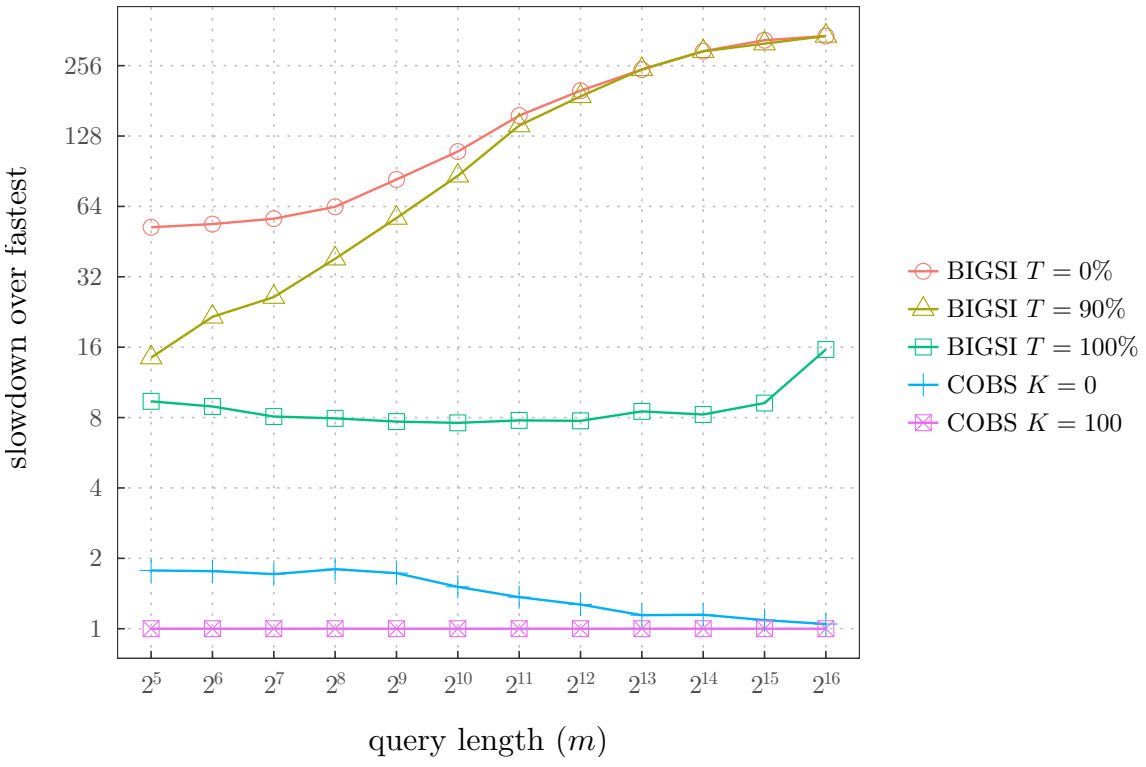


Figure 4.2: Slowdown of the `Python` implementation over our `C++` implementation.

longer to sort than the distributions produced by the synthetic queries. Since sorting is not the focus of this thesis, no further investigation has been done. Performance of the other processing steps remains unaffected. This makes sense for `hash` and `and` since the same work needs to be done for these steps regardless of the input. Since there are many different types of real-world queries that vary greatly in length and similarity, we opted to continue our measurements with purely synthetic queries that should offer more consistent results. At the same time the performance of a synthetic query should not differ much from a real-world counterpart of equal length.

## 4.4 Practical Benefits of `COBS` over `BIGSI`

`BIGSI` is implemented in `Python` with an index stored in a `Berkeley DB` key-value storage. This results in performance that cannot be compared fairly to our `C++` implementation with direct, aligned device access (Linux aio with `O_DIRECT`). To show the practical implications of using `COBS` over `BIGSI` we still decided to do a comparison. For the experiment we query both indices with synthetic queries of various lengths. Let $T$ be the threshold parameter of `BIGSI` which corresponds to $\epsilon$ in Problem 1. Furthermore, let $K$ be the parameter that defines how many of the documents should be returned (top-K) by `COBS`. Then the multiplicative slowdown of the `BIGSI Python` implementation over our `C++` implementation of `COBS` can be seen in Figure 4.2. For $K = 0$ all results are returned.

This figure should give a good idea of the practical benefits of using `COBS` over `BIGSI`. For $T = 100\%$ `BIGSI` uses an exact matching algorithm which is outperformed by our approximate solution by an order of magnitude. For $T = 0\%$ and $T = 90\%$ the performance

advantage increases to two orders of magnitudes. Since synthetic random queries are used a threshold value of $T = 90\%$ should result in a very small result set. This early pruning increases performance for smaller queries but does not impact the performance of bigger queries significantly.

In practice the performance differences would mean an execution time for a query with 65536 $q$-grams of under 5 seconds compared to over 27 minutes. To fairly compare the scientific ideas behind `COBS` and `BIGSI`, the rest of our experiments will include an `COBS` index with only one subindex which can be seen as an improved version of `BIGSI`. However, many of our improvements such as not using a key-value store and padding the data are applied. Therefore, a pure reimplementation of `BIGSI` would perform worse.

## 4.5 False Positive Distribution

One of the challenges recognized by Zobel et al. [ZMR98] is the varying false positive rate within the index. The variance stems from the differences in document lengths. In other words, documents of different sizes are inserted into Bloom filters of equal size. We introduced `COBS` which mitigates this problem by having multiple subindices. `COBS` groups documents of similar lengths which results in lower variance in false positive rate and a smaller memory footprint.

For this experiment, we queried each index 100 times with synthetic random queries that did not contain $q$-grams of the original documents. That is to say, the results only consist of false positives. A histogram was created for each of the indices. Figure 4.3 shows a relative histogram of the number of false positives for $m = 1000$, $p = 0.3$ and different subindex sizes. The y-axis shows the percentage of results that had a certain value. Each result only contains false positives since the queries were constructed to not contain any of the document's $q$-grams.

For a subindex size of 1 bit, each document has a signature with a false positive rate of exactly 0.3. As a result, the histogram resembles the probability mass function of the binomial distribution with 1000 tries, 300 successes and a success rate of 0.3. The histogram for 1 bit was calculated to illustrate what an optimal distribution would look like. The histograms of the other indices reflect the document size distribution of the ENA dataset. As expected, smaller subindex sizes resulted in false positive distributions that are closer to the optimal distribution. That is to say, the distribution for a subindex size of 1 bit. Notably, the largest index, which corresponds to `BIGSI`, has very few documents with the desired number of false positives.

## 4.6 Execution Performance

The execution time has been measured for the different indices and various query lengths. A test run consists of 1000 synthetic queries. Each test run was preceded by 200 warm-up queries. Figure 4.4 shows the total execution time for queries of different lengths. For smaller queries, the execution time is dominated by the `sort` step. Since the performance of this step only depends on the number of documents in the index, this step takes constant

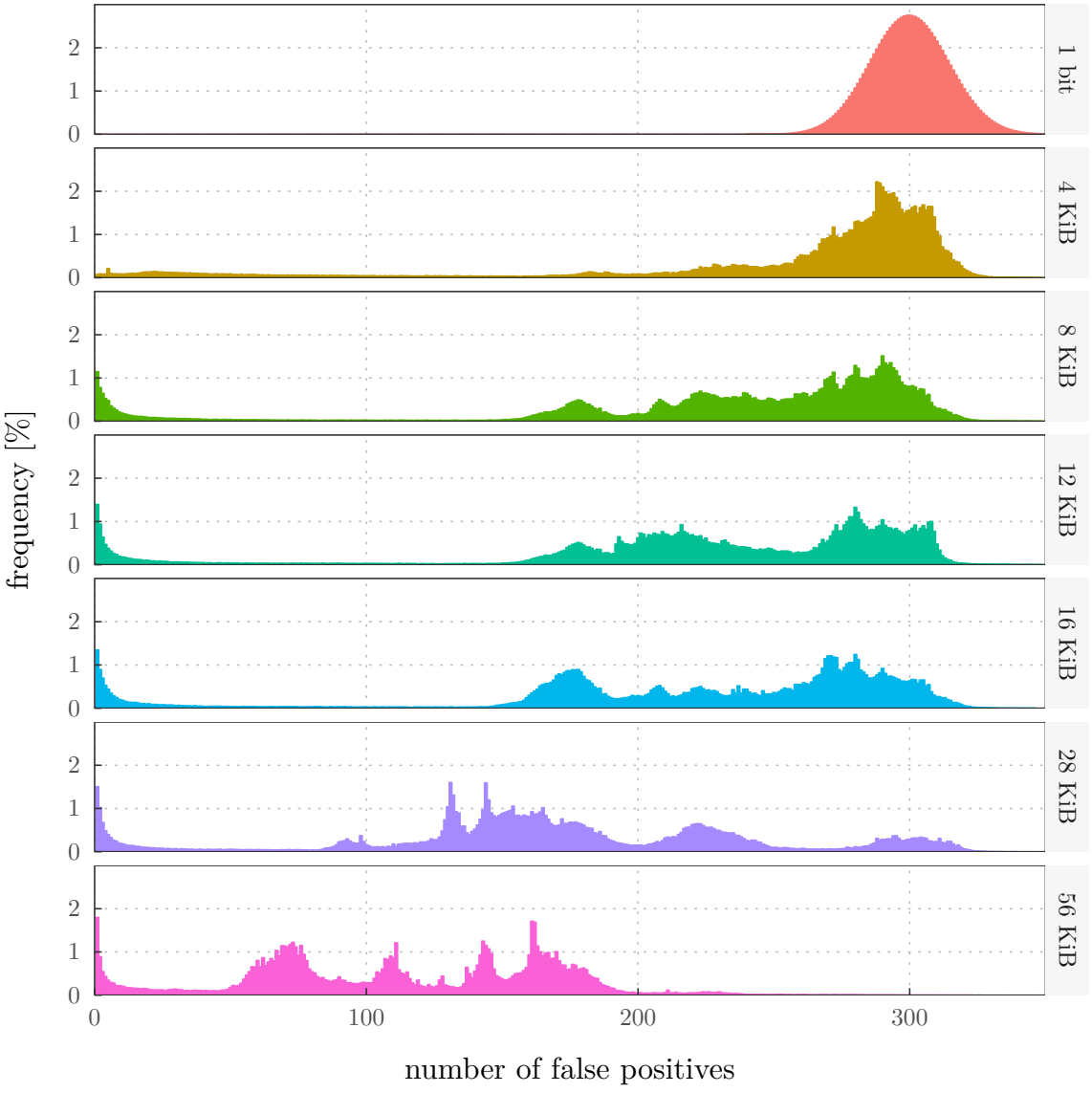


Figure 4.3: Relative histogram of the number of false positives for $m = 1000$ and a desired false positive rate of 0.3. The subindex with size of 1 bit illustrates how an optimal distribution would look like. That is to say, it shows how the distribution would look like if each document had a false positive rate of exactly 0.3.

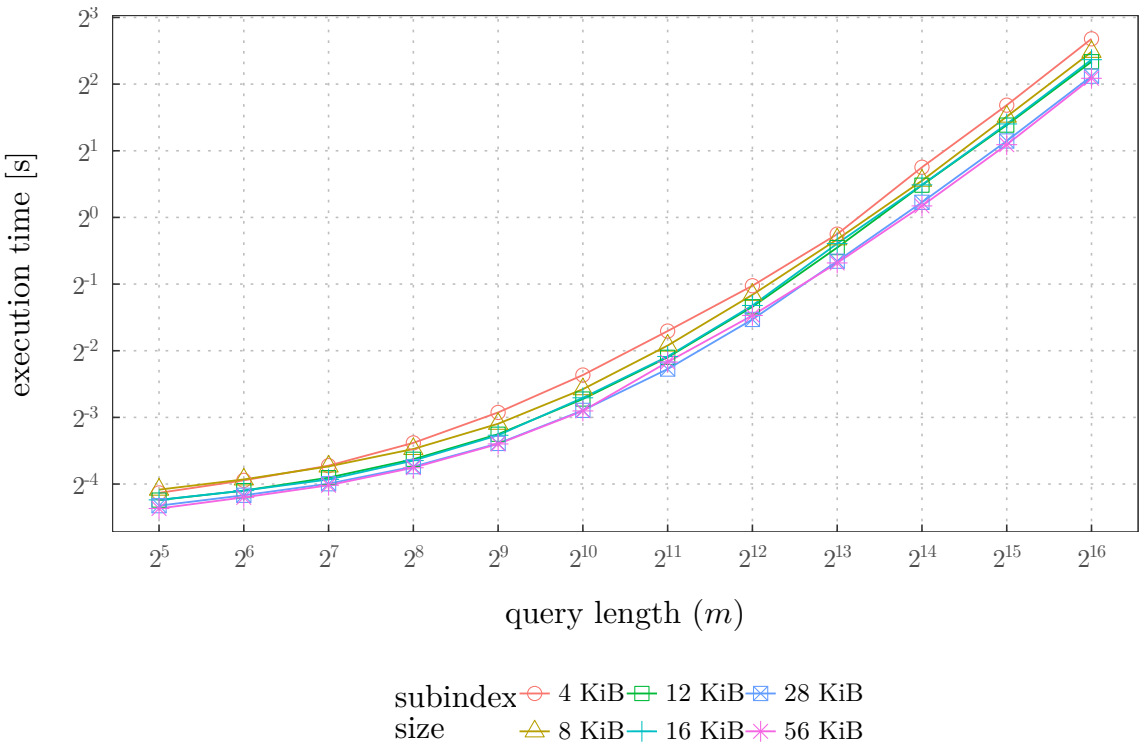


Figure 4.4: Total execution time of queries of different lengths.

time for a given index. For larger queries, the execution time scales linearly with the length of the query.

Figure 4.5 illustrates this scaling more clearly. All indices except the 4 KiB index scale as expected. That is to say, the `sort` step has a large influence on the execution time of small queries and the execution time of larger queries increases linearly with the size of the query. It is not quite clear what causes the decrease of performance for the 4 KiB index and queries of 256-1024 $q$-grams.

## 4.7 Space-Time Trade-Off

When creating a `COBS` two parameters need to be considered. By adjusting the number of hash functions the false positive rate can be controlled. For the viral and bacterial dataset a false positive rate of 0.3 is sufficient. For the reasons outlined in Section 3.1.3, using one hash function is enough to achieve this rate. The other parameter that can be adjusted is the number of subindices. By using more subindices the mean signature size decreases and the mean false positive rate increases. The values of the increases depend on the distribution of the document lengths. In Section 3.1.4 it was shown that the same amount of data needs to be read from disk regardless of the subindex size. However, smaller subindices result in more random disk accesses which, while not prohibitively slow on SSDs, negatively impacts execution time. While SSDs do not incur as much as a penalty for small random reads as HDDs, the performance impact of using more subindices is substantial. Figure 4.6 illustrates the trade-off between index size and execution time slowdown for the ENA dataset.

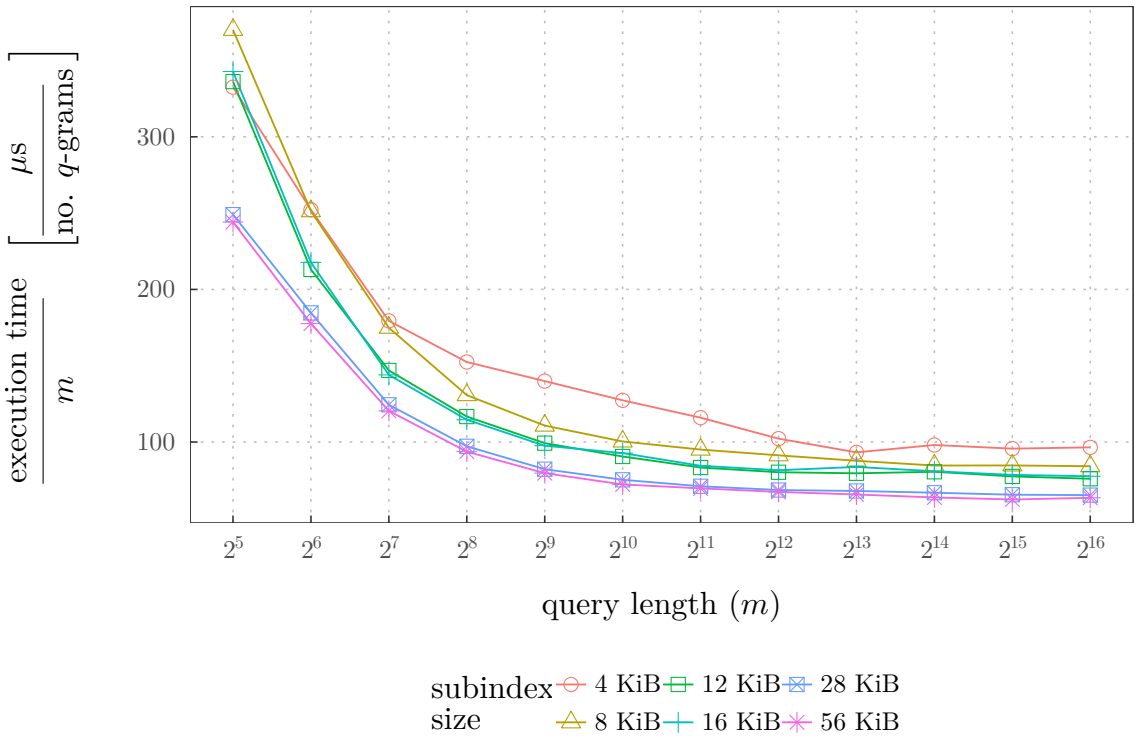


Figure 4.5: Execution time per $q$-gram of queries of different lengths.

Each of the plots shows the trade-off between index size and query performance for one query length. The performance benefit of increasing the index size diminishes for indices over 28 KiB. These measurements can be used to determine the optimal index parameters for the experimental machine and the ENA dataset. For the real world index, a subindex size of 8 KiB was chosen as a reasonable trade-off between the available disk space and the execution time slowdown of any query. Generally, multiple additional factors influence the choice of parameters.

**Maximum Document Size** The maximum document size linearly influences the signature size of the last subindex. By excluding unusually large documents, the total index size can be reduced (especially when using a small number of subindices). These large documents could be indexed in a different fashion. For example, they could be split up and then inserted into the index.

**False Positive Rate** The false positive rate can be changed by adjusting the number of hash functions and/or changing the number of subindices. Its value depends on the expected query lengths and the desired false discovery rate [BdBR+17]. For longer queries a significantly higher false positive rate can be chosen. Although the disk space reduction is diminishing as the false positive rate approaches 1.

**Document Length Distribution** The document length distribution directly influences the effectiveness of increasing the number of subindices. For some distributions it is even possible that the index size increases when using more subindices (a solution to this problem is proposed in Section 5.1). For example, when all documents have similar lengths the disk footprint can only be decreased by adjusting the number of hash

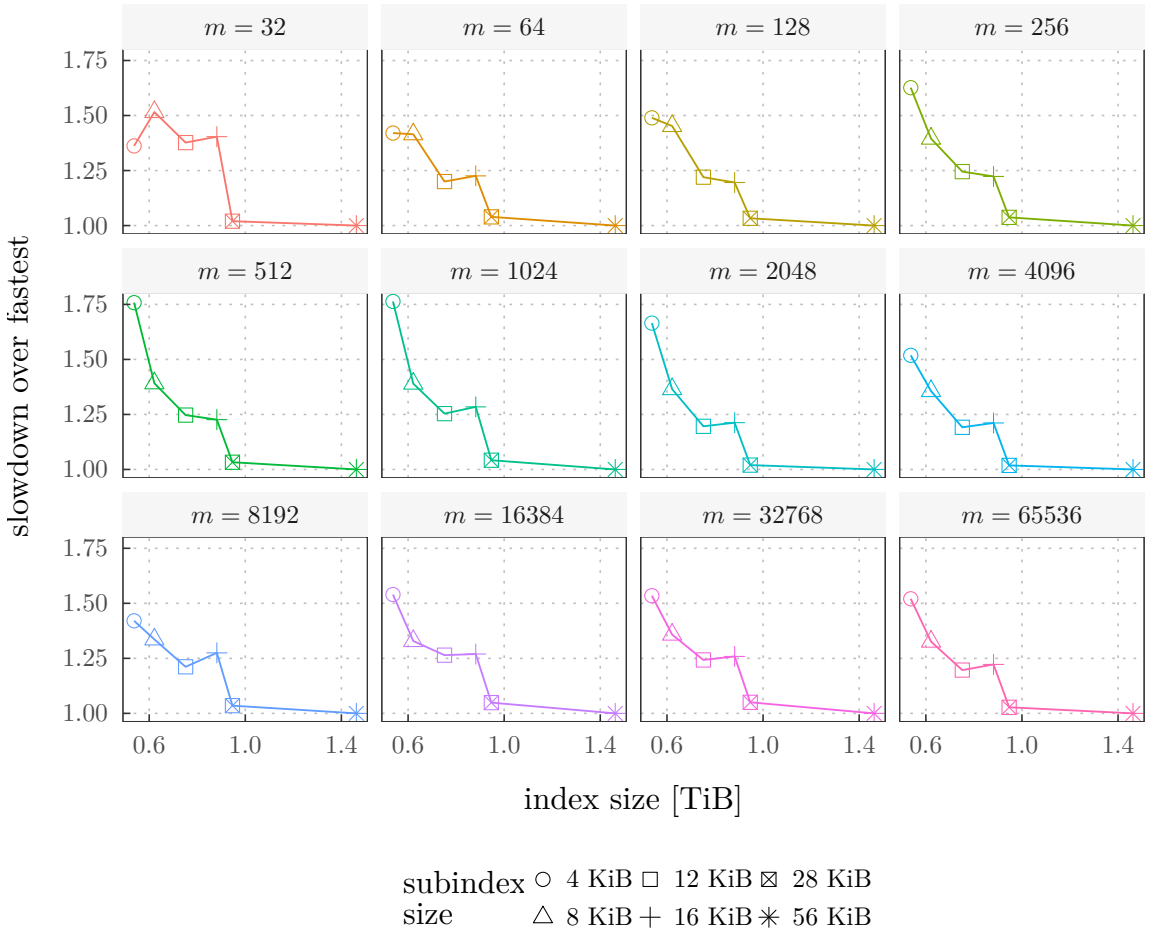


Figure 4.6: The trade-off between index space and execution time. Shown as the slowdown over the largest index and with different query lengths.

functions.

**Disk Space** The available disk space constrains the false positive rate and the execution time. The index size can be decreased by adjusting the number of hash functions and the number of subindices. Notably for certain false positive rates, decreasing the number of hash functions can decrease the required disk space.

**Execution Time** The execution time can be decreased by lowering the number of hash functions and/or the number of subindices. The effectiveness of increasing the size of the subindices diminishes.

Space and execution time are dependent on the multitude of factors described above. Nevertheless, it is reasonable to assume that a subindex size of 8 KiB will result in a good trade-off between disk space and execution time. The trade-off of the index with 4 KiB is less consistent, and, for the most relevant query lengths, inferior. Therefore, we choose a subindex size of 8 KiB to be used as the real world index.

## 4.8 Execution Time Breakdown

In this experiment we look at the breakdown of the execution time for our chosen subindex size of 8 KiB. For each of the different query lengths 400 warm-up queries were executed followed by 800 measurement queries. This experiment was repeated three times. For each of the individual steps outlined in Section 3.1.2 the mean execution time can be seen in Figure 4.7. As previously explained, the `and` step is not represented since $k = 1$.

The first chart shows the breakdown of the absolute execution time, while the second shows the percentage breakdown. The results are in line with our previous analysis. The `sort` step dominates the performance of smaller queries. This can, however, be mitigated by only doing a partial sort. The `C++` implementation of `COBS` supports partial sorting of the result set. By using partial sorting the execution time of smaller queries can be reduced significantly. For the longest queries, the execution time is equally split between CPU time and I/O time.

The `hash` step takes approximately 29µs for each $q$-gram. The rest of the measurements confirm the hypothesis that constant work is performed to hash each $q$-gram. The `input` step is one of the two main factors in deciding which subindex size to use, therefore its execution time per $q$-gram is shown in Figure 4.8. There is about a twofold increase in I/O time when comparing the smallest subindex size to the largest. The only index that does not perform as expected is the index with a subindex size of 4 KiB. For this index the I/O time per $q$-gram increases for queries of lengths between 128 and 1024. We suspect that this is due to the properties of the specific SSD used in the experiment.

In the `sort` step the findings from Section 4.3 are repeated. This step, in theory, only depends on the number of documents in the index and should be independent of the number of $q$-grams. Again, we suspect that caching effects lead to a better execution time for smaller queries since the construction of the result pairs requires less reordering. In other words, when constructing the result set fewer elements have to be reordered and thereby the array access becomes more sequential.

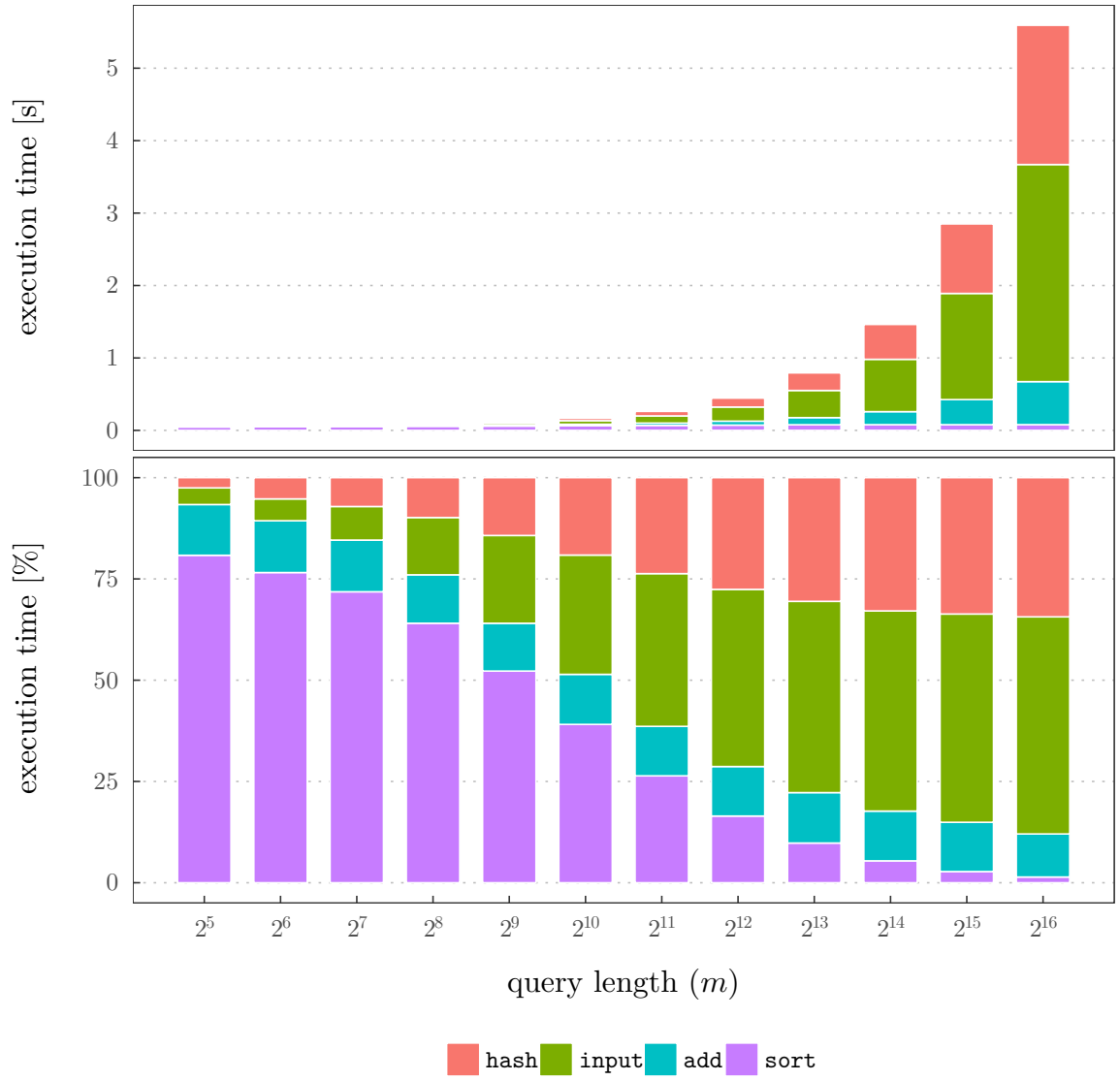


Figure 4.7: Breakdown of the execution time of queries with different lengths. The subindex size is 8 KiB.

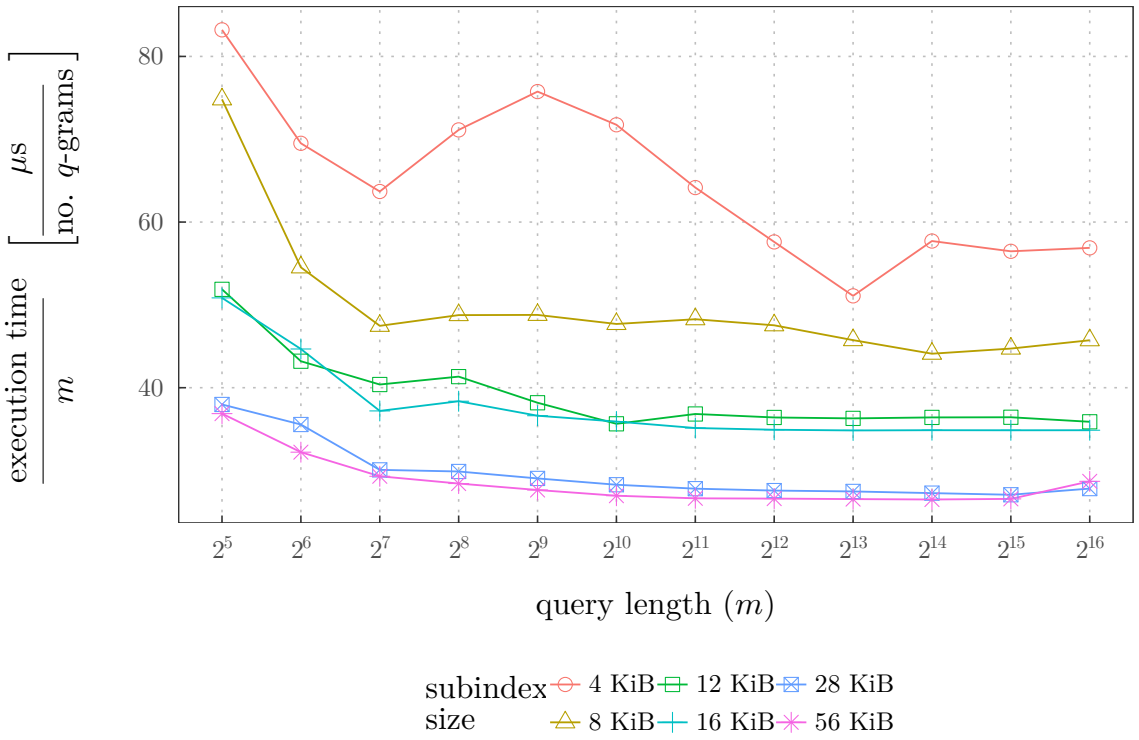


Figure 4.8: Time taken by the `input` step for different query lengths and subindex sizes.

## 4.9 Engineering Optimizations

In Section 3.2 several engineering optimizations were introduced. The impact of the `SIMD` and `OpenMP` improvements is illustrated in this section. As previously described, we used the index with a subindex size of 8 KiB. For each of the different query lengths, 400 warm-up queries were executed, followed by 800 measurement queries. Figure 4.9 shows the speedup of the optimizations for different query lengths.

The four plots represent the four relevant processing steps introduced in Section 3.1.2. `OpenMP` is used in every step of the algorithm while `SIMD` instructions are only used in the `add` step. During the `hash` step, `OpenMP` is used to parallelize the generation of hashes. The number of hashes that need to be generated is the product of the number of hash functions ($k$) and the query length ($m$). Surprisingly, using `OpenMP` does not improve performance. On the contrary, for smaller queries not using `OpenMP` results in a significant speedup. We suspect that the very fast hash algorithm (`xxHash`) is limited by memory speed. Further investigation is needed to determine what exactly causes this behaviour. During the `input` step the creation of the I/O control block (`IOCB`) structures is parallelized. This does not result in a significant performance impact. One `IOCB` structure must be created for each of the $km$ disk reads.

The performance of the `add` step can be significantly improved by using `SIMD` and `OpenMP`. By using 128-bit `SIMD` instructions to expand the bitpacked rows, the performance of the 64-bit base implementation can be increased almost exactly twofold. In the future, the execution speed could be further improved by using 256-bit or 512-bit `SIMD` instructions. Furthermore, the expansion is parallelized by `OpenMP`. Since a reduction is used to calculate

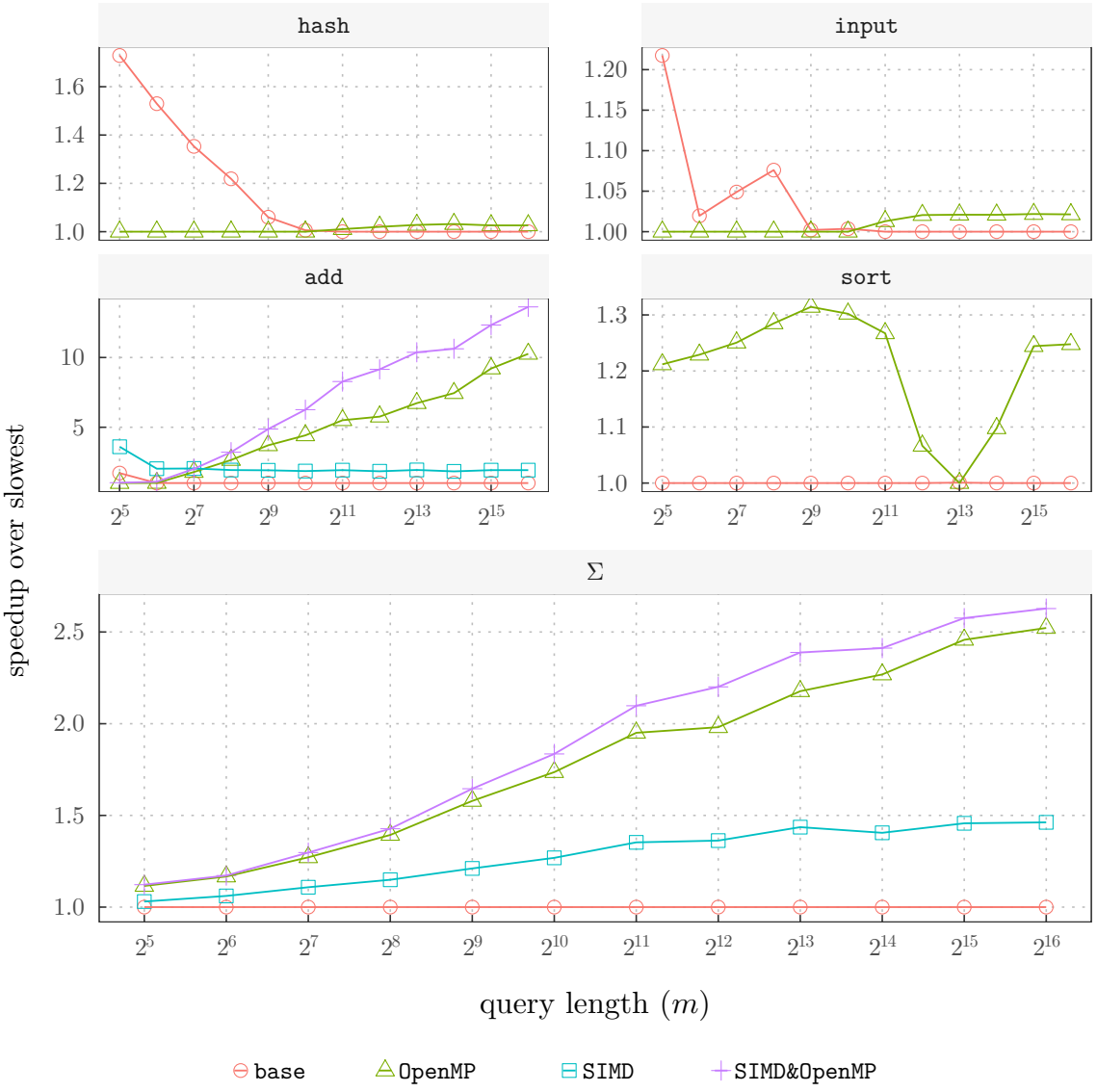


Figure 4.9: The impact of the engineering optimizations on the total execution time and on the execution times of the individual steps of the query process. `OpenMP` used all 32 available cores.

the sum in a memory-friendly way, using `OpenMP` results in better performance for longer queries. Parallelization is used during the `sort` step to construct the result pairs. Since sorting only accounts for a small portion of total execution time for longer queries the drop at $m = 2^{13}$ does not significantly change the total execution time. As a result, the drop in speedup does not warrant further investigation. The implementation of `COBS` supports partial sorting to greatly increase the total execution time of smaller queries.

# 5. Conclusion

This work builds on the ideas of `BitFunnel` and provides multiple techniques to solve the challenges of approximate pattern matching on genomic data. We first examine the specific properties of viral and bacterial genomic data and its implications on the pattern matching and indexing approach. As a result, a $q$-gram-based approximate pattern matching approach is chosen. Next, related genomic sequence-search indices are explored. Most of them are ruled out since they rely on the similarity of the genomic sequences which is lower for viral and bacterial data. The applicability of classic information retrieval indices is evaluated. Inverted index-based approaches are not considered further since the number of unique terms greatly exceeds the number of documents for the viral and bacterial dataset. Lastly, signature-based indices are shown to be a promising foundation for our index since they do not scale with the number of unique terms across documents.

The Bit-Sliced Signature Index is introduced which uses Bloom filter-based signatures. This index is similar to the Bitsliced Genomic Signature Index [BdBR$^+$17] but has practical improvements such as not using a key-value storage and padding the rows to the disk block size. To help with the choice of parameters for this index, the trade-off between using more hash functions and using more space is examined. We come to the conclusion that the parameter choice made in [BdBR$^+$17] is not optimal since the parameters were chosen with an equation that only minimizes space. Next, the Compact Bit-Sliced Signature Index is introduced, which uses multiple subindices to store the documents compactly. The second trade-off between space and execution performance is illustrated, which depends on the distribution of the document lengths.

The quality of the result is improved by introducing Problem 2. Instead of returning a result which consists of an unknown number of true and false positives, the probability distribution for the number of true positives should be returned. The provided equations solve the problem and thereby increase the confidence of the user in any returned result. Next, the computational complexity and the engineering implementation details are explored. `SIMD` is used to reduce the processing time of the `add` step by almost 50%. Lastly, ideas are introduced to outline how the index can scale dynamically and how it can support sharding

on multiple machines.

To evaluate the impact of the new techniques several experiments are conducted. It is shown how an index containing 447,833 viral and bacterial datasets can be constructed in under 140 CPU hours. The difference in performance between synthetic queries and queries based on the Comprehensive Antibiotic Resistance Database is shown to be below 1.5%. This confirms the hypothesis that the execution time is largely independent of the type of query used. Next, the practical benefits of using `COBS` over `BIGSI` are explored. For the ENA dataset `COBS` is up to two orders of magnitude faster and 59% smaller.

As an indicator for the amount of space wasted by the index, the false positive distribution is examined. As expected, indices with more subindices waste less space and therefore have more consistent false positives rates. The computational complexity is confirmed experimentally by looking at the execution time of queries of varying lengths. The execution time of small queries is dominated by the `sort` step and longer queries scale linearly with the number of $q$-grams. The space-time trade-off of different subindex sizes is examined and a subindex size of 8 KiB is chosen for the ENA dataset. To show where future research and engineering efforts are most useful, the total execution time is broken down into the different processing steps. This confirms the dominance of the `sort` step for smaller queries which can be reduced by using the partial sorting capabilities of the provided `C++` implementation. Lastly, the engineering improvements are evaluated for each of the processing steps. Surprisingly, `OpenMP` does not increase the performance of the `hash` step. Nevertheless, the engineering optimizations decrease the execution time of a query of average length by a factor of two.

## 5.1 Future Work

With `COBS` we have provided an index which substantially improves on the current state of the art genomic index `BIGSI`. Furthermore, this work provides insight into space-time trade-off, engineering optimizations, true positive distributions and the computational complexity of the index. In the process, new areas of interest in research and implementation have been uncovered.

While the current index can already scale to millions of documents, its `C++` implementation does not yet support dynamic scaling or sharding across machines. We outline the basic ideas in Section 3.3 but further research is necessary to evaluate these ideas or to suggest better alternatives.

Currently, only the core of the application is implemented. To make the index accessible to researchers everywhere an online version needs to be deployed. This underlines the need for the creation of both a web server and a suitable front end. Additionally, an integration with the big genomic archives would enable the creation of a dynamic online index.

Further research is needed to explore the possibility of improving the execution time by ending execution early for documents which do not meet the required $\epsilon$-threshold (Problem 1). By pruning early, the execution time of the `add` step could be reduced significantly for high values of $\epsilon$. Furthermore, the performance of the `hash` step might be

improved by using a different hashing implementation that benefits from parallelization. Lastly, a parallel sorting algorithm could further improve the performance of small queries.

Another promising research direction is the trade-off between subindex size/index size and execution performance. This largely depends on the distribution of lengths of the documents. For collections with documents of similar lengths, using more subindices, will not result in a big reduction in space but will reduce the execution time significantly. On the contrary, for collections containing documents of varying lengths, large amounts of space can be saved for only a small loss in performance.

Therefore, one could use subindices of varying sizes that fit the document size distribution more closely. For example, for a corpus of 98,304 documents where two-thirds of the documents are small and one-third of the documents are large, two subindices of 4 KiB and 8 KiB could be used to minimize both index size and execution performance ($4096 + 8192 = 98,304/8$). In the current implementation three subindices with 4 KiB each would be used which results in three random accesses per $q$-gram and hash function. By using two subindices of 4 KiB and 8 KiB the space requirements should stay nearly identical but only two random accesses per $q$-gram and hash function would be necessary. As we have shown in Chapter 4 this significantly reduces the execution time of the `input` step which accounts for roughly half of the execution time of long queries.

# Acronyms

**ASBT** AllSome Sequence Bloom Tree.

**BIGSI** Bitsliced Genomic Signature Index.

**BLAST** Basic Local Alignment Search Tool.

**BLAT** BLAST-like Alignment Tool.

**BSSF** Bit-Sliced Signature Files.

**CARD** Comprehensive Antibiotic Resistance Database.

**COBS** Compact Bit-Sliced Signature Index.

**ENA** European Nucleotide Archive.

**IOCB** I/O control block.

**IR** Information Retrieval.

**OpenMP** Open Multi-Processing.

**SBT** Sequence Bloom Tree.

**SIMD** Single Instruction Multiple Data.

**SSBT** Split Sequence Bloom Tree.

# List of Figures

# Bibliography


[AGM+90] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[BCC16] S. Büttcher, C. L. Clarke, and G. V. Cormack, *Information retrieval: Implementing and evaluating search engines*. Mit Press, 2016.

[BdBR+17] P. Bradley, H. den Bakker, E. Rocha, G. McVean, and Z. Iqbal, "Real-time search of all bacterial and viral genomic data," *bioRxiv*, p. 234955, 2017.

[Blo70] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[BM77] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[BM04] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[BP12] S. Brin and L. Page, "Reprint of: The anatomy of a large-scale hypertextual web search engine," *Computer networks*, vol. 56, no. 18, pp. 3825–3833, 2012.

[BW94] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," 1994.

[BXH15] B. Buchfink, C. Xie, and D. H. Huson, "Fast and sensitive protein alignment using diamond," *Nature methods*, vol. 12, no. 1, p. 59, 2015.

[CAA+12] G. Cochrane, B. Alako, C. Amid, L. Bower, A. Cerdeño-Tárraga, I. Cleland, R. Gibson, N. Goodgame, M. Jang, S. Kay *et al.*, "Facing growth in the european nucleotide archive," *Nucleic acids research*, vol. 41, no. D1, pp. D30–D35, 2012.

[CKZ09] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '09. New York, NY, USA: ACM, 2009, pp. 181–192. [Online]. Available: http://doi.acm.org/10.1145/1555349.1555371

[CLZ11] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, Feb 2011, pp. 266–277.

[FBY92] W. B. Frakes and R. Baeza-Yates, "Information retrieval: data structures and algorithms," 1992.

[FC84] C. Faloutsos and S. Christodoulakis, "Signature files: An access method for documents and its analytical performance evaluation," *ACM Transactions on Information Systems (TOIS)*, vol. 2, no. 4, pp. 267–288, 1984.

[FC88] C. Faloutsos and R. Chan, "Fast text access methods for optical and large magnetic disks: Designs and performance comparison." 1988.

[FM05] P. Ferragina and G. Manzini, "Indexing compressed text," *Journal of the ACM (JACM)*, vol. 52, no. 4, pp. 552–581, 2005.

[FO98] C. Faloutsos and D. W. Oard, "A survey of information retrieval and filtering methods," Tech. Rep., 1998.

[GHL+17] B. Goodwin, M. Hopcroft, D. Luu, A. Clemmer, M. Curmei, S. Elnikety, and Y. He, "Bitfunnel: Revisiting signatures for search," in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2017, pp. 605–614.

[HBA+16] L. A. Hug, B. J. Baker, K. Anantharaman, C. T. Brown, A. J. Probst, C. J. Castelle, C. N. Butterfield, A. W. Hernsdorf, Y. Amano, K. Ise *et al.*, "A new view of the tree of life," *Nature microbiology*, vol. 1, p. 16048, 2016.

[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, no. 2, pp. 226–232, 2012.

[ITM12] Z. Iqbal, I. Turner, and G. McVean, "High-throughput microbial population genomics using the cortex variation assembler," *Bioinformatics*, vol. 29, no. 2, pp. 275–276, 2012.

[Ken02] W. J. Kent, "Blat - the blast-like alignment tool," *Genome research*, vol. 12, no. 4, pp. 656–664, 2002.

[KMP77] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt, "Fast pattern matching in strings," *SIAM journal on computing*, vol. 6, no. 2, pp. 323–350, 1977.

[Kru16] J. Krugel, "Approximate pattern matching with index structures." Ph.D. dissertation, Technical University Munich, 2016.

[KSDR90] A. Kent, R. Sacks-Davis, and K. Ramamohanarao, "A signature file scheme based on multiple organizations for indexing very large text databases," *Journal of the american society for information science*, vol. 41, no. 7, p. 508, 1990.

[LG09] P. Lapierre and J. P. Gogarten, "Estimating the size of the bacterial pan-genome," *Trends in genetics*, vol. 25, no. 3, pp. 107–110, 2009.

[MWN+13] A. G. McArthur, N. Waglechner, F. Nizam, A. Yan, M. A. Azad, A. J. Baylay, K. Bhullar, M. J. Canova, G. De Pascale, L. Ejim *et al.*, "The comprehensive antibiotic resistance database," *Antimicrobial agents and chemotherapy*, vol. 57, no. 7, pp. 3348–3357, 2013.

[OTM+16] B. D. Ondov, T. J. Treangen, P. Melsted, A. B. Mallonee, N. H. Bergman, S. Koren, and A. M. Phillippy, "Mash: fast genome and metagenome distance estimation using minhash," *Genome biology*, vol. 17, no. 1, p. 132, 2016.

[PAB+17] P. Pandey, F. Almodaresi, M. A. Bender, M. Ferdman, R. Johnson, and R. Patro, "Mantis: A fast, small, and exact large-scale sequence search index," *bioRxiv*, p. 217372, 2017.

[PBJP17] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, "A general-purpose counting filter: Making every bit count," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 775–787.

[RRC+13] M. G. Ross, C. Russ, M. Costello, A. Hollinger, N. J. Lennon, R. Hegarty, C. Nusbaum, and D. B. Jaffe, "Characterizing and measuring bias in sequence data," *Genome biology*, vol. 14, no. 5, p. R51, 2013.

[SDKR87] R. Sacks-Davis, A. Kent, and K. Ramamohanarao, "Multikey access methods based on superimposed coding techniques," *ACM Transactions on Database Systems (TODS)*, vol. 12, no. 4, pp. 655–696, 1987.

[SHCM17] C. Sun, R. S. Harris, R. Chikhi, and P. Medvedev, "Allsome sequence bloom trees," in *International Conference on Research in Computational Molecular Biology*. Springer, 2017, pp. 272–286.

[SK16] B. Solomon and C. Kingsford, "Fast search of thousands of short-read sequencing experiments," *Nature biotechnology*, vol. 34, no. 3, pp. 300–302, 2016.

[SK17] ——, "Improved search of large transcriptomic sequencing databases using split sequence bloom trees," in *International Conference on Research in Computational Molecular Biology*. Springer, 2017, pp. 257–271.

[SLF+15] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big data: astronomical or genomical?" *PLoS biology*, vol. 13, no. 7, p. e1002195, 2015.

[TAA+17] A. L. Toribio, B. Alako, C. Amid, A. Cerdeño-Tarrága, L. Clarke, I. Cleland, S. Fairley, R. Gibson, N. Goodgame, P. ten Hoopen *et al.*, "European nucleotide archive in 2016," *Nucleic acids research*, vol. 45, no. D1, pp. D32–D36, 2017.

[TRL12] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 1, pp. 131–155, 2012.

[Ukk92] E. Ukkonen, "Approximate string-matching with q-grams and maximal matches," *Theoretical computer science*, vol. 92, no. 1, pp. 191–211, 1992.

[WMB99] I. H. Witten, A. Moffat, and T. C. Bell, *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999.

[Won85] H. K. Wong, "Bit transposed files," 1985.

[ZM06] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM computing surveys (CSUR)*, vol. 38, no. 2, p. 6, 2006.

[ZMR98] J. Zobel, A. Moffat, and K. Ramamohanarao, "Inverted files versus signature files for text indexing," *ACM Transactions on Database Systems (TODS)*, vol. 23, no. 4, pp. 453–490, 1998.