kmtricks: Efficient construction of Bloom filters for large sequencing data collections

Téo Lemane, Paul Medvedev, Rayan Chikhi, Pierre Peterlongo

kmtricks: Efficient construction of Bloom filters for large sequencing data collections

Téo Lemane [1], Paul Medvedev [2,3,4], Rayan Chikhi [5] and Pierre Peterlongo [1,*]

[1]Univ. Rennes, Inria, CNRS, IRISA, Rennes, France
[2]Department of Computer Science and Engineering, The Pennsylvania State University, USA
[3]Department of Biology, The Pennsylvania State University, USA
[4]Huck Institutes of the Life Sciences, The Pennsylvania State University, USA
[5]Department of Computational Biology, Institut Pasteur, Paris, France

*To whom correspondence should be addressed.

Abstract

When indexing large collection of sequencing data, a common operation that has now been implemented in several tools (Sequence Bloom Trees and variants, BIGSI, ..) is to construct a collection of Bloom filters, one per sample. Each Bloom filter is used to represent a set of k-mers which approximates the desired set of all the non-erroneous k-mers present in the sample. However, this approximation is imperfect, especially in the case of metagenomics data. Erroneous but abundant k-mers are wrongly included, and non-erroneous but low-abundant ones are wrongly discarded. We propose `kmtricks`, a novel approach for generating Bloom filters from terabase-sized collections of sequencing data. Our main contributions are 1/ an efficient method for jointly counting k-mers across multiple samples, including a streamlined Bloom filter construction by directly counting hashes instead of k-mers; 2/ a novel technique that takes advantage of joint counting to preserve low-abundant k-mers present in several samples, improving the recovery of non-erroneous k-mers. In addition, our experimental results highlight that the usual yet crude filtering of low-abundant k-mers is inappropriate for complex data such as metagenomes.
Availability: `https://github.com/tlemane/kmtricks`
Contact: `pierre.peterlongo@inria.fr`

1 Introduction

Consortia such as the 100,000 Genomes Project (Turnbull *et al.*, 2018), GEUVADIS (Lappalainen *et al.*, 2013), MetaSub (Mason *et al.*, 2016) and Tara Ocean (Karsenti *et al.*, 2011) have generated large collections of genomic, transcriptomic, and metagenomic sequencing data, respectively. Rather than deep coverage of a single sample, such datasets contain a collection of sequencing experiments across many samples. For example, the Tara Ocean project generated metagenomic sequencing data across ecological niches all over the oceans, totalling at least 171 thousand billions of nucleotides. Such valuables resources are unfortunately difficult to comprehensively analyze, since their size makes bioinformatics analyses difficult.

Traditional sequence analyses such as alignment to a reference database or *de novo* assembly are both difficult and limited in the results they yield. For instance, metagenome assembly of individual samples (e.g. using MetaSPAdes (Nurk *et al.*, 2017)) is often not able to reconstruct low abundance genomes and tends to collapse variants between close strains. Co-assembly of multiple samples pools together coverage from multiple sites to alleviate this but results in further loss in strain specificity. Alternatively, aligning raw sequencing data to genome databases is hindered by the incompleteness of those databases.

A recently proposed alternative is to build an index of the raw sequencing data and then later query sequences of interest, e.g. genes or shorter sequence fragments around variants such as SNPs or indels. Traditional indexing approaches, such as those used by BLAST (Altschul *et al.*, 1990) or DIAMOND (Buchfink *et al.*, 2015), do not scale to those large collections (Marchet *et al.*, 2021). Instead, customized indexing methods have been under development. A recent review surveyed 20 tools that were all published in the last couple years, aiming to index large collections of sequencing data (Marchet *et al.*, 2021), for example `BIGSI` (Bradley *et al.*, 2019), `HowDe-SBT` (Harris and Medvedev, 2019), and `Mantis` (Pandey *et al.*, 2018). These indexes are typically able to answer whether an arbitrary fixed-length sequence (k-mer) belongs to any of the samples, and, if so, which ones. Though much progress has been made, indexing a collection such as Tara Ocean has remained out of practical reach.

The vast majority of these large-scale k-mer indexing tools are based on common building blocks, three of them being: 1) k-mer counting, which summarizes sequencing data into a set of k-mers along with their abundances, 2), k-mer matrix construction, which aggregates lists of k-mer counts over a collection of samples (e.g. as in Marchet *et al.* (2020); Muggli *et al.* (2019)) in the form of a k-mer/sample matrix with abundances as values, and 3) Bloom filters construction, where the k-mer presence/absence information for each sample is converted into a Bloom

filter to save space and allow fast queries. Note that these building blocks are not specific to k-mer indexing tools, e.g. 1) and 3) are commonly used in short-read *de novo* assembly, and 2) also appears in transcriptomics analysis (Audoux *et al.*, 2017).

Importantly, these three steps are often categorized as "pre-processing" in k-mer indexing papers (e.g. (Pandey *et al.*, 2018; Harris and Medvedev, 2019)) and discounted from the running time of these indexing tools. Yet, for a dataset like Tara Oceans, these steps dwarf the running time of the subsequent index construction by up to several orders of magnitude. Although construction only needs to be done once per collection, its prohibitive running time for large collections represents an important roadblock to the usability of the tools.

In addition to the inefficiency of construction methods, sequencing errors are also dealt with sub-optimally. Even though contemporary sequencing error rates are low (0.1-1%), a vast amount of k-mers present in raw data contain sequencing errors and should be discarded during indexing. There are many read error-correction tools (Song and Florea, 2015), however, they are not a viable option for metagenomics (and RNA-seq) due to the presence of low-abundance genomes and the limited availability of reference genomes. Current approaches therefore filter out k-mers solely by checking if their abundance is below a pre-set threshold. This has the unsatisfactory drawbacks of being either too conservative (discarding all low-abundant genome data if the threshold is set too high), or too permissive (too many erroneous k-mers are kept if the threshold is set too low). In this paper, we will propose an improved method for filtering out erroneous k-mers.

Here we propose an improved algorithm for this construction step that improves both its efficiency and the ability to correct errors. Current tools take a modular approach. They first use an off-the-shelf k-mer counting tool separately for each sample, and then construct a Bloom filter from the k-mers in that sample. We observe in this paper that this modular approach has several drawbacks. First, it prevents fine-grain optimizations that can be obtained by integrating these steps. Second, it prevents efficient data lookups such as being able to identify all the samples to which a given k-mer belongs. As we will show, such lookups can help improve error-correction for these samples. In summary, by limiting themselves to a modular approach, current tools leave both significant speed-up and joint filtering opportunities on the table. Given the maturity and abundance of Bloom filter-based indexing tools, as well as a plateau in performance improvement of k-mer counting tools (Kokot *et al.*, 2017), we believe that designing better construction algorithms through integration is an important research task.

Our method for constructing k-mer matrices and Bloom filters is based on a partitioned k-mer counting procedure carefully optimized for joint multi-sample counting. The main novelty is the combination of three relatively straightforward contributions that together address the issues of long running times and sub-optimal k-mers filtering. 1) We introduce a procedure for rescuing low-abundance k-mers at the heart of the joint multi-sample k-mer counting procedure. This enables more sensitive results yet discarding truly erroneous k-mers, saving the prohibitive indexing of all (vastly erroneous) k-mers. We show that joint k-mer counting is more powerful than performing one-sample k-mer counting over a metagenomics collection. 2) We introduce the concept of *hash counting* that enables direct construction of Bloom filters indices without resorting to k-mers, saving significant time and space. 3) We incorporate for the first time existing efficient matrix transposition techniques in such a workflow, to efficiently output Bloom filter rows directly from partitioned intermediate data, saving intermediate disk space during joint multi-sample counting.

Using our workflow, we perform for the first time a massive-scale joint k-mer counting and Bloom filter construction of a 6.5 terabase metagenomics collection, in under 50 hours and 50 GB of memory, which is 3.5 times faster than the next best alternative.

2 Related works

KMC (Kokot et al.*, 2017) and DSK (Rizk* et al.*, 2013)* are two disk-based k-mer counting tools. Counting k-mers is an operation that identifies the set of k-mers present within one (or multiple) datasets and records the abundance of each k-mer. Intuitively, k-mers having a low abundance (i.e. seen few times) are more likely to be the result of one or multiple sequencing errors within reads, while k-mers above a certain abundance threshold are more likely to be correct.

In its original publication, `DSK` directly stored k-mers in a hash table by carefully controlling for memory and disk usage using partitioning. Recent versions of `KMC` and `DSK` are based on variants of an algorithm introduced by MSPKmerCounter (Li *et al.*, 2015) subsequently made popular by `KMC`. In a nutshell, sequencing reads are split into partitions stored on disk, then each partition is loaded in memory and k-mers are counted within them. Partitions are constructed in such a way that all occurrences of a k-mer appear within a single partition, and also, overlapping k-mers are attempted to be stored as longer sequences to avoid redundancy. These properties are achieved using the concepts of *minimizers* and *super-k-mers* that we will review in the Methods section, and the concept of $(k + x)$-mers that we will not use here.

`kmc_tools` is a binary tool included in `KMC` for manipulation of its results files, which can perform e.g. set operations on list of k-mers, such as intersection, union, or more complex ones. However, to the best of our knowledge `kmc_tools` does not support collections of k-mer lists (i.e. k-mer matrices) and is therefore not applicable to the work presented here.

Jellyfish (Marçais and Kingsford, 2011) is an in-memory k-mer counting tool that relies on an optimized hash table. One of its key advantages, besides efficiency, is that as a byproduct it constructs a k-mer *dictionary*: i.e. a data structure that can efficiently associate values to k-mers and supports efficient queries. Jellyfish was notably used to perform k-mer counting in `HowDe-SBT`. However, its key-value store feature was not used. Several other k-mer counting tools exist, however a recent benchmark (Manekar and Sathe, 2018) determined that `KMC 3` is one of the most efficient ones on a single node.

Simka (Benoit *et al.*, 2016) is a multi-sample k-mer counting and k-mer matrices construction tool that was used for metagenomics. It is based on a variation of the original `DSK` algorithm, modified to run on a distributed cluster.

HowDe-SBT (Harris and Medvedev, 2019) is a k-mer indexing method for large sequencing data collections, that extends the original concept of Sequence Bloom Trees (SBT) (Solomon and Kingsford, 2016). In a nutshell, `HowDe-SBT` (and in general, any SBT-based method) indexes each sample using a Bloom filter and organizes filters inside a binary tree for performing queries efficiently. `HowDe-SBT` is the most efficient variant of SBTs to date, showing fast construction and query times, yet requires an expensive pre-processing step. Precisely, the pre-processing step consists in generating from a set of samples, one Bloom filter per sample. Each Bloom filter indexes the k-mers considered as not erroneous contained in its corresponding set. This requires to perform a time-consuming k-mer counting process for each data set.

BIGSI (Bradley et al.*, 2019) and COBS (Bingmann* et al.*, 2019)* are k-mer indexing methods which also use Bloom filters, however organized in a different layout than in the SBT family. These tools represent all Bloom filters of indexed samples in a flat manner, in a way that limits cache

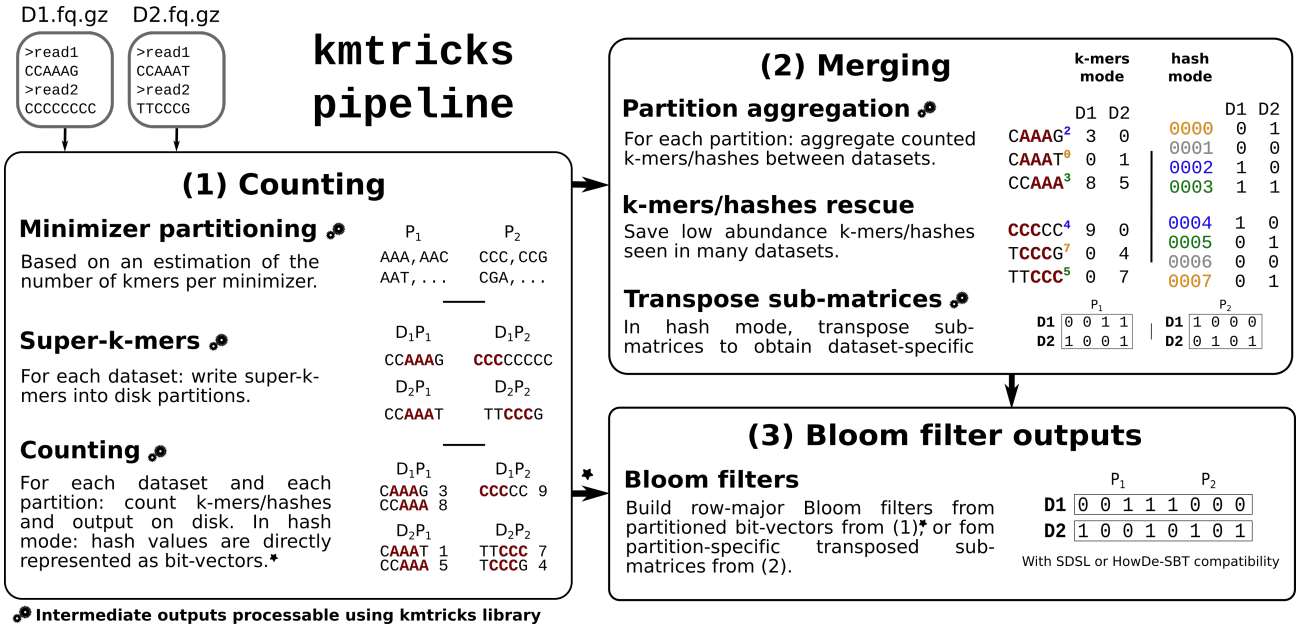

Fig. 1. `kmtricks` pipeline overview on two samples, D_1 and D_2, using two partitions, P_1 and P_2, with $k = 5$ and minimizer size of 3. Bold read sequences are minimizers (AAA and CCC). Superscript integers represent hash values.
(1) Counting: Minimizer repartition is determined by sub-sampling D_1 and D_2 and super-k-mers are then dumped on disk according to this partitioning. Each partition is then counted with the possibility to see each element as a k-mer or as its hash value. In both modes, counted partitions may be optionally dumped to disk. In hash mode specifically, for each partition a bit-vector can be output directly ($\star$ symbol).
(2) Merging: Counted k-mers or hashes from equivalent partitions are aggregated. Different kind of matrices can be obtained: a count matrix (in ASCII or binary format), a presence/absence matrix (each row represents a k-mer or a hash value associated with a presence/absence bit-vector), and, only in hash mode, a vector of Bloom filters (i.e. a matrix of presence/absence bit-vectors, where row indices represent hashes). All these matrices can be filtered using a k-mer rescue procedure described in 4.3. In hash mode, in order to obtain samples in rows and then build Bloom filters, each partition-specific sub-matrix can be transposed.
(3) Bloom filter outputs: a Bloom filter is built for each sample through concatenation of transposed sub-matrices (in those, each row corresponds to a sample). Bloom filters can be also obtained from first counting step if aggregation is not required. In this case, this corresponds to a concatenation of bit-vectors from (1) in hash mode without k-mer rescue.

misses during query. This flat structure however does not enable to reduce redundancy between samples.

3 Results

3.1 `kmtricks`: A modular pipeline and library for construction k-mer matrices and Bloom filters on large datasets

In this section we give an overview of our software `kmtricks` (for "*kmer matrix tricks*"). A more in-depth presentation and algorithmic details are provided Section 4. Essentially, `kmtricks` is a set of software components that together perform joint multi-sample k-mer counting and color matrix construction. This allows to efficiently construct the data structures (e.g. Bloom filters) needed for indexing terabase-scale collections of samples.

The components of `kmtricks`, along with an example execution, are presented in Fig. 1. We highlight the following features which differentiates `kmtricks` from related works:

- Joint k-mer counting allows to rescue large amounts of k-mers that would otherwise be discarded when processing samples independently.
- Direct counting of k-mer hash values instead of counting k-mers saves significant time for subsequent Bloom filter construction.
- `kmtricks` has been designed to be a stand-alone pipeline (Fig. 1), yet it is composed of modular tools (described in Section 4) which are of independent interest: partitioning a set of k-mers (according to their minimizer), jointly count k-mers, construct k-mer matrices, transpose them, and construct Bloom filters.
- `kmtricks` also provides a C++ library for interfacing with any stage of the pipeline, enabling for instance downstream sequence analyses based on streaming a k-mer matrix in row-major order.

We evaluated the performance of `kmtricks` in terms of running time, peak memory usage and maximal disk space on 100 and 674 RNA-seq samples (see Supplementary Material, Table S1). On these collections, `HowDe-SBT/kmtricks` is 1-1.5x faster to construct than `HowDe-SBT/KMC`, 3-4x faster than `HowDe-SBT/Jellyfish`, 7x faster than `COBS`, 2x faster than `Mantis`. Thus `kmtricks` yields superior or comparable performance to other methods for indexing sequencing data, even though it performs the more complex operation of joint k-mer counting. We will show in Section 3.3 that the performance gap between other methods and `kmtricks` further widens on larger inputs, i.e. terabyte-sized collections.

3.2 Dataset and computing setup

The Tara Ocean experiments (Section 3.3) were performed on large and complex sea water metagenomic data composed of 241 samples (distinct locations) by the Tara Ocean project (Karsenti *et al.*, 2011). This dataset is composed of approximately 6.5 thousand billion nucleotides, consisting of around 266 billion distinct k-mers ($k = 20$), among which 174 billions k-mers occur twice or more as estimated by `ntCard` (Mohamadi *et al.*, 2017). Executions were performed on a TGCC node with 4x16-cores Intel Xeon E7-8860 2.20 GHz with 3 TB of memory, on a SDD with 4.5 GB/s and 800 MB/s sequential read write, using 60 threads. Description of the

data, tool versions and command lines are provided in a companion Github website (see reference (Lemane and Peterlongo, 2021)).

3.3 Scaling to a large sea water metagenome collection

	Time (min)	Memory (GB)	Disk (TB)
`kmtricks`	2631	50.3	6.29
`Jellyfish` [a] + `makebf`	>10000[b]	80.6	≈ 1.1
`KMC` [a] + `makebf`	>8500[b]	213	≈ 1.1

Table 1. Comparison of construction times between `kmtricks` and other k-mer counting tools on the 6.5 terabases Tara Ocean collection. The `makebf` step corresponds to Bloom filter creation from counted k-mers by `howdesbt makebf`. The Memory column indicates peak RAM usage. `KMC` and `Jellyfish` counted each sample independently and removed k-mers with abundance one; whereas `kmtricks` performed join k-mer counting and low-abundance rescuing (see Section 4.3) which kept some of the unit abundance k-mers. `Mantis` and `COBS` were not executed due to their significantly longer construction times observed on smaller data.
[a] Stopped after 50h computation. [b] Extrapolated estimation.

`kmtricks` enabled to construct Bloom filters for a very large metagenomics collection with very limited amount of RAM and reasonable computation time, outperforming all other methods (Table 1). Disk usage was higher than other tools but of similar magnitude than the input data. `kmtricks` is ≥3.5 times faster than other pipelines, while achieving superior results as it performs joint k-mer counting and is able to rescue low-abundance shared k-mers.

We additionally ran `HowDe-SBT` on the Bloom Filters generated by `kmtricks`, thereby creating the first complete index of all metagenomics bacterial sequences obtained in the Tara Ocean project. With Bloom filters given as input, `HowDe-SBT` ran in 2100 minutes, with a peak RAM of 163 GB. The size of the final index is 612 GB. Of note, `kmtricks` executed in rescue mode but also discarding any k-mer seen only once takes twice less disk space (around 3 TB).

3.4 Collection-aware k-mer filtering recovers large amounts of weak signal present in complex metagenomes

k-mer filtering consists in removing from a sample any k-mer whose number of occurrences is below a certain threshold (called *solidity threshold*), classically set to 2 or 3. However, with data such as metagenomics or RNA-seq that have uneven coverage and include low abundance species or expressed genes, abundance does not enable to distinguish erroneous k-mers from real ones, as highlighted in Fig.2(a). Hence we propose to *rescue* low-abundant k-mers, through a *rare but shared k-mer rescue procedure*. This procedure consists in keeping any k-mer whose abundance is below the solidity threshold whenever this k-mer is sufficiently abundant in one or several other samples. Section 4.3 provides formalization and in-depth description of the procedure.

We evaluated the effect of the k-mer rescue procedure on the Tara Ocean experiment. Intermediate results and dedicated scripts are provided in the companion github web site (Lemane and Peterlongo, 2021). For each sample i, we computed the solidity threshold t_i as the smallest value ≥ 1 such that the number of k-mers occurring t_i times is smaller than 10% of the total number of k-mers. (using `ntCard` (Mohamadi *et al.*, 2017)). In a sample i, any k-mer with abundance higher or equal to t_i is conserved. Applying the rescue procedure, we then rescued any k-mer whose coverage was in a range $[1, t_i]$ whenever it had a coverage $> t_j$ in at least one other sample j. Hence a k-mer is considered as erroneous in a sample i if its abundance in i is lower than t_i and there exists no other sample j in which the k-mer is seen at abundance higher or equal to t_j.

We validated this strategy as follows. For each sample we computed

1. err_{th}: the theoretical expected number of erroneous k-mers,
2. err_{one}: the number of k-mers occurring only once, and
3. $err_{unrescued}$: the number of k-mers that are still considered as erroneous after our rescue procedure.

We then look at the ratio $err_{unrescued}/err_{th}$ and compare it to the ratio err_{one}/err_{th}. The closer a ratio is to one, the better.

The Tara Oceans dataset was mainly generated by HiSeq 2000 technology (222 samples out of 241), 8 samples were generated by HiSeq2500, 4 samples by GAIIx. For each of these technologies, we computed the theoretical error rate err_{th}. Given raw sequencing data from *Acinetobacter baylyi* generated by these three sequencing technologies[1] we counted the number of erroneous k-mers ($k = 20$), i.e. those absent from the reference genome. Error rates are respectively 0.0641%, 0.4838%, and 0.1715% for HiSeq2000, HiSeq2500 and GAIIx.

Results shown in Fig.2(b) highlight the importance and efficacy of our k-mer rescue procedure. Indeed, the quantity of k-mers filtered out is close to the theoretical expected value when using the rescue procedure (average ratio of 1.01). An order of magnitude too many k-mers appear to be wrongly filtered out when removing k-mers occurring only once (average ratio of 9.12).

Thus, when indexing low-error-rate data containing low-abundant genomes as in the Tara ocean bacterial metagenomes, k-mer rescuing appears to be essential as it: **1.** side-steps the issue of removing low-abundant k-mers which ends up discarding an order of magnitude too many k-mers; **2.** recovers a number of k-mers close to the expected one using co-occurrence across samples.

4 Methods

4.1 Definitions

A *minimizer* of length m within a sequence s is the smallest m-mer within s, where typically "smallest" is understood in the lexicographical sense. A *super-k-mer* is a sequence in which all constituent k-mers have the same minimizer.

A *Bloom filter* (Bloom, 1970) is an approximate membership query (AMQ) data structure that allows two operations: inserting and querying elements $u \in \mathcal{U}$. It is a bit array $B[0..n]$ with l hash functions $h_i : \mathcal{U} \rightarrow \{0, ..., n\}\ \forall i \in [1..l]$. Insertion can be defined as follows $B[h_i(x)] \leftarrow 1, \forall i \in [1..l]$ and lookup as $\bigwedge_{i=1}^{l} B[h_i(x)]$. Lookups can return false positives but no false negatives. In the following, we will consider one-hash Bloom filters ($l = 1$).

We use the terms *color-aggregative* and *k-mer-aggregative* as defined in (Marchet *et al.*, 2021). A color-aggregative data structure represents within a single index all k-mers of the collection, and each k-mer is associated to its pattern of presence/absence across the whole collection. Conversely, a k-mer-aggregative data structure constructs separate k-mer indices, one per sample.

We refer to *hash counting* as the process of counting hash values of a set of elements instead of counting the elements themselves. This is the counterpart of k-mer counting except that here k-mers are represented by their hash values, and several k-mers may collide to the same hash value.

The strand of each sequenced read being unknown, in `kmtricks`, as in all k-mer counting and indexing tools, each k-mer is represented by its canonical representation: the smallest string (in the lexicographic order) between itself and its reverse complement.

[1] Jean-Marc Aury, Genoscope, personal communication.

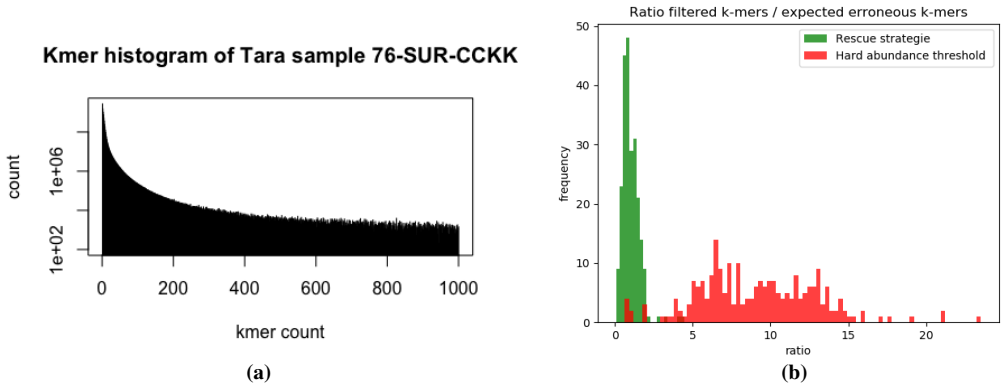

Fig. 2. (a) Kmer histogram of one of the Tara ocean samples (chosen arbitrarily), showing a flat distribution of abundances indicative of the presence of low-abundance microbes, also highlighting the lack of separation between erroneous and correct k-mers. (b) Number of k-mers filtered divided by the number of expected number of erroneous k-mers (ideal is close to 1). Green histogram shows results obtained with the proposed rescue procedure. Red histogram shows results obtained by the classical removal of k-mers occurring only once.

4.2 A modular pipeline for large-scale Bloom Filters construction: kmtricks

`kmtricks` supports the construction of either a k-mer matrix or Bloom filters. In both cases, the input is a collection of sequencing data files in FASTQ format. The output is either a matrix having k-mers as rows, samples as columns and k-mer counts as values, or a collection of Bloom filters, one per sample. In the following, we will focus on Bloom filter construction as this pipeline includes all the ingredients necessary for k-mer matrix construction. We will describe two operational modes: *hash counting* for Bloom filters, *k-mer counting* for k-mer matrices.

In other tools, the construction process of Bloom filters can typically be broken into two steps: 1) efficiently counting k-mers then 2) inserting distinct k-mers into filters, on a per-sample basis. `kmtricks` streamlines this process by realizing that in the case of Bloom filters only the hashes need to be counted, not k-mers; furthermore, in order to cope with terabytes of input data and still be able to efficiently count hashes, a careful partition-aware hashing scheme is designed.

4.2.1 Partitioning

`kmtricks` performs parallel k-mer counting following the classical paradigm of partitioning k-mers based on their minimizers and then constructing super-k-mers, as in `KMC 2` (Kokot *et al.*, 2017). However, the process is newly modularized so that intermediate tasks correspond to separate programs. Conceptually, the set of all possible minimizers is first partitioned in the following balanced way: all partitions should contain a roughly equal total number of k-mers. This is performed by the `km_minim_repart` module, using the GATB library (Drezen *et al.*, 2014) that implements a previously-known algorithm from DSK (Rizk *et al.*, 2013).

In hash counting mode, in order to take advantage of the partitioning scheme in the context of Bloom filters construction, we use *partitioned Bloom filters* (pBFs). These are Bloom filters that are partitioned into P sub-filters with exclusive (and consecutive) hash spaces $h_p : \mathcal{U}_p \rightarrow \{p \times s, ..., p \times s + s - 1\}$ with $p \in [0..P-1]$ and s = $\left\lceil \frac{bits}{P} \right\rceil$ (rounded up to 8) with "$bits$" corresponding to the user-requested Bloom filter size. Having pBFs allows to populate only a small part of a Bloom filter when processing a k-mer partition, which saves memory and enables coarse-grained parallelization at both the construction and query stages. A classical Bloom filter (except for a slightly more complex query operation, see Section 4.2.5) can be obtained by a simple concatenation of the pBFs thanks to the consecutive hash spaces.

4.2.2 Counting

The second step, performed in `km_reads_to_superk`, consists in computing for each sample its super-k-mers and writing them to their corresponding partitions on disk (Fig 3.a). From those super-k-mers, k-mers (or their hash values, depending on whether hash counting is performed) are de-duplicated and the abundance of each distinct k-mer (or hash value) is determined within each partition (`km_superk_to_kmer_counts` module).

In hash counting mode, `kmtricks` optimizes the output in the following way. If k-mer rescue (see 3.4) is not performed, bit-vectors are output immediately instead of a list of counted hashes (Fig 3.1.b). One bit vector is output per partition and per sample. In other words, these bit-vectors correspond to pBFs built from the hashes. Otherwise if k-mer rescue is performed (still in hash counting mode), bit-vectors cannot be immediately output as counts of the same hash value must be examined over all samples. In this case, hashes and their counts are dumped to disk for each partition from each sample (Fig. 3.2.b).

4.2.3 Merging

Finally if k-mer rescue is performed or a holistic view of k-mers (i.e. joint counting) is sought, k-mers (resp. partitions of hash values) need to undergo a merging step in order to obtain a k-mer matrix (resp. a collection of Bloom filters). Aggregation of k-mers or hashes over multiple samples is achieved using the classical k-way merge algorithm on equivalent partitions between samples. This algorithm assumes sorted inputs, which is inexpensive in our case since sorting is already performed by the counting algorithm. Merging is performed in the `km_merge_within_partition` module.

In both k-mer counting mode and hash counting mode, each row count vector (corresponding to a single k-mer or hash value) is processed according to the k-mer rescue procedure, details are given in Section 4.3.

In k-mer mode, there is no additional operation after merging, except output formatting (see Section 4.2.4). Therefore the rest of this section is dedicated to the hash counting mode, i.e. the Bloom filters construction pipeline.

In hash counting mode, row count vectors are transformed into a binary representation during the merge step. In a partition, all possible hashes are considered. This means that for each missing hash value (corresponding to a k-mer not seen in the partition), an empty bit-vector is appended to the matrix. Hashes are not stored (only the bit-vectors are), as they implicitly correspond to row indices. At the end of this step, we have P sub-matrices of $\left\lceil \frac{bits}{P} \right\rceil$ presence/absence bit-vectors each.

At this point, the resulting matrices (k-mers matrices or Bloom filters) are color-aggregative, i.e. each row represents the presence or the absence (or counts) of the corresponding hash value across samples. If `kmtricks` is set to construct Bloom filters, one seeks to convert the data into k-mer-aggregative, where each filter represents hashes for a single collection. Switching from a color-aggregative representation to a k-mer-aggregative representation can be achieved through a bit-matrix transposition. The input matrix needs to be exhaustive in the following sense: missing hashes are represented by empty presence/absence bit-vectors, i.e. in each partition the number of presence/absence bit-vectors corresponds to s, the number of bits in a partition, as described in Section 4.1. It also needs to be sorted so that each bit-vector row corresponds to consecutive hashes in $\{p \times s, ..., p \times s + s - 1\}$.

When performing a transposition, we transform a matrix with hashes in rows associated with presence/absence bit-vectors into a matrix with samples in rows associated with a one-hash pBF. Due to Bloom filter partitioning, P transposed matrices are in fact obtained, each with a number of rows corresponding to the number of samples. The horizontal concatenation of each corresponding row from these matrices allows one to build one Bloom filter per sample.

4.2.4 Outputs

`kmtricks` can output different sort of k-mer matrices. In k-mer mode: count or presence/absence k-mer matrices. In hash mode: hashes presence/absence vectors (kmer-aggregative) or pBFs vectors (color-aggregative), both seen as bit matrices. Of note, pBFs vectors can be converted into sample-specific Bloom filters that are compatible with the SDSL library (Gog *et al.*, 2014) and `HowDe-SBT` (`km_output_-convert` module).

All these outputs are readable and writable using `kmtricks` C++ library. In the case of k-mer counts matrix, a text output format is also supported.

4.2.5 Query

Due to how `kmtricks` creates Bloom filters that are partitioned according to minimizers, the minimizer of each k-mer from a query sequence must be computed in order select the correct hash function. A compatible version of `HowDe-SBT` is available through the `kmtricks` release. `kmtricks` does not yet provide a stand-alone tool to directly query a Bloom filter, however this operation is supported within the `kmtricks` C++ library.

4.3 A novel technique to rescue rare k-mers

To rescue low-abundant but likely correct k-mers, as performed in Section 3.4, we design a rather simple technique based on examining the abundance of each k-mer across sequencing samples. This technique is only practically applicable in conjunction with joint k-mer counting. It cannot be directly implemented in a one-sample-at-a-time construction procedure, unless such procedure discards no k-mer which would result in prohibitively large intermediate storage. When a low-abundant k-mer is observed in a sample (with abundance lower than a user-defined threshold), its abundance in other samples is used to decide whether to keep its abundance for that sample or not. Several thresholding procedures can be specified in `kmtricks`:

- A hard threshold applied during the counting step, called `count-abundance-min`. This threshold filters out any k-mer whose abundance is below its value, regardless of the presence of the k-mer in other samples. Kept k-mers are said to be *solid*. Note that this threshold can be set on a per-sample basis.
- A soft threshold, this time applied during the merging step, called `merge-abundance-min`. Any k-mer whose abundance is higher than its value is kept, otherwise it is provisionally discarded yet is considered candidate for being rescued. We call such a k-mer *rescue-able*. This threshold can be set on a per-sample basis.
- The `merge-abundance-min` threshold is modulated by a parameter that we call `save-if`. A *rescue-able* k-mer in a sample is kept (rescued) if it is *solid* in at least `save-if` other sample(s).
- The last parameter, `recurrence-min`, allows to discard k-mers that occur in less than `recurrence-min` distinct samples. Such k-mers are simply removed from the final matrix (i.e. result in a null bit-vector in hash mode).

Figure 4 presents several examples showing the application of those thresholds.

	D0	D1	D2	D3	D4						
m-a-min	**3**	**2**	**2**	**3**	**2**						
ⓐ **H1**	2	0	4	5	3	→	1	0	1	1	1
ⓑ **H2**	4	1	6	2	1	→	1	0	1	0	0
ⓒ **H3**	2	8	1	2	1	→	-/0	-/0	-/0	-/0	-/0

Fig. 4. Example of the rescue procedure for three k-mers/hashes and five samples using sample-specific `merge-abundance-min` and the following set of parameters: `count-abundance-min=1`, `save-if=3`, `recurrence-min=2`. ⓐ H1 has a abundance lower than 3 in D0 but it is solid in at least `save-if` samples (D2, D3, D4). ⓑ H2 is non-solid on D1, D3 and D4 and is solid only in 2 samples, H2 is therefore discarded in D1, D3 and D4. ⓒ H3 is solid only in one sample, `recurrence-min` cannot be satisfied, the whole row is therefore discarded (dash signs in the Figure, or corresponds to the null bit-vector in hash mode).

5 Discussion

We propose a novel method for efficiently counting k-mers across multiple samples and for generating Bloom filters. In addition to being the fastest method for generating Bloom filters over terabyte-scale collections, our approach `kmtricks` proposes a novel mechanism to filter erroneous k-mers using their co-occurrence across samples, i.e. going beyond filtering on a per-sample basis. This approach leads to significantly improved recovery of k-mers in metagenomes.

In our tests on relatively small collections (100-674 RNA-seq datasets, with on average hundreds of millions of distinct k-mers per sample) the performance of `kmtricks` is roughly equivalent to the state of the art `KMC` k-mer counter combined with the Bloom filters construction procedure of `HowDe-SBT`. However `kmtricks` stands out on larger collections having higher number of k-mers per sample, such as Tara Ocean ($>$ 6 TB of sequences, with several billions of distinct k-mers per sample). For those, Bloom filter construction becomes a bottleneck and highlights the superior efficiency of the streamlined `kmtricks` pipeline.

At a high level, `kmtricks` is able to output matrices either in column-major order or in row-major order, where rows can either be k-mers or hash values. This flexibility allows `kmtricks` to provide inputs for both types of indexing data structures: k-mer-aggregative and color-aggregative (as defined in Marchet *et al.* (2021) and recalled in the Methods section). Row-major order makes the presence/absence of a k-mer directly accessible across all samples, and is contiguous in memory. In column-major order, each Bloom filter is independent and provides information about the existence of all k-mers in one sample.

Contemporary of `kmtricks`, the MetaGraph software (Karasikov *et al.*, 2020) is a k-mer indexing structure that represents k-mers exactly (i.e. not using a Bloom filter) and does not support creating k-mer matrices. MetaGraph was applied to very large collections (hundreds of terabases)

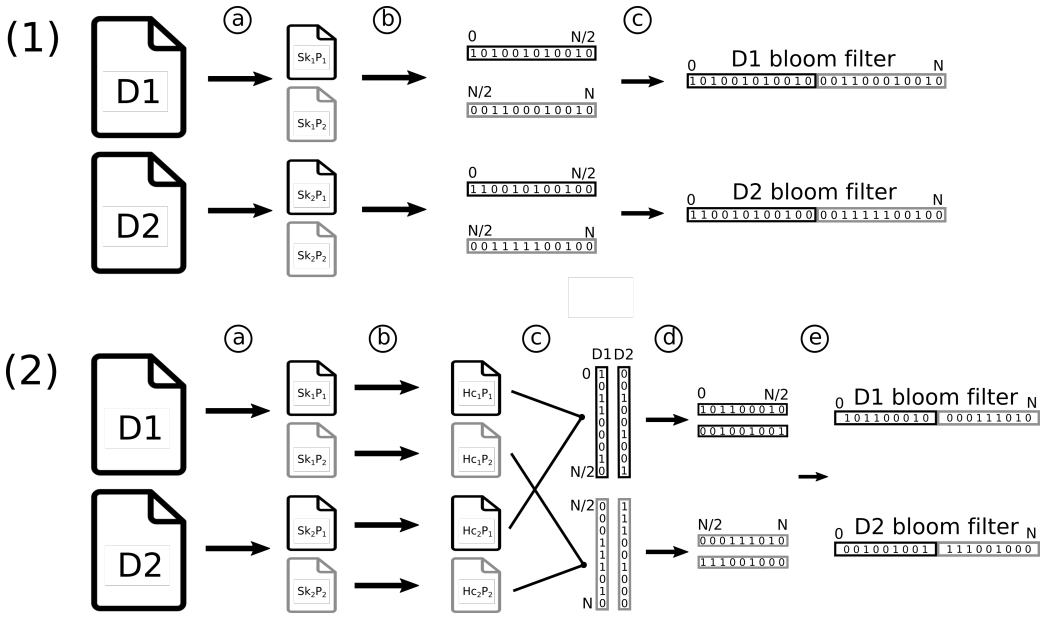

Fig. 3. Bloom filters construction pipeline with two samples D1 and D2 using two partitions: black (1) and gray (2). Sk and Hc denote respectively super-k-mers and hash counted (1) Bloom filters pipeline without k-mer rescue: ⓐ Divide sample into partitioned super-k-mers. ⓑ Split super-k-mers into k-mers before hashing them and counting hashes in partitions. For each partition, output presence/absence bit-vectors, i.e. partitioned Bloom filters. ⓒ Concatenate equivalent partitions between samples to obtain one Bloom filter per sample. (2) Bloom filters pipeline with rare k-mer rescue: ⓐ same as (1). ⓑ same as (1), but output hashes and their counts. ⓒ Merge and binarize (according to the rescue procedure, see 4.3) equivalent partitions to build one sub-matrix per partition with pBFs in columns. ⓓ Transpose sub-matrices to obtain pBFs in rows. ⓔ same as (1)-ⓒ.

using cloud resources and KMC 3, which makes its contribution orthogonal to `kmtricks`, the latter being geared towards making computation more efficient on a single server. Ideally, `kmtricks` could be integrated within a MetaGraph-like approach to combine single-server efficiency within a cloud architecture.

Acknowledgements

This work used HPC resources from the Très Grand Centre de Calcul of CEA (`http://www-hpc.cea.fr/fr/complexe/tgcc.htm`) and the GenOuest bioinformatics core facility (`https://www.genouest.org`). The authors are grateful to Bob Harris for discussion on HowDeSBT, and Eric Pelletier & Jean-Marc Aury who provided links to Tara and *Acinetobacter* datasets, and precious information about these data.

Funding

The work was funded by IPL Inria Neuromarkers, ANR Inception (ANR-16-CONV-0005), ANR Prairie (ANR-19-P3IA-0001), ANR SeqDigger (ANR-19-CE45-0008).

References

Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. (1990). Basic local alignment search tool. *Journal of molecular biology*, **215**(3), 403–410.

Audoux, J., Philippe, N., Chikhi, R., Salson, M., Gallopin, M., Gabriel, M., Le Coz, J., Drouineau, E., Commes, T., and Gautheret, D. (2017). De-kupl: exhaustive capture of biological variation in rna-seq data through k-mer decomposition. *Genome biology*, **18**(1), 243.

Benoit, G., Peterlongo, P., Mariadassou, M., Drezen, E., Schbath, S., Lavenier, D., and Lemaitre, C. (2016). Multiple comparative metagenomics using multiset k-mer counting. *PeerJ Computer Science*, **2016**(11), e94.

Bingmann, T., Bradley, P., Gauger, F., and Iqbal, Z. (2019). COBS: a Compact Bit-Sliced Signature Index. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, **11811 LNCS**, 285–303.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, **13**(7), 422–426.

Bradley, P., den Bakker, H. C., Rocha, E. P., McVean, G., and Iqbal, Z. (2019). Ultrafast search of all deposited bacterial and viral genomic data. *Nature Biotechnology*, **37**(2), 152–159.

Buchfink, B., Xie, C., and Huson, D. H. (2015). Fast and sensitive protein alignment using diamond. *Nature methods*, **12**(1), 59–60.

Drezen, E., Rizk, G., Chikhi, R., Deltel, C., Lemaitre, C., Peterlongo, P., and Lavenier, D. (2014). GATB: Genome Assembly & Analysis Tool Box. *Bioinformatics (Oxford, England)*, **30**(20), 2959–2961.

Gog, S., Beller, T., Moffat, A., and Petri, M. (2014). From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337.

Harris, R. S. and Medvedev, P. (2019). Improved representation of sequence Bloom trees. *Bioinformatics*.

Karasikov, M., Mustafa, H., Danciu, D., Zimmermann, M., Barber, C., Rätsch, G., and Kahles, A. (2020). MetaGraph: Indexing and Analysing Nucleotide Archives at Petabase-scale. *bioRxiv*, page 2020.10.01.322164.

Karsenti, E., Acinas, S. G., Bork, P., Bowler, C., De Vargas, C., Raes, J., Sullivan, M., Arendt, D., Benzoni, F., Claverie, J.-M., *et al.* (2011). A holistic approach to marine eco-systems biology. *PLoS biol*, **9**(10), e1001177.

Kokot, M., Dlugosz, M., and Deorowicz, S. (2017). KMC 3: counting and manipulating k-mer statistics. *Bioinformatics (Oxford, England)*, **33**(17), 2759–2761.

Lappalainen, T., Sammeth, M., Friedländer, M. R., Ac't Hoen, P., Monlong, J., Rivas, M. A., Gonzalez-Porta, M., Kurbatova, N., Griebel, T., Ferreira, P. G., *et al.* (2013). Transcriptome and genome sequencing uncovers functional variation in humans. *Nature*, **501**(7468), 506–511.

Lemane, T. and Peterlongo, P. (2021). `https://github.com/pierrepeterlongo/kmtricks_benchmarks`.

Li, Y. *et al.* (2015). Mspkmercounter: a fast and memory efficient approach for k-mer counting. *arXiv preprint arXiv:1505.06550*.

Manekar, S. C. and Sathe, S. R. (2018). A benchmark study of k-mer counting methods for high-throughput sequencing. *GigaScience*, **7**(12), giy125.

Marçais, G. and Kingsford, C. (2011). A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, **27**(6), 764–770.

Marchet, C., Iqbal, Z., Gautheret, D., Salson, M., and Chikhi, R. (2020). Reindeer: efficient indexing of k-mer presence and abundance in sequencing datasets. *bioRxiv*.

Marchet, C., Boucher, C., Puglisi, S. J., Medvedev, P., Salson, M., and Chikhi, R. (2021). Data structures based on k-mers for querying large collections of sequencing data sets. *Genome Research*, **31**(1), 1–12.

Mason, C., Afshinnekoo, E., Ahsannudin, S., Ghedin, E., Read, T., Fraser, C., Dudley, J., Hernandez, M., Bowler, C., Stolovitzky, G., Chernonetz, A., Gray, A., Darling, A., Burke, C., Łabaj, P. P., Graf, A., Noushmehr, H., Moraes, s., Dias-Neto, E., Ugalde, J., Guo, Y., Zhou, Y., Xie, Z., Zheng, D., Zhou, H., Shi, L., Zhu, S., Tang, A., Ivanković, T., Siam, R., Rascovan, N., Richard, H., Lafontaine, I., Baron, C., Nedunuri, N., Prithiviraj, B., Hyat, S., Mehr, S., Banihashemi, K., Segata, N., Suzuki, H., Alpuche Aranda, C. M., Martinez, J., Christopher Dada, A., Osuolale, O., Oguntoyinbo, F., Dybwad, M., Oliveira, M., Fernandes, A., Oliveira, M., Fernandes, A., Chatziefthimiou, A. D., Chaker, S., Alexeev, D., Chuvelev, D., Kurilshikov, A., Schuster, S., Siwo, G. H., Jang, S., Seo, S. C., Hwang, S. H., Ossowski, S., Bezdan, D., Udekwu, K., Udekwu, K., Lungjdahl, P. O., Nikolayeva, O., Sezerman, U., Kelly, F., Metrustry, S., Elhaik, E., Gonnet, G., Schriml, L., Mongodin, E., Huttenhower, C., Gilbert, J., Hernandez, M., Vayndorf, E., Blaser, M., Schadt, E., Eisen, J., Beitel, C., Hirschberg, D., Schriml, L., Mongodin, E., and Consortium, T. M. I. (2016). The Metagenomics and Metadesign of the Subways and Urban Biomes (MetaSUB) International Consortium inaugural meeting report. *Microbiome*, **4**(1), 24.

Mohamadi, H., Khan, H., and Birol, I. (2017). ntcard: a streaming algorithm for cardinality estimation in genomics data. *Bioinformatics*, **33**(9), 1324–1330.

Muggli, M. D., Alipanahi, B., and Boucher, C. (2019). Building large updatable colored de bruijn graphs via merging. *Bioinformatics*, **35**(14), i51–i60.

Nurk, S., Meleshko, D., Korobeynikov, A., and Pevzner, P. A. (2017). MetaSPAdes: A new versatile metagenomic assembler. *Genome Research*, **27**(5), 824–834.

Pandey, P., Almodaresi, F., Bender, M. A., Ferdman, M., Johnson, R., and Patro, R. (2018). Mantis: A Fast, Small, and Exact Large-Scale Sequence-Search Index. *Cell Systems*, **7**(2), 201–207.e4.

Rizk, G., Lavenier, D., and Chikhi, R. (2013). DSK: k-mer counting with very low memory usage. *Bioinformatics*, **29**(5), 652–653.

Solomon, B. and Kingsford, C. (2016). Fast search of thousands of short-read sequencing experiments. *Nature Biotechnology*, **34**(3), 300–302.

Song, L. and Florea, L. (2015). Rcorrector: efficient and accurate error correction for illumina rna-seq reads. *GigaScience*, **4**(1), s13742–015.

Turnbull, C., Scott, R. H., Thomas, E., Jones, L., Murugaesu, N., Pretty, F. B., Halai, D., Baple, E., Craig, C., Hamblin, A., *et al.* (2018). The 100 000 genomes project: bringing whole genome sequencing to the nhs. *Bmj*, **361**.

Supplementary

Human RNA-seq benchmarks

The human RNA-seq benchmarks were done on two subsets with 100 and 674 samples from a common set of 2,585 human RNA-seq sequencing used as inputs in several k-mer indexing benchmarks, and first proposed in Solomon and Kingsford (2016). Computations were performed on the GenOuest platform on a node with 4x8-cores Xeon E5-2660 2,20 GHz with 200 GB of memory. Benchmarks were performed on SSD disk with 900 MB/s and 290 MB/s sequential read/write. All benchmarks are done using 20 cores. Details about data and scripts are available from the `kmtricks` github companion website (see reference Lemane and Peterlongo (2021)). A Conda environment is also provided to reproduce these benchmarks.

On these datasets, `kmtricks` outperformed the indexing steps of `Mantis`, `HowDe-SBT` and `COBS` in terms of computing time (by 2-8x) and memory usage (by 1-8x), and uses comparable disk space. We also substituted `Jellyfish` with `KMC` in `HowDe-SBT`, yielding comparable time/memory performance to `kmtricks` on this collection. However, `KMC` does not support joint k-mer counting, and its integration in a Bloom filter construction pipeline turns out to be significantly less scalable than `kmtricks` as shown Section 3.3, dealing with larger and more complex data.

A : 100 RNA-seq (44 GB fasta.gz)

	Time (min)	Memory (GB)	Disk (GB)
makebf^{*,1} - HowDe-SBT	147 + 21	13.2 | 2.6	55.1
makebf$^{+,1}$ - HowDe-SBT	33 + 21	2.9 | 2.6	28.4
McCortex^{2} - COBS	256 + 67	27 | 1.5	327
Squeakr^{1} - Mantis	64 + 24	3.6 | 27.8	25.8
kmtricks ^{1} - HowDe-SBT	34 + 21	4.1 | 2.6	30.3
kmtricks ^{1,R} - HowDe-SBT	36 + 21	3.3 | 2.6	54
kmtricks ^{2,R} - HowDe-SBT	32 + 21	3.1 | 2.6	53.9

B : 674 RNA-seq (961 GB fasta.gz)

	Time (min)	Memory (GB)	Disk (GB)
makebf^{*,1}	3543	13.2	206
makebf$^{+,1}$	1958	18.7	165
kmtricks ^{1}	1206	21.6	233
kmtricks ^{1,R}	1246	17.4	327

Table S1. Benchmarks on two datasets. The `makebf` step corresponds to Bloom filter creation by `howdesbt makebf`, k-mers being counted either with `Jellyfish` (* symbol) or `KMC` ($^{+}$ symbol). For Time and Memory, when two value are provided, the first corresponds to the pre-processing time (Bloom filters creation) and the second to the index construction. Memory and Disk correspond to the peak. A: Disk usage corresponds to the total required space to build the index, including temporary files, Bloom filters and the final index. For the couple `McCortex-COBS`, the disk usage corresponds mainly to the ctx files from `McCortex`. B: Disk usage corresponds to the count step, including temporary files and Bloom filters build by `howdesbt makebf` or `kmtricks`.
^{1}k=20, ^{2}k=31, ^{R}Rescue mode