Scalable sequence database search using Partitioned Aggregated Bloom Comb-Trees

Camille Marchet, Antoine Limasset

Scalable sequence database search using Partitioned Aggregated Bloom Comb-Trees

Camille Marchet[1*]
Antoine Limasset[1]
[1]Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Abstract

A public database such as the SRA (Sequence Read Archive) has reached 30 peta-bases of raw sequences and doubles its nucleotide content every two years. While BLAST-like methods can routinely search a sequence in a single genome or a small collection of genomes, making accessible such immense resources is out of reach for alignment-based strategies. In the last years, an abundant literature tackled the fundamental task of locating a sequence in an extensive dataset collection by converting the query and datasets to k-mers sets and computing their intersections. Among those methods, approximate membership query data structures conjugate the ability to query small signatures or variants while being scalable to collections of thousands of sequencing experiments. However, at present, more than 10,000 eukaryotic samples are not analyzable in reasonable time and space frames. Here we present PAC, a novel approximate membership query data structure for querying collections of sequence datasets. PAC presents several advantages over the state-of-the-art, enabling users to scale to the next order of magnitude. PAC index construction works in a streaming fashion without any disk footprint besides the index itself. It shows a 3 to 6 fold improvement in construction time compared to other compressed methods for comparable index size. Using inverted indexes and a novel data structure dubbed aggregative Bloom filters, a PAC query can need single random access and be performed in constant time in favorable instances. Thanks to efficient partitioning techniques, index construction and queries can be performed in parallel with a low memory footprint. Using limited computation resources and in five days, we built a PAC index that included more than 30,000 human RNA-seq samples (corresponding to more than 40 billion distinct k-mers) to assess PAC's scalability. We also showed that PAC's ability to query 500,000 transcript sequences in less than an hour. The only bottleneck being the index size, PAC should scale on hundred of thousand datasets on a regular cluster. PAC is open-source software available at `https://github.com/Malfoy/PAC`.

*Corresponding Author. Email: camille.marchet@univ-lille.fr

1 Introduction

Public databases such as the SRA (Sequence Read Archive) or ENA (European Nucleotide Archive) overflow with sequencing data. The vast amount of sequences, experiments, and species allows in principle ubiquitous applications for biologists and clinicians. Such databases are becoming a fundamental shared community resource for daily sequence analysis. But concretely, we only witness the onset their exploitation, while their exponential growth poses serious scalability challenges (for instance, SRA has reached 30 Peta-bases of raw reads and roughly doubles every two years [1]).

Today, searching a query sequence in a single genome/dataset or a restrained collection of genomes is considered routine through alignment-based tools as BWA-MEM [2], BLAST [3], BEETL-fastq [4], population BWT [5] or SRA-BLAST [6]. On the contrary, the scale of databases such as SRA makes BLAST and any alignment-based method ill-suited because of the prohibitive cost of the alignment phase. Broader usages of such resources necessitate algorithms allowing hyper-scalable membership queries as a foundation. Therefore recently, abundant literature tackled the fundamental task of locating a sequence in an extensive dataset collection using alignment-free, k-mer based strategies.

The known most scalable solutions are **sketching methods**, which are based on the principle of locality-sensitive hashing. However, the loss of resolution implied by these methods narrows the query possibilities to large queries of the order of magnitude of genomes or larger. The query size is significantly smaller for numerous applications, e.g., variant calling of SNP/small indels, finding alternative splicing sites, or other small genomic signatures.

Thus, recent alignment-free literature has described novel methodologies to fill a dual need: small-sized queries in vast datasets collections. Mainly two paradigms cover this question. Both consider each dataset and the query itself as k-mer sets (the distinction between the different approaches are summarized in Figure 1). The first type of approach relies on **exact k-mer set representations**. These methods build an associative index (using hash tables or FM-indexes) where the key set corresponds to all k-mers of the collection. K-mers are associated with bit-vectors that indicate their presence/absence in the different datasets. Jointly, some methods implement a co-assembly graph over the datasets. These methods (that we denoted as *color aggregative* in a previous survey [7]) are interesting for queries with high precision requirements or when the graph structure is needed. Since exact representation is costly, color aggregative methods proposed different optimizations to increase their scalability. The main directions have been: efficient representations for the key set (i.e., k-mers) [8, 9], key set compression via self-indexing [10], optimization of the values space requirements using minimal perfect hashing or other hashing techniques [11, 8, 9, 12], compression of the presence/absence information [13, 14] (and combinations of these strategies). However, we observe that this approach hardly scales to extensive instances with very large k-mer cardinality, and the literature shows they are currently limited to ∼2,500 eukaryotic datasets.

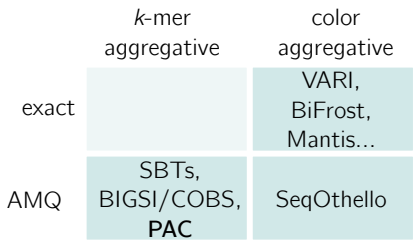

Figure 1: A 2-dimensional summary of the sets of k-mer sets indexing literature paradigms. We report examples of methods in each category.

The second family of methods relies on probabilistic set representation. These **approximate membership query (AMQ) structures** trade false positives during the query for improved speed

and memory performances in comparison to the previous category. The methods discussed in this paper fall in the category of AMQ structures. Unlike colored-aggregative methods, most AMQ methods separately represent each dataset's k-mers using Bloom filters. Bloom filters combine a very space-efficient set representation with the ability to stream k-mers for their construction and constant time queries. K-mers are then aggregated in a data structure allowing queries on the totality of the Bloom filters (hence we named these methods *k-mer aggregative* in [7]). Finally, a unique approach was proposed in SeqOthello [15]. While it is conceptually close to color aggregative methods, as it works on the entire set of distinct k-mers from the collection of datasets, SeqOthello is also an AMQ relying on a different hashing strategy than Bloom filters.

AMQ approaches enabled indexing hundreds of thousands of microbial datasets [16]. For mammalian datasets, scaling to more samples is a timely challenge since the increase of k-mer/datasets content poses serious issues for the construction/footprint of current methods. Thus, to our knowledge, no published method has yet overcome the barrier set by SeqOthello [15] of indexing over 10,000 mammalian RNA samples. This manuscript proposes a novel AMQ method for querying biological sequences in collections of datasets dubbed PAC (Partitioned Aggregative Bloom Comb Trees). In our application case, queries are typically alternative splicing events or a short genomic context around a small variant or mutation. Our structure is designed to be highly scalable with the number of indexed samples and requires moderate resources.

2 Methods

The method presented in this manuscript is a novel AMQ structure. First, we recall some basic definitions in Section 2.1. Thus, in the "cheat-sheet" section (2.2), we recall some algorithmic aspects of related AMQ methods from the literature in order to clarify further discussions. We summarize important aspects in Table 2.2.3. Finally, in section 2.3, we present PAC, our structure.

2.1 Preliminary definitions

A *set of sequences* R (also called *dataset*) is a set of finite strings over the alphabet $\Sigma = \{A, C, G, T\}$. In practice sets of sequences can be read sets or genome sets. K-mers are strings of fixed length $k > 0$ over these sets. An *input data-base* $D = \{R_1, \dots R_n\}$ is a set of *datasets*.

Similarly to the datasets, a *query* is a finite string from which all distinct k-mers are extracted (we denote the query of a single k-mer a *single query*).

Problem statement: The structures described hereafter compute the cardinality of the intersection of each dataset's k-mers with the query's k-mers. Then, according to a threshold parameter τ, the query is said to be *in* a dataset if their intersection is larger or equal to τ.

A *Bloom filter* (BF) is a space-efficient data structure for inexact set representation. For BFs, inexact means that queries to such a representation have a controlled rate of false positives and no false negatives. A BF has two parameters (b, h), which define the size of a bitvector and the number of hash functions of the filter. A Bloom filter maps each set element in the bitvector using h hash functions, and sets mapped bits to 1 (initially, all bits are 0's). The query follows the same principle and verifies that all hashed locations contain a 1. False positives on foreign keys happen because of hash collisions (which can be controlled by tuning (b, h)).
Bloom filters' usage in this work: for fixed parameters (k, b, h), for each R_i $(0 < i \leq n)$ in D, we build a Bloom filter BF_i and populate BF_i with R_i's k-mers. Thus, each Bloom filter represents

the k-mers of a dataset from D. We later describe how Bloom filters are organized to optimize the space, time, and query.

Minimizers [17] and super-k-mers: the m-minimizer of a $k-$mer S ($m < k$) is the smallest hashed substring of size m in S. As consecutive k-mers in a sequence largely overlap, blocs of k-mers tend to share their minimizer. These blocs are called super-k-mers [18]. As hashing tends to distribute minimizers evenly in $[0, m-1]$ space (and therefore, the k-mers they come from), minimizers and super-k-mers can be used to divide a k-mer set in m partitions. See supplemental Figure S1 in Appendix for an example.

PAC's overview: PAC's structure is based on 1- tree structures organizing BFs for efficient query with small space/time requirements (described in Sections 2.3.1 and 2.3.2), and 2- a partitioning strategy based on minimizers and super-k-mers (in Section 2.3.3).

2.2 Sequence sets of sets AMQ structures cheat-sheet

We start by stating intuitive definitions for each type of structure. **A sequence Bloom tree (SBT)** is an approximate membership query structure introduced in [19]. For a set of n Bloom filters built over an input database D, a SBT is a balanced binary tree that aggregates the k-mer contents in the different datasets of the input. We call a **sequence Bloom matrix** the matrix of BFs introduced in BIGSI [20]. A Bloom filter is built for n datasets in an input database and becomes a matrix column. **A SeqOthello** is an AMQ structure relying on a different paradigm than sequence Bloom trees and matrices. It maps any distinct k-mer from the whole database to their presence/absence in datasets using compressed bit-vectors v. Pairs of (k-mer, vector) are associated using a static hashing strategy.

2.2.1 Operations

Construction In sequence Bloom trees, the Bloom filters corresponding to the datasets are in the tree's leaves, and each internal node contains the union Bloom filter of its two children nodes (which is equivalent to a bitwise OR of the two children). All Bloom filters are built using the same parameters (b, h). Thus, the tree's root represents the union of all distinct k-mers in D. Construction of recent SBT's improvements [21, 22] (including the latest, HowDeSBT) involves a prior clustering of the leaves to obtain a tree's fill-up favorable to efficient compression. Sequence Bloom matrices use the property that all Bloom filters are constructed with the same parameters to stack them and obtain an inverted index on the BFs. Thus, a row accesses the same bit position in n Bloom filters. In SeqOthello, an associative index handles (k-mer, presence/absence vector) pairs, built by combining two passes of Othello hashing. In short, Othello hashing is a static hashing strategy, which creates a surjection for sets of sets so that each key (here k-mer) is associated with a value corresponding to its origin set. SeqOthello hashes each k-mer to a bucket, then to an index in the bucket using this associativity. The index leads to a presence/absence vector. The static nature of the hashing makes it prone to false positives when querying foreign keys. All three methods require dataset processing (converting datasets to BFs for sequence Bloom trees and matrices, converting datasets to k-mer sets for SeqOthello). Once it is done, the index construction itself is in $\mathcal{O}(n)$ time.

Insertion of a new dataset While in theory, SBTs can be updated in $\mathcal{O}(height \times b)$ ($height$ being the tree height), in practice, the leave clustering might make it necessary to rebuild the whole

structure from time to time. In theory, sequence Bloom matrices can be updated in $\mathcal{O}(b)$ time by adding a new Bloom filter to the others and updating the inverted index. From a practical point of view, neither the latest SBT nor matrix provide an insertion operation in their implementation. SeqOthello's update is implemented and in $\mathcal{O}(n)$, as it requires to re-compute the hash function.

Query In any case, the worst-case time complexity for queries is in $\mathcal{O}(n)$. Thus, the number of random access impacts is critical for runtimes in practice. In the particular case where a single query is present in a single dataset, sequence Bloom tree structures allow a $\mathcal{O}(log(n))$ random accesses. We can note that false positives will increase the number of accesses in the case of SBT queries as they wrongly lead to descending in more subtrees. Methods using inverted indexing strategy, i.e., SeqOthello and COBS, share the guarantee to perform $\mathcal{O}(1)$ "costly" random accesses. This guarantee is an improvement upon SBTs, which can need a large number of random accesses according to the number of hits and their locations. However, COBS or SeqOthello still may need to read a bit slice of size $\mathcal{O}(n)$.

2.2.2 Implementation

Bloom filters settings and false positives In sequence Bloom trees structures, the BF size is adapted to all distinct k-mers over the database. In matrix structures, the BF size is chosen according to the largest k-mer set among datasets. However, a single size does not fit all Bloom filters in the general case. Therefore, COBS [23], an improvement of BIGSI, Bloom filters are folded in order to adapt to datasets of different sizes.

Motivated by [19] work on false-positive tradeoffs, both SBT and matrix approaches chose to work with a single hash function to optimize the query time. The BF sizes can be set to control the average false positive rate for SBT and matrix structure. It is conversely not possible to control it in SeqOthello.

Compression The SBT structure implies redundancy in the internal nodes, following the consecutive unions of Bloom filters. This redundancy motivated the application of compression schemes in SBTs. Recent SBT approaches [22, 24, 21] propose compression strategies through an improved representation of the informative bits at each level of the tree. Compression does not change the false-positive rates but can increase the query time. Matrix structures' query requires rows of the same size in the matrix, and therefore it is not adapted to compression schemes. The folding strategy used in COBS allows to resize Bloom filters according to the dataset size to limit memory usage, but this optimization comes at the price of a higher false-positive rate. SeqOthello compresses the presence/absence vectors by buckets, using different compression strategies (RRR, delta encoding) according to the sparsity of the vectors.

Disk/RAM requirements Both matrix and tree structures take $\mathcal{O}(n \times b)$ space. One can expect SBTs to be a little less efficient in space since they maintain $\sim 2n$ Bloom filters instead of n for matrix structures. However, their compression scheme can make them less costly than sequence Bloom matrices in practice at a fixed false-positive rate. SeqOthello requires a $\mathcal{O}(Nlog(n))$ bits of space, with N the number of distinct k-mers. All three structures keep all preprocessed data (datasets BFs or k-mer sets) on the disk as temporary files during the indexes construction, which leads to a heavy temporary disk footprint. During the query, each approach is loaded partially in

AMQ method	clustering	compression	overhead	update	query
COBS	no	no (folding)	$\mathcal{O}(1)$	yes* $\mathcal{O}(b)$	$\mathcal{O}(n)$
HowDeSBT	yes	yes	$\mathcal{O}(n)$	yes* $\mathcal{O}(height \times b)$	$\mathcal{O}(n)$
SeqOthello	NA	yes	$\mathcal{O}(n)$	yes $\mathcal{O}(n)$	$\mathcal{O}(n)$
PAC	no	yes	$\mathcal{O}(log(n))$	yes $\mathcal{O}(b)$	$\mathcal{O}(n)$

Table 1: Comparison between the most recent AMQ approach and ours. We give worst-case complexities. The "clustering" columns denote whether the methods necessitate a prior clustering of the Bloom filters."compression" indicates if the method compresses the Bloom filters. "overhead" indicates the time overhead for constructing the index after input preprocessing. "update" gives the time complexity for inserting a new dataset ($heigh$ is the tree height). "yes*" means that the operation is theoretically feasible but not yet implemented. NA stands for not applicable. "accesses for query" gives the expected order of magnitude of memory accesses for a single query in the general case.

RAM. Using a parameter, COBS and previous generations of SBTs can be fully loaded on request. It is not the case for HowDeSBT and SeqOthello.

2.2.3 Use cases

With their stacks of Bloom filters, sequence Bloom matrices are well adapted to datasets of similar sizes, such as collections of similar genomes. SBT and SeqOthello's compression factors will be particularly well suited to somewhat similar datasets, such as sequencing data from cohorts.

The recursive query that prunes the query space at each level also makes SBT structures suitable for querying many similar datasets. Similarly, we can expect a batch of very similar queries to be accelerated in SBTs as they will all descend in identical (cached) subtrees.
On massive query sequences and diverse query k-mer sets, sequence Bloom matrices or SeqOthello can be preferred because they ensure less random accesses in the worst case when k-mer look-ups become a bottleneck.

2.2.4 An introduction to PAC through conceptual comparison to other AMQs

PAC is based on BFs as sequence Bloom trees and matrices. It inherits from SBTs by keeping the idea of organizing data in tree structures and finding an efficient encoding for the internal bits of the tree. As SBTs, it compresses its leaves, but contrary to all other methods, it never has high disk footprints during the construction.

On the query side, PAC is also related to COBS because queries are handled using an inverted index on a matrix of BFs. During the query, its number of random accesses is constant as in COBS and SeqOthello's schemes. Table 2.2.3 summaries aspects of PAC and other methods' comparison. In the following, we expend on PAC's structure and novelties.

2.3 Partitioned Aggregative Bloom Comb Trees

In this section we present our Partitioned Aggregative Bloom Comb Trees. We kept PAC as an acronym that covers the three keywords, including main novelties compared to previous tree structures. PAC is available at `https://github.com/Malfoy/PAC`.

2.3.1 Aggregative Bloom filters

We work using a concept partly similar to *cascading Bloom filters* in [25, 26] in the sense that consecutive Bloom filters are used to store information. However the possibilities of BFs content is more restrained in our case.

Definition 1 ***Aggregative Bloom filter.*** *We define a series of Bloom filters* BF_1, BF_2, $\dots BF_t$ *built using common* (b, h) *parameters. They represents* k*-mer sets* $R_1, R_2, \dots, R_t$ *such that* $R_t \subseteq \dots R_2 \subseteq R_1$*. We encode this particular series by seeing it as a matrix* M *of size* $t \times b$*. Let* V *be a an integer vector of size* b*. For* $1 \leq i \leq t, V[i]$ *is the length of the run of 1 in the row* i *of* M*. We call such a series of Bloom filters encoded in* V *an aggregative Bloom filter.*

See the grey area of Figure 2 for an example of aggregative Bloom filter. V allows to encode efficiently 2^{size} aggregated Bloom filters using $b \times size$ bits. This way we encode of t Bloom filter of size n using $n \times log(t)$ bits.

Observation 1 *One can notice that any branch of a SBT is a series of aggregated Bloom filters, with* BF_t *being the leaf node in the branch, and* BF_1 *the root node.*

2.3.2 Aggregative Bloom Comb Trees

PAC relies on comb-trees to structure the set of Bloom filters representing each dataset. It builds two comb-trees that share the same set of leaves. As in SBTs, these trees' inner nodes are unions of Bloom filters, but for PAC, they are organized in a binary left-comb tree and another binary Bloom right-comb tree (see Figure 2 for an example of the difference between a tree used by SBTs and a single binary Bloom comb-tree). Similarly to sequence "Bloom trees" we name these trees "Bloom comb-trees".

Given a list of BFs $BF_1, BF_2, \dots BF_n$ representing datasets, The binary Bloom left-comb tree is such that the leftmost inner node BF_{n+1} represents $BF_1 \cup BF_2$. Each other inner node up to the root contains the union BF of its child inner node and of a leaf BF, then we have $BF_{n+1} \subseteq BF_{n+2} \subseteq \dots BF_{n+height}$ (with *height* being the comb height). See Figure 2 for an example of a Bloom left-comb tree. The binary right comb-tree can be defined by symmetry.

Observation 2 *Similarly to the previous observation, one can notice that the longest branch of the comb is a series of aggregated Bloom filters.*

Following Observation 2, we propose to use an aggregative Bloom filter to encode the leftmost branch of the left Bloom comb-tree (respectively, the rightmost branch of the right comb Bloom tree).

Definition 2 ***Aggregative Bloom Comb Tree.*** *The Aggregative Bloom Comb Tree structure is composed of a pair of Bloom left and right comb trees. They are represented using two components. First, two aggregative Bloom filters* V_l, V_r *(see Definition 1) of size* b*, representing the branch going through all internal nodes down to the deepest leaf in both left and right-comb tree. Second,* n *Bloom filters are the leaves of both combs.*

Panel A of Figure 3 shows an example of two Bloom comb trees and the aggregative Bloom filters that represent them. An aggregative Bloom filter represents a run of 1's in an inclusion series of BFs.

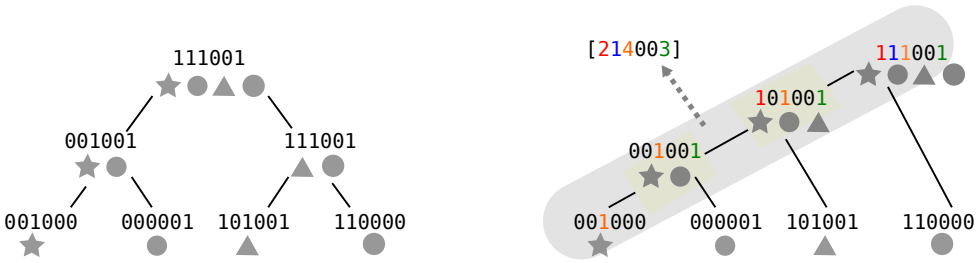

Figure 2: Left: a SBT structure. Right: a Bloom left comb-tree for the same database. Four datasets of a database are represented using grey shapes. Toy example Bloom filters are represented as bit vectors associated with shapes, and the content of each node is recalled. In the case of the Bloom comb tree, the inner nodes' Bloom filters (highlighted zone in grey) are not explicitly represented but are encoded using an aggregative Bloom filter instead (integer vector). Runs of 1's corresponding to the integers are colored. For instance, the leftmost 1 is found at levels 1 and 2, therefore a run of length 2 in the vector.

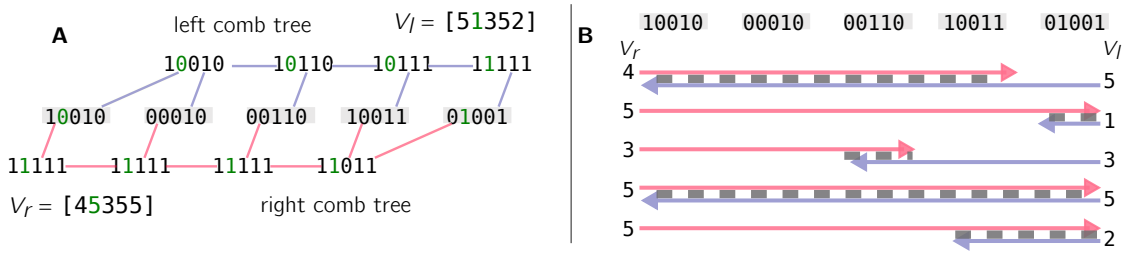

Figure 3: **Panel A:** Two Bloom comb trees and their aggregative Bloom filters. The leaves BFs are in the middle, colored in grey. They are the BFs representing the input datasets. The left-comb tree (top) and right-comb tree (bottom) are built by aggregating the BF's k-mers. The aggregative Bloom filter V_l represents the leftmost path of the top comb. Respectively, V_r represents the rightmost path of the bottom comb. We colored in green the second bit of leaves in order to show its encoding in the aggregative BFs. In the left comb, this bit is set to 1 only in the root for the longest branch; therefore, the value is 1 (green) in V_l. Conversely, the longest branch of the right tree has a run of 5 1's on the second position, hence a 5 in V_r. **Panel B:** using V_l and V_r from panel A, we show how aggregative Bloom filters define restrained search spaces in the leaves. Again, the leaves' BFs are grey, and the values of the vectors are printed vertically. Arrows represent the intervals given by each V, and the final search space at each position is the dotted grey area).

Thus, V_l (and symmetrically, V_r) defines the maximal depth at which 1's can be encountered at a given position in Bloom filters of the combs, i.e., the search space for a hit. Therefore, intersecting the two $V's$ intervals refines bounds for the search space at each position (see Panel B of Figure 3 for an example).

2.3.3 Partitioning

Several methods exploited the fact that overlapping k-mers have a high probability of sharing the same minimizer for partitioning [18]. A PAC is also a partitioned structure based on minimizers. In practice, we build a distinct *aggregative Bloom comb tree* for each distinct possible minimizer

value (4^m for m-minimizers). Each *aggregative Bloom comb tree* indexes all k-mers sharing a given minimizer. Panel A of Figure 4 shows which information is stored in a partition to represent a PAC.

The interest of this partitioning is two-fold. By working on smaller substructures, both construction and query can benefit from improved cache coherence as several successive k-mers will be located in the same small structure, generating cache miss for groups of k-mers instead of doing so for nearly each k-mer.

The combined m *aggregative Bloom comb trees* stored in m partitions represent the complete *partitioned aggregative Bloom comb tree* (PAC) structure.

2.3.4 Query

Inverting the index and querying single k-mers. For fast query operations, we build an inverted PAC index. Per partition, we associate to each possible hash value h a bit vector of size n that represents which leaves have a one at position h. Such bit-vectors are dubbed "bit-slices" by COBS and BIGSI papers. Before querying sequences, we build the inverted index by reading each Bloom filter successively. Then for a single query, its hash value h is computed, and the bit-slice h is browsed to find which BFs include this k-mer.

Query of a sequence. Partitioning by minimizers allows querying blocs of consecutive k-mers at once using super-k-mers. Thus, a query sequence is first split in super-k-mers, buffered according to their partition (supplemental Figure S1 shows how super-k-mers are related to partitions). Then for each partition, if the super-k-mer buffer is not empty, the partition and its corresponding V_l, V_r and bit-vectors are loaded, and k-mers are hashed. A k-mer whose hash value is i accesses the ith bit-vector and directly obtains the list of Bloom filters which are candidates to contain it through the slice given by V_l and V_r. An example is presented in Figure 4, Panel B (super-k-mers are omitted). Finally, to report the overall k-mer presence/absence over the whole query sequence, we maintain and return an accumulator vector of size n. It is incremented by one at a position j each time a k-mer is reported to appear in leaf j.

2.3.5 Update PAC by inserting new datasets

As our structure is accumulative, adding new datasets does not require changing the index structure. New datasets can be added by constructing their BFs and updating cells of the aggregative Bloom filters when needed. We implemented this functionality to allow the user to insert a dataset collection to an existing index. This insertion presents a very similar cost to building an index from the said collection. However it is way less costly than rebuilding from scratch an index including previous and novel datasets.

2.3.6 Implementation details

K-mer representation Each k-mer is represented by the lexicographically smallest of the forward string and reverse-complemented string (canonical k-mer) and is hashed using a fast xorshift hashing function.

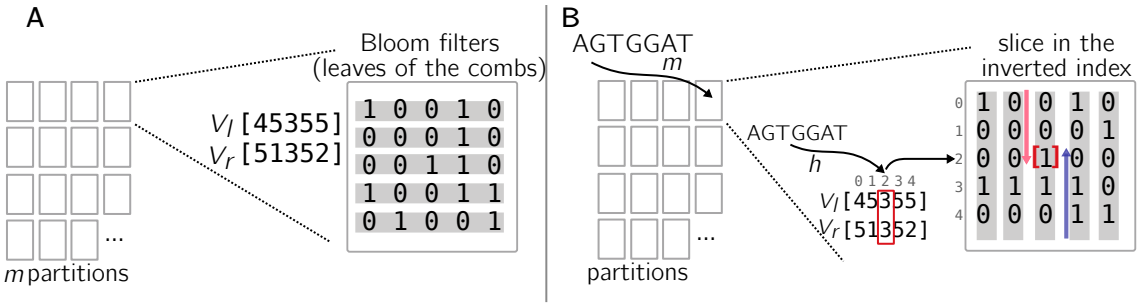

Figure 4: **Panel A:** the content of one partition in PAC. The aggregative Bloom comb trees constructed for a given partition are represented using leaves BFs (in grey) and two aggregative BFs V_l and V_r. **Panel B:** a single query in the same partition as Panel A. The index has been inverted. A k-mer enters a PAC by finding the partition corresponding to its minimizer. V_l and V_r are loaded, and a search space (here [3] only) is defined given the position obtained by hashing the k-mer (here the k-mer's hash value is 2). Then, a slice is extracted from the inverted index at the given position, and bits are checked to find 1's. Here there's a single position, so necessarily an 1.

Memory usage and parallelization Bloom filters are computed in RAM and serialized on disk to avoid heavy memory usage. This way, only one Bloom filter is stored in RAM at a given moment (C if C threads are used to treat the datasets in parallel). A PAC index is distributed into P partitions serialized in P different files. Partitions can be handled separately, being mutually exclusive. This strategy provides inherent coarse-grained parallelism as different threads can operate on different partitions without mutual exclusion mechanism. It also grants a low memory usage as only small partitions (i.e., a fraction of the total index) have to be stored in memory at a given time.

The query also benefits from partitioning similarly. All k-mer associated with a given partition are queried at once. Each Bloom filter is loaded separately in RAM and deallocated after the bitvector is filled (C Bloom filters can be simultaneously in RAM if C threads are used). Partitions are also handled sequentially. This behavior guarantees that each partition will only be read from the disk and stored in RAM once. Furthermore, partitions not associated with any query k-mer can be skipped.

Inverted index Bloom filters are represented as sparse bit-vectors (using the efficient BitMagic library[1]) in order to optimize memory usage. To avoid performing several "costly" queries on the same filter several times, we built an inverted index before actually performing the queries. The bit-slices used by the inverted index are encoded using sparse bit-vectors; therefore, iteration on the 1's of a bit-slice is performed in $\mathcal{O}(o)$ where o is the number of bits set to 1, instead of $\mathcal{O}(n)$ in other Bloom matrices structures.

[1] `https://github.com/tlk00/BitMagic`

3 Results

3.1 Preamble

All experiments were performed on a single cluster node running with Intel(R) Xeon(R) CPU E5-2420 @1.90GHz with 192GB of RAM and Ubuntu 16.04.

In the following experiments, we show RNA-seq indexing use cases for PAC. We compared PAC against the latest methods of the AMQ paradigm, i.e., HowDeSBT [21] for SBTs and SeqOthello [15]. HowDeSBT is the third and most recent generation of SBTs. We also tested COBS [23], the most recent sequence Bloom matrix, although it is not designed to work on sequencing data but rather on genomes.

We used kmtricks [27] for an optimized construction of HowDeSBT and its Bloom filters (commit number `532d545`). SeqOthello (commit 68d47e0) uses Jellyfish [28] for its pre-processing, we worked with version 2.3.0. COBS's version was v0.1.2, and PAC's commit was a2a0112.

3.2 Indexing 2,500 human datasets and comparison to other AMQ structures

The dataset first used in the initial SBT contribution [19] has become a *de facto* benchmark in almost any following paper describing a related method. We make no exception and benchmarked PAC and other AMQ structures on this dataset. It contains 2,585 human RNA-seq datasets, with low frequency k-mers filtered according to previous works [22][2], resulting in a total of ~ 3.8 billions of distinct k-mers. The input files are represented as compacted de Bruijn graphs generated by Bcalm2 [29] in gzipped FASTA files. We also kept settings used in previous benchmarks and used a k value of 21. We chose COBS, PAC, and HowDeSBT's settings to have an average 0.5% false positive rate. Notably, SeqOthello's false positive rate cannot be controlled.

In Table 2, we present the costs of indexing this database with the different tools. Several observations can be made. First, PAC does not generate any temporary disk footprint; only the final index is written on disk, while HowDeSBT, SeqOthello, and COBS generate many temporary files that are an order of magnitude larger than the index itself. This cost is not to neglect as it could become a hindrance in a disk-limited environment and should be seen as a scalability problem to index more significant instances. Another observation is that PAC and COBS's preprocessing and index construction steps are comparable and faster than the other tools, showing at least a two-fold improvement of required CPU time. The total computational cost of a PAC index is improved threefold over SeqOthello and six-fold over HowDeSBT. Memory-wise, the memory footprint of HowDeSBT and PAC are both low, while SeqOthello and COBS are somewhat higher without being prohibitively high. We want to highlight that PAC construction memory is roughly proportional to the number of cores used and could therefore be reduced using less cores at the cost of longer wall-clock time (see Table S1 in Appendix). Finally, excluding COBS, the different produced indexes are of the same order of magnitude even if PAC presents the heaviest index, slightly larger than SeqOthello, while HowDeSBT is the smallest. Being compression-free, COBS' index is two orders of magnitude larger than the other tools and produces even larger temporary files. COBS' CPU time is similar to PAC's, but its heavy disk usage presents a higher wall-clock time on our hard-disk drives.

[2] https://www.cs.cmu.edu/ ckingsf/software/bloomtree/srr-list.txt

Tool	Max temporary disk (GB)	Pre-processing time (CPU,h)	Index construction time (CPU,h)	Peak RAM (GB)	Final index Size (GB)
COBS	4,914	**7**	**1**	111	2,458
HowDeSBT	650	16	44	**4.2**	**15**
SeqOthello	1,000	19	12	43	25
PAC	**0**	8	**1**	9.6	28

Table 2: Index construction resource requirements on 2,585 RNA-seq. The "Max temporary disk" column reports the maximal external space taken by the method (usually during pre-processing). "Final index size" indicates the final index size when stored on disk. Execution times of methods are reported in CPU hours. All methods were run with 12 threads.

3.3 Indexing over 32,000 human datasets and scalability regarding the number of datasets

We downloaded 32,768 RNA-seq samples from SRA (accession list in available on the github repository[3]) for a total of 30 TB of uncompressed FASTQ data. We created incremental batches of 2^8, 2^9, ..., up to 2^{15} datasets to document methods' scalability according to the number of datasets. In this second experiment, the indexes are built directly on sequencing datasets in raw FASTA format, containing redundant k-mers and sequencing errors. Each tool was configured to filter unique k-mers before indexing to remove most sequencing errors (we used $k = 31$). Since COBS does not provide such an option and produces huge index files, we did not include it in this benchmark. To approximate the order of magnitude of the amount of (non-unique) distinct k-mers, we used ntcard [30]. For example, the batch of 2^{11} datasets contains approximately 3 billion k-mers to index, while the 2^{14} dataset contains more than 40 billion k-mers.

In Figure 5 (Panel A), we report the CPU time required for the indexing construction along with the end-to-end index production (including pre-processing) to display their evolution on databases of increasing size. On Panel B of Figure 5, we report the disk usage of the index along with the total disk usage required during the construction (including pre-processing).

We can make similar observations to the first experiments. PAC is the only tool that managed to build an index of the 32,000 human RNA-seq samples, in ∼5 days (∼60 CPU days), using 22GB of RAM and for a total index size of 1.1TB. PAC's construction is faster than state-of-the-art for the whole construction process. It produces slightly bigger indexes but requires much less external memory. SeqOthello crashed during index construction on the 2^{12} datasets experiment, and HowDeSBT failed to end before our ten-day timeout on the 2^{11} datasets experiment. PAC was the only tool to build an index on the 2^{13}, 2^{14}, and 2^{15} datasets.

3.4 Query results

As pointed out in [21], SBTs' query time has a relatively high variance and is impacted by batch effects. For a fair comparison, we designed a similar experiment to the one presented in HowDeSBT's paper, with sequences batches of increasing sizes. We repetitively selected random transcripts from human RefSeq (using seqkit [31]), and gathered 10, 10, 5, 3 and 3 batches of sizes 1, 10, 100, 1,000 and 10,000 transcripts. We report HowDeSBT, SeqOthello, and PAC's CPU time on each batch size in Figure 6. We warmed the cache prior to profile the queries. We observe that, according to our description in Section 2.2, sequence Bloom tree structures perform the best on small query sets.

[3] `https://github.com/Malfoy/PAC/blob/main/listpacpaper.txt.gz`

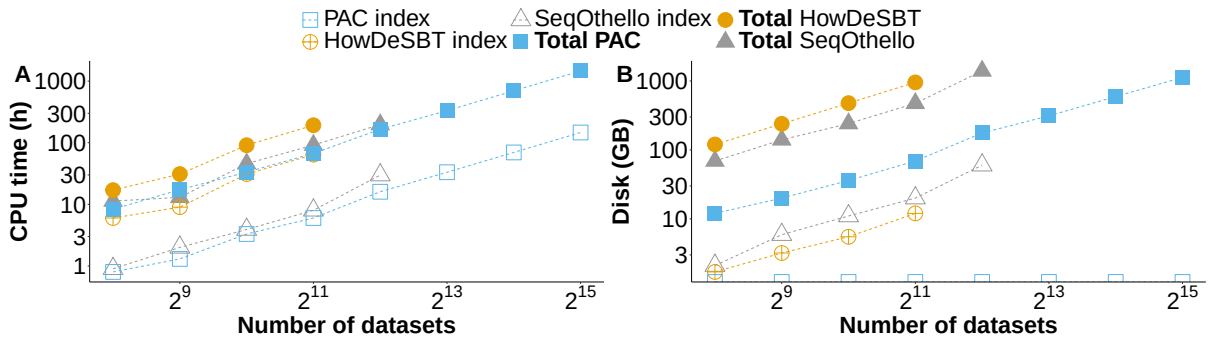

Figure 5: Results on increasing human dataset sizes. We report CPU hours for constructing the different indexes, including the pre-processing (methods preceded with "total") and the index alone. X-axis is in log2 scale and Y-axis is in log10 scale. All methods were run with 12 threads.

In larger instances, other methods can be preferred. SeqOthello shows the best performance and keeps a relatively constant query time over the input size. PAC has the same behavior while being slightly slower than SeqOthello. We also note that no method required more than 10GB of RAM to perform the queries. PAC had the lowest RAM footprint with up to 2GB for 10,000 transcripts.

COBS CPU time usages are not representative of its actual queries performances as it incurs almost no CPU times (typically less than 1 CPU second per transcript), so we chose not to plot it along with its competitors. We performed a separate benchmark to assess the queries wall-clock times. On a batch of 100 transcripts, HowDeSBT lasted 1h30, COBS lasted 30 minutes, SeqOthello lasted 10 minutes, and PAC lasted 5 minutes. Interestingly on a larger batch of 10,000 transcripts, SeqOthello and PAC presented very similar results (10 and 5 minutes, respectively). Those results denote that the query time of these two tools is dominated by loading the index. To assess when query time became predominant, we queried a large batch of 100,000 transcripts for which PAC lasted 11 minutes (2 CPU hours) and an even larger batch of 500,000 transcripts handled by PAC in 40 minutes (6.5 CPU hours).

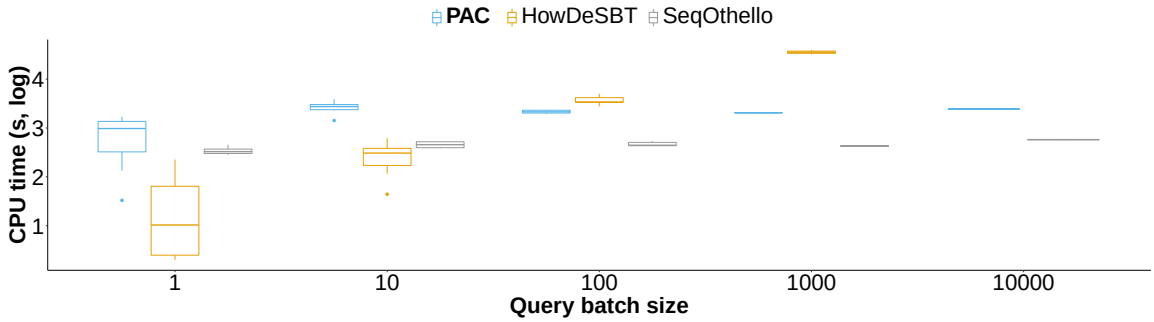

Figure 6: Results on query batches. We present query CPU times computed on batches of 1 to 10k transcript sequences. The Y-axis is on a log10 scale. The queries were computed using 12 threads.

4 Discussion

4.1 Comparison to other AMQ structures

Time and memory for index construction. PAC reached an unprecedented index construction for over 30TB of 32,000 human RNA-seq samples, in 5 days (60 CPU days), using 22GB of RAM and for a total index size of 1.1TB. In previous papers, authors tended to show the files pre-processing (BFs or k-mer sets computation) as a side task, assuming that it can be done on streaming while downloading the reads. The time/memory requirements for these tasks are presented apart from the main benchmarks. We think that this ideal situation is not always met in practice. This observation motivated PAC's behavior where index construction is not separated from input pre-processing. PAC allows an optimized streaming construction as a single BF is written at a time on the disk. It stands among the fastest methods for index construction while maintaining low disk and memory footprints.

Query We showed PAC's ability to query 500,000 human transcripts in less than an hour, being the fastest in wallclock time, and comparable to SeqOthello in CPU time. The worst-case query complexity remains the same for all methods, $\mathcal{O}(n)$ for a k-mer present in all n datasets. We saw that inverted indexes methods perform $\Theta(1)$ random accesses but still need to read a bit-slices of size $\mathcal{O}(n)$. Using aggregated Bloom filters, PAC improves these indexes, as in favorable cases (k-mers present in a single dataset with no collision), PAC answers in $\mathcal{O}(1)$ since only one position can and will contain a 1. Moreover, since we implement our bit-slices as sparse vectors, enumerating the 1's is done in $\Theta(o)$ (o being the number of 1's) instead of $\mathcal{O}(n)$. Moreover, in current version, the inverted is computed at query time, a way to achieve even faster query time would to to invert the index a construction time.

4.2 Future work

AMQ's trade-off for efficiency is to accept false positives during the query. Recently, Findere [32] provided a solution for reducing the false-positive rate of any AMQ data structure indexing k-mers while speeding up the queries. We plan on integrating Findere to later versions of PAC and thus, reaching a negligible amount of false positives in the structure.

While much effort has been put into designing efficient indexes, the query expressivity has remained practically the same since the first SBT paper. Two data-structures [33, 34] improved on that question by allowing to query k-mers abundances instead of presence/absences in collections. These methods are based on exact k-mer representations and show scalability up to 2,500 human RNA-seq. As a follow-up for PAC, we would like to augment the index with probabilistic structures such as count-min-sketches for efficient abundance storage per dataset.

Currently, two visions complementing each other emerge for dealing with large sequence collections. Some works capitalize on tremendous computing resources to pre-build giant indexes that are then make them available to the community through API on webservers [35, 36]. Conversely, our goal with PAC is to make accessible index construction for a large range of projects directly from the lab, even in moderate computational conditions as SRA aggregated over 200,000 human RNA-seq[4], our next goal is to reach this order of magnitude.

[4]Illumina human RNA-seq over 30M reads, accessed through ENA's API as of February 2022.

5 Acknowledgements

The authors would like to thank Léonid for a renewed sense of what tiredness, teamwork and joy are. We also thanl Daniel Gautheret for providing us with tips on the SRA API, Bastien Cazaux for proofreading the manuscript, and Anatoliy Kuznetsov for his outstanding Bitmagic library and sustained support. This work was supported by a grant by the Agence Nationale de la recherche for the project "Transipedia " [ANR-18-CE45-0020].

References

[1] European nucleotide archive. ena statistics – reads growth - reads doubling time. `https://www.ebi.ac.uk/ena/about/statistics`. Accessed: 2022-02-01.

[2] Li, H. Aligning sequence reads, clone sequences and assembly contigs with bwa-mem. *arXiv preprint arXiv:1303.3997* (2013).

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

[4] Janin, L., Schulz-Trieglaff, O. & Cox, A. J. BEETL-fastq: a searchable compressed archive for DNA reads. *Bioinformatics* **30**, 2796–2801 (2014).

[5] Dolle, D. D. *et al.* Using reference-free compressed data structures to analyze sequencing reads from thousands of human genomes. *Genome research* **27**, 300–309 (2017).

[6] Camacho, C. *et al.* Blast+: architecture and applications. *BMC bioinformatics* **10**, 1–9 (2009).

[7] Marchet, C. *et al.* Data structures based on k-mers for querying large collections of sequencing data sets. *Genome Research* **31**, 1–12 (2021).

[8] Marchet, C., Kerbiriou, M. & Limasset, A. Indexing de bruijn graphs with minimizers. *bioRxiv* (2019). URL `https://www.biorxiv.org/content/early/2019/02/13/546309`.

[9] Almodaresi, F., Sarkar, H., Srivastava, A. & Patro, R. A space and time-efficient index for the compacted colored de Bruijn graph. *Bioinformatics* **34**, i169–i177 (2018).

[10] Muggli, M. D., Alipanahi, B. & Boucher, C. Building large updatable colored de Bruijn graphs via merging. *Bioinformatics* **35**, i51–i60 (2019). URL `https://doi.org/10.1093/bioinformatics/btz350`. `http://oup.prod.sis.lan/bioinformatics/article-pdf/35/14/i51/28913259/btz350.pdf`.

[11] Holley, G. & Melsted, P. Bifrost–Highly parallel construction and indexing of colored and compacted de Bruijn graphs. *BioRxiv* 695338 (2019).

[12] Pandey, P. *et al.* Mantis: a fast, small, and exact large-scale sequence-search index. *Cell systems* **7**, 201–207 (2018).

[13] Mustafa, H. *et al.* Dynamic compression schemes for graph coloring. *Bioinformatics* **35**, 407–414 (2018).

[14] Karasikov, M. *et al.* Sparse binary relation representations for genome graph annotation. In *International Conference on Research in Computational Molecular Biology*, 120–135 (Springer, 2019).

[15] Yu, Y. *et al.* Seqothello: querying rna-seq experiments at scale. *Genome Biology* **19**, 167 (2018).

[16] Almeida, A. *et al.* A unified catalog of 204,938 reference genomes from the human gut microbiome. *Nature biotechnology* **39**, 105–114 (2021).

[17] Roberts, M., Hayes, W., Hunt, B. R., Mount, S. M. & Yorke, J. A. Reducing storage requirements for biological sequence comparison. *Bioinformatics* **20**, 3363–3369 (2004).

[18] Deorowicz, S., Debudaj-Grabysz, A. & Grabowski, S. Disk-based k-mer counting on a pc. *BMC bioinformatics* **14**, 1–12 (2013).

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

[20] Bradley, P., den Bakker, H. C., Rocha, E. P., McVean, G. & Iqbal, Z. Ultrafast search of all deposited bacterial and viral genomic data. *Nature biotechnology* **37**, 152 (2019).

[21] Harris, R. S. & Medvedev, P. Improved representation of sequence bloom trees. *Bioinformatics* **36**, 721–727 (2020).

[22] Sun, C., Harris, R. S., Chikhi, R. & Medvedev, P. Allsome sequence bloom trees. In *Research in Computational Molecular Biology - 21st Annual International Conference, RECOMB 2017, Hong Kong, China, May 3-7, 2017, Proceedings*, vol. 10229 of *Lecture Notes in Computer Science*, 272–286 (2017).

[23] *COBS: a Compact Bit-Sliced Signature Index*.

[24] Solomon, B. & Kingsford, C. Improved search of large transcriptomic sequencing databases using split sequence bloom trees. *Journal of Computational Biology* **25**, 755–765 (2018).

[25] Rozov, R., Shamir, R. & Halperin, E. Fast lossless compression via cascading bloom filters. In *BMC bioinformatics*, vol. 15, 1–8 (BioMed Central, 2014).

[26] Salikhov, K., Sacomoto, G. & Kucherov, G. Using cascading bloom filters to improve the memory usage for de brujin graphs. *Algorithms for Molecular Biology* **9**, 1–10 (2014).

[27] Lemane, T., Medvedev, P., Chikhi, R. & Peterlongo, P. kmtricks: Efficient construction of bloom filters for large sequencing data collections (2021). Preprint on webpage at `https://www.biorxiv.org/content/10.1101/2021.02.16.429304v1`.

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

[29] Chikhi, R., Limasset, A. & Medvedev, P. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics* **32**, i201–i208 (2016). URL `https://doi.org/10.1093/bioinformatics/btw279`. `http://oup.prod.sis.lan/bioinformatics/article-pdf/32/12/i201/17130531/btw279.pdf`.

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

[31] Shen, W., Le, S., Li, Y. & Hu, F. Seqkit: a cross-platform and ultrafast toolkit for fasta/q file manipulation. *PloS one* **11**, e0163962 (2016).

[32] Robidou, L. & Peterlongo, P. findere: fast and precise approximate membership query. In *International Symposium on String Processing and Information Retrieval*, 151–163 (Springer, 2021).

[33] Marchet, C., Iqbal, Z., Gautheret, D., Salson, M. & Chikhi, R. REINDEER: efficient indexing of k-mer presence and abundance in sequencing datasets. In *ISMB* (2020).

[34] Karasikov, M., Mustafa, H., Rätsch, G. & Kahles, A. Lossless indexing with counting de bruijn graphs. *bioRxiv* (2021).

[35] Karasikov, M. *et al.* Metagraph: Indexing and analysing nucleotide archives at petabase-scale. *bioRxiv* (2020).

[36] Edgar, R. C. *et al.* Petabase-scale sequence alignment catalyses viral discovery. *Nature* 1–6 (2022).

Appendix

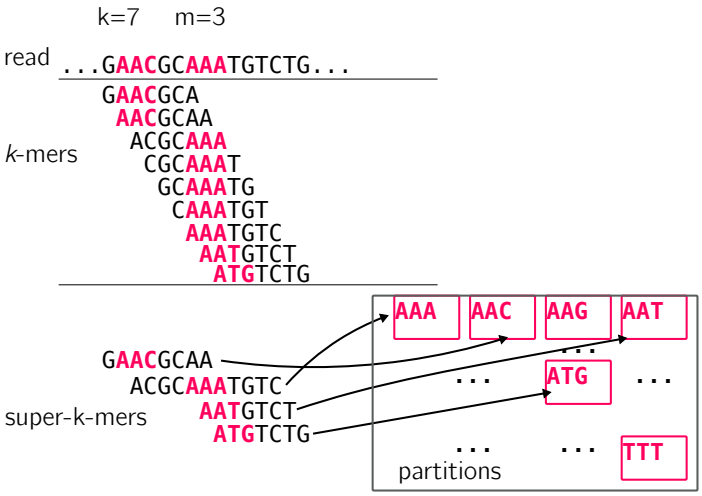

Figure S1: Example of minimizers and super-k-mers computed from a read ($k = 7, m = 3$). Minimizers are represent in pink (for the sake of simplicity, we consider the lexicographic order on minimizers). See how the second super-k-mer aggregates several k-mers having AAA as a minimizer. We show partitions corresponding to minimizers in the same color.

Threads	Bloom filters time	Index time	Total time	Peak RAM
1	4.2 h	30 min	4.7 h	0.73
2	2.2 h	17 min	2.5 h	1
4	1.27 h	8 min	1.4 h	1.57
8	44 min	4 min	48 min	2.78
12	40 min	4 min	44 min	3.9
24	34 min	4 min	38 min	7.2
4	1.27 h	7 min	1.38 h	2.4
8	43 min	4 min	47 min	3.5
12	32 min	3 min	35 min	4.4
24	28 min	2 min	30 min	7.4

Table S1: Multi-threading benchmark on 2,585 RNA-seq. We report the wall-clock times according to the number of threads used on our 12 cores machines.