Scalable sequence database search using partitioned aggregated Bloom comb trees

Camille Marchet, Antoine Limasset

Bioinformatics, 2023, **39**, i252–i259
https://doi.org/10.1093/bioinformatics/btad225
ISMB/ECCB 2023

OXFORD

Scalable sequence database search using partitioned aggregated Bloom comb trees

Camille Marchet[1,*] and Antoine Limasset[1,*]

[1]University of Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

*Corresponding authors. UMR CRIStAL Université de Lille - Campus scientifique, Bâtiment ESPRIT Avenue Henri Poincaré, 59655 Villeneuve d'Ascq, France. E-mails: antoine.limasset@univ-lille.fr (A.L.) and camille.marchet@univ-lille.fr (C.M.)

Abstract

Motivation. The Sequence Read Archive public database has reached 45 petabytes of raw sequences and doubles its nucleotide content every 2 years. Although BLAST-like methods can routinely search for a sequence in a small collection of genomes, making searchable immense public resources accessible is beyond the reach of alignment-based strategies. In recent years, abundant literature tackled the task of finding a sequence in extensive sequence collections using *k*-mer-based strategies. At present, the most scalable methods are approximate membership query data structures that combine the ability to query small signatures or variants while being scalable to collections up to 10 000 eukaryotic samples. **Results.** Here, we present PAC, a novel approximate membership query data structure for querying collections of sequence datasets. PAC index construction works in a streaming fashion without any disk footprint besides the index itself. It shows a 3–6 fold improvement in construction time compared to other compressed methods for comparable index size. A PAC query can need single random access and be performed in constant time in favorable instances. Using limited computation resources, we built PAC for very large collections. They include 32 000 human RNA-seq samples in 5 days, the entire GenBank bacterial genome collection in a single day for an index size of 3.5 TB. The latter is, to our knowledge, the largest sequence collection ever indexed using an approximate membership query structure. We also showed that PAC's ability to query 500 000 transcript sequences in less than an hour.
Availability and implementation: PAC's open-source software is available at https://github.com/Malfoy/PAC.

1 Introduction

Public databases, such as the Sequence Read Archive (SRA) or 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 resource for daily sequence analysis. But concretely, we only witness the onset of their exploitation, while their exponential growth poses serious scalability challenges [for instance, SRA has reached 45 petabytes of raw reads and roughly doubles every 3 years (ena)].

Today, searching for a query sequence in a single genome/dataset or a restricted collection of genomes is considered routine through alignment-based tools, such as BLAST (Altschul *et al.* 1990) and others (Camacho *et al.* 2009, Li and Durbin 2009, Janin *et al.* 2014, Dolle *et al.* 2017). In contrast, the scale of databases, such as SRA makes BLAST and any alignment-based method ill-suited due to the prohibitive cost of the alignment phase. Broader usages of such resources necessitate algorithms allowing hyper-scalable membership queries as a foundation. In particular, they must be able to quickly discard a sequence that is absent from a substantial dataset collection. They must also identify the datasets where the query is present. Therefore, recent literature has tackled these problems using alignment-free *k* -mer-based strategies.

The known most scalable solutions are sketching methods, which reduce the datasets to a small set of signatures 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. Numerous applications rely on significantly smaller query size, e.g. variant calling of SNP/small indels, finding alternative splicing sites, or other small genomic signatures.

Recent alignment-free literature has described novel methodologies to fill a dual need: small queries in vast dataset collections. Mostly, two paradigms (Marchet *et al.* 2021a) cover this question, both considering each dataset and the query itself as *k*-mer sets. The first type of approach relies on exact *k*-mer set representations. These methods build an associative index [using hash tables (Almodaresi *et al.* 2018, Holley and Melsted 2020, Marchet *et al.* 2021b, Pibiri 2022) or FM-indexes (Chikhi *et al.* 2015, Muggli *et al.* 2017, Belazzougui *et al.* 2018)] where the key set corresponds to all *k*-mers of the collection. Such static structure hardly scale to extensive instances with large *k*-mer cardinality but can be used for queries with high precision or to build de Bruijn graphs. Some indexing implement some kind of dynamicity either via merging or using dynamic bit vectors (Holley *et al.* 2016, Muggli *et al.* 2019, Alipanahi *et al.* 2021).

The second family of methods relies on probabilistic set representations (Solomon and Kingsford 2016, Bingmann *et al.* 2019, Bradley *et al.* 2019, Harris and Medvedev 2020). These approximate membership query (AMQ) structures trade false positives during the query for improved speed and memory performance compared to the previous category.

AMQ approaches allowed the indexing of hundreds of thousands of microbial datasets (Blackwell *et al.* 2021). For mammalian datasets, scaling to more samples is a timely challenge since the increase in the content of *k*-mer/datasets 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 (Yu *et al.* 2018) of indexing over 10 000 mammalian RNA samples.

This manuscript proposes a novel AMQ method to query biological sequences in collections of datasets dubbed Partitioned Aggregative Bloom Comb Trees (PAC). In our application case, queries are typically alternative splicing events or a short genomic context around a small variant or mutation, for both eukaryotic and microbial species. Our structure is designed to be highly scalable with the number of indexed samples and requires moderate resources.

2 Materials and methods

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

2.1 Prerequisites

A *set of sequences* R (also called "dataset") is a set of finite strings on the alphabet $\Sigma = \{A, C, G, T\}$. In practice, sets of sequences can be read sets or genome sets. An input "database" $D = \{R_1, \dots R_n\}$ is a set of "datasets." Similarly to the datasets, a "query sequence" is a finite string from which all distinct k-mers are extracted (typically, a gene, a transcript, or some genomic context around a base mutation).

In the following, we suppose that broadly used concepts in the context of computational genomics are known [namely, k-mers, Bloom filters (Bloom 1970), and minimizers (Roberts *et al.* 2004, Chikhi *et al.* 2015)]. However, their definitions are recalled in the online supplementary appendix if needed.

Problem statement: The structures described hereafter estimate the cardinality of the intersection of 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 its intersection is larger or equal to τ.

A sequence Bloom tree (SBT) is an AMQ structure introduced in Solomon and Kingsford (2016). SBTs use a set of n Bloom filters, each representing the distinct k-mers of the n datasets in an input database. A SBT is a balanced binary tree that represents separately each Bloom filter in its leaves and the union of all Bloom filters in its root. Therefore, it allows fast membership queries in the whole database using recursive queries along the tree. We call a "sequence Bloom matrix" the matrix of Bloom filters introduced in BIGSI (Bradley *et al.* 2019). For SBTs, a Bloom filter is built for each of the n datasets in an input database. Then, these filters are stacked to become a matrix in which each Bloom filter is a column. By accessing a row, one can directly know whether an element is present or absent in all Bloom filters. A *SeqOthello* is an AMQ structure that relies on a different paradigm. It operates as a hash table where pairs of (k-mer, presence/absence bitvector) are associated using a static hashing strategy and can efficiently be interrogated for k-mer's presence in bit vectors. In the following section, we will define the necessary concepts to detail our novel structure.

2.2 PAC construction

2.2.1 Aggregated Bloom filters

PAC, as most AMQ structures, relies on Bloom filters to represent k-mers presence. For fixed parameters (k, b, h) (respectively, k-mer size, Bloom filter size, and number of hash functions), 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. As SBT, we organize our Bloom filters in a tree whose inner nodes result from a merge operation on Bloom filters:

Definition 1 (**Bloom filter merge**) Let two Bloom filters BF_1 and BF_2 with the same set of parameters (b, h). We define the *bf_merge* operation as a bitwise OR:

$$bf\_merge(BF_1, BF_2) = BF_1 | BF_2.$$

Definition 2 (**union Bloom filter**) A union Bloom filter is the Bloom filter result of a *bf_merge* operation, with conserved parameters (b, h).

The tree is then built by "merging" filters from bottom to top. The tree root represents the union Bloom filters of all distinct k-mers in D. PAC involves a novel tree topology in comparison to other SBTs that are balanced binary trees, which will be detailed in the following.

Definition 3 (**Aggregated Bloom filter**) We define a series of Bloom filters BF_1, BF_2, $\dots BF_t$ built using common (b, h) parameters. They represent 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 of t filters of size b as a matrix M of size $t \times b$. Let V be an integer array 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. V can effectively encode such a series of 2^S Bloom filters of size b using b integers of size S. In this way, we can encode t of such Bloom filters of size b using $b \times \log t$ bits. We call such an integer array V encoding for the series of Bloom filters BF_1, BF_2, $\dots BF_t$ an "Aggregated Bloom filter."

Observation 1 One can notice that any branch of a SBT could be represented by an Aggregated Bloom filters, with BF_t being the leaf node in the branch, and BF_1 the root node.

2.2.2 Aggregated Bloom comb trees

Definition 4 (**Comb tree**) We call a comb tree a binary tree whose each internal node has at least one leaf as a child. When considering an order on the leaves, a "left-comb tree" (respectively, "a right-comb tree") has its root connected to the rightmost leaf (respectively, the leftmost leaf).

PAC relies on Bloom comb trees to organize the set of Bloom filters representing each dataset.

Definition 5 (**Bloom comb tree**) We call a Bloom comb tree a comb tree built using the *bf_merge* operation. First, given a list of Bloom filters $BF_1, BF_2, \dots BF_n$ representing datasets, the leaves of the comb tree are built. Each leaf contains a Bloom filter as in SBTs. A Bloom left-comb tree is such that the leftmost inner

node $BF_{n+1} = bf\_merge(BF_1, BF_2)$. The right-comb tree can be defined by symmetry.

Property 1 In a Bloom left-comb tree built from a list of Bloom filters $BF_1, BF_2, \ldots BF_n$, we have $BF_{n+1} \subseteq BF_{n+2} \subseteq \ldots BF_{n+height}$ (with "height" being the height of the comb) due to the union operation performed in *bf_merge*.

See Fig. 1 for an example of the difference between a tree used by SBTs and a Bloom comb tree.

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 using an Aggregated Bloom filter to encode the longest branch of the Bloom comb tree.

Definition 6 (**Aggregated Bloom Comb Tree**) Unless otherwise specified, we denote by "Aggregated Bloom Comb Tree" a structure composed of a pair of Bloom left and right-comb trees built on the same list of leaves. They are represented using two components. First, two Aggregated Bloom filters V_l, V_r (see Definition 3) 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.

Figure 2A shows an example of two Bloom comb trees and the Aggregated Bloom filters that represent them. An Aggregated Bloom filter represents a run of 1's in an inclusion series of Bloom filters. 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 the bounds for the search space at each position (see Fig. 2B for an example).

2.2.3 Structure partitioning

In PAC, we build a distinct "Aggregated Bloom comb tree" for each minimizer associated with a non-empty k-mer set.

Definition 7 [**super-k-mer** (Deorowicz *et al.* 2013)] From an input string, a super-k-mer is a substring containing all consecutive k-mers that share a minimizer of size m.

As consecutive k-mers in a sequence largely overlap, blocks of k-mers tend to share their minimizer and yield super-k-mers. By associating super-k-mers to their minimizers (and therefore the k-mers they come from) one can divide a k-mer set into up to 4^m partitions (Deorowicz *et al.* 2013). See the Supplementary Fig. S1 in the Appendix for an example.

Figure 3A shows which information is stored in a partition to represent a PAC. The combined 4^m "Aggregated Bloom comb trees" stored in 4^m partitions represent the complete PAC structure.

2.2.4 Query

Definition 8 (**Single query**) We call a single query the query of a single k-mer to an AMQ data structure. The single query outputs a bit vector of size n indicating the presence/absence of a k-mer in each dataset.

Definition 9 (**Multiple query**) Let s be a string of size $|s| > k$. We call a multiple query the query of consecutive k-mers in s through single queries to an AMQ data structure. The results scores are reported in an integer vector of size n.

For all AMQ structures studied in this article, the worst-case time complexity for single queries is in $\mathcal{O}(n)$. Thus, the number of random access impacts is critical for runtimes in practice.

Definition 10 (**Inverted Bloom filter index**) In this framework, we call an inverted Bloom filter index an index that maps each possible hash value h to a bit vector of size n that represents which filters have one at position h.

To minimize the amount of random access, we rely on Inverted Bloom indexes during query time.

For each k-mer, its hash value h is computed, and the bit slice h is searched to find which Bloom filters include it. To further increase data locality, we construct one Inverted Bloom filter index per partition. In this way, querying successive k-mers from a super-k-mer is done on the same small

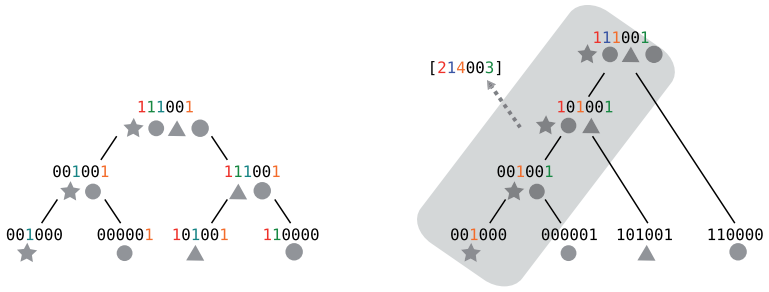

Figure 1. Left: A SBT structure. Right: A Bloom left-comb tree for the same database. Four datasets of a database are represented using grey shapes. Toy example Bloom filters are represented as bit vectors associated with shapes, and the content of each node is recalled. In the case of the Bloom comb tree, the inner nodes' Bloom filters (highlighted zone in grey) are not explicitly represented but are encoded using an Aggregated 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 two in the vector.

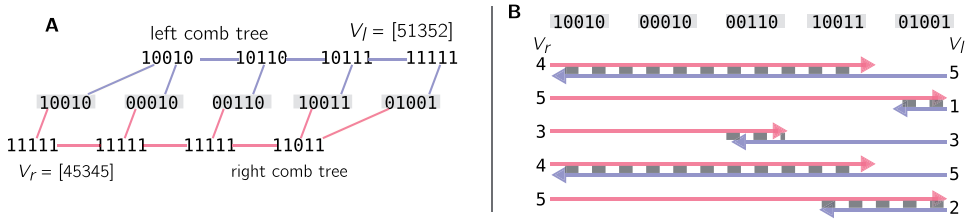

Figure 2. (A) Two Bloom comb trees and their Aggregated Bloom filters. The leaves Bloom filters are in the middle, colored in grey. They are the Bloom filters representing the input datasets. The left-comb tree (top) and right-comb tree (bottom) are built by aggregating the Bloom filter's *k*-mers. The Aggregated 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 Aggregated Bloom filters. 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 five 1's on the second position, hence, a 5 in V_r. (B) Using V_l and V_r from (A), we show how Aggregated Bloom filters define restrained search spaces in the leaves. Again, the leaves' Bloom filters 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.

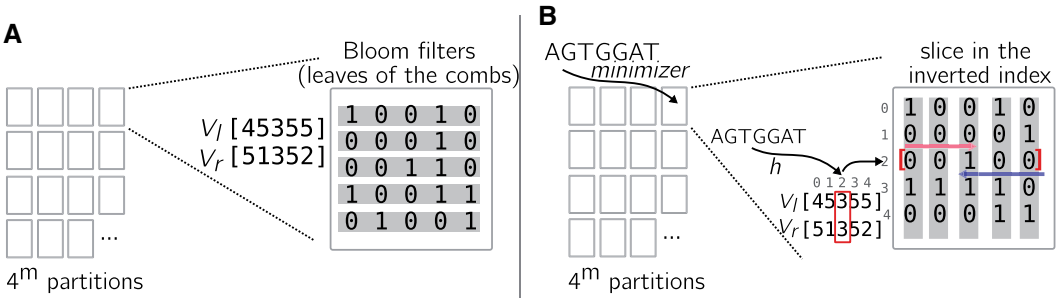

Figure 3. (A) The content of one partition in PAC. The Aggregated Bloom comb trees constructed for a given partition are represented using Bloom filter leaves (in gray) and two Aggregated Bloom filters V_l and V_r. (B) A single query in the same partition as (A). The index has been inverted. A *k*-mer 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 is a single position, so necessarily an 1.

data structure, reducing the amount of cache misses. Such partition indexes are only constructed if needed and kept to avoid redundant operations following a lazy strategy. An example is presented in Fig. 3B (super-*k*-mers are omitted). Finally, we maintain and return a score vector of size n. It is incremented by one at a position j each time a *k*-mer appears in leaf j. Therefore, PAC is also related to Sequence Bloom matrices because queries are handled using a matrix of Bloom filters.

2.2.5 Update PAC by inserting new datasets

Adding new datasets does not require changing the index structure. New datasets can be added by constructing their Bloom filters and updating the values of the Aggregated Bloom filters when needed. We implemented this functionality to allow the user to insert a dataset collection into an existing index. This insertion presents a very similar cost to building an index from the said collection, as both the construction and update algorithms consist of successive insertions. It avoids the need to rebuild the index from scratch due to novel datasets.

2.2.6 Implementation details

2.2.6.1 Memory usage and parallelization

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

Similarly, the query also benefits from partitioning. All *k*-mer associated with a given partition are queried at once in a sequential way to limit memory usage. Each Bloom filter is loaded separately in RAM and freed after the inverted index is constructed to be queried (C Bloom filters can be loaded simultaneously in RAM if C threads are used). This behavior guarantees that each partition will only be read from the disk once and that only one partition will be stored in RAM at a given time. Furthermore, partitions not associated with any query *k*-mer can be skipped. Both construction and query benefit from improved cache coherence as several successive *k*-mers are located in the same small structure, generating cache misses for groups of *k*-mers instead of doing so for nearly each *k*-mer.

2.2.6.2 Inverted index

Bloom filters are represented as sparse bit vectors (using the BitMagic library (https://github.com/Malfoy/PAC/blob/main/listpacpaper.txt.gz)] in order to optimize memory usage.

3 Results

All experiments were performed on a single cluster node running with Intel(R) Xeon(R) Gold 6130 CPU @ 2.10 GHz with 128 GB of RAM and Ubuntu 22.04.

3.1 Indexing 2500 human datasets and comparison to other AMQ structures

We compared PAC against the latest methods of the AMQ paradigm, i.e. HowDeSBT (Harris and Medvedev 2020) for SBTs and SeqOthello (Yu *et al.* 2018). We also tested COBS (Bingmann *et al.* 2019), the most recent sequence Bloom matrix, although it is not designed to work with sequencing data, but rather genomes. We used kmtricks (Lemane *et al.* 2022) for an optimized construction of HowDeSBT and its Bloom filters (commit number `532d545`). SeqOthello (commit 68d47e0) uses Jellyfish (Marçais and Kingsford 2011) for its preprocessing, we worked with version 2.3.0. COBS's version was v0.1.2, and PAC's commit was `cee1b5c`. We used the classic mode of COBS that does not rely on folding to ensure a fair comparison of Bloom filter size and practical false-positive rate across tools.

The dataset first used in the initial SBT contribution (Solomon and Kingsford 2016) has become a *de facto* benchmark in almost any subsequent article that describes a related method. We make no exception, and benchmarked PAC and other AMQ structures on this dataset. It contains 2585 human RNA-seq datasets, with low-frequency k-mers filtered according to previous works (Solomon and Kingsford 2018) (https://www.cs.cmu.edu/ ckingsf/software/bloomtree/srr-list.txt), resulting in a total of ∼3.8 billion of distinct k-mers. The input files are represented as compacted de Bruijn graphs generated by Bcalm2 (Chikhi *et al.* 2016) in gzipped FASTA files. We also kept the 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 1, 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. 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 2-fold improvement of required CPU time. The total computational cost of a PAC index is improved three times over SeqOthello and six times 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. 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. This can be explained by the fact that tree-based tools (as HowDeSBT) spend a lot of time to choose how to group and merge files to optimize the index compressibility resulting in a hard to construct but smaller indexes tradeoff. Computing an efficient file ordering to boost compression is an interesting problem that would benefit any matrix-based index. 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.

3.2 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 (https://github.com/tlk00/BitMagic)] 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 the scalability of methods 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 (Mohamadi *et al.* 2017). For example, the batch of 2^{11} datasets contains ∼3 billion k-mers to index, while the 2^{14} dataset contains more than 40 billion k-mers.

In Fig. 4, we report the CPU time required for index construction (including preprocessing) to display their evolution on databases of increasing size. In Fig. 5, we report temporary disk usage during index construction and the final index sizes on the disk after compression.

We can make observations similar to those in 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 22 GB of RAM and for a total index size of 1.1 TB. PAC's construction is faster than the 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 in the experiment with 2^{12} datasets, and HowDeSBT failed to end before our ten-day timeout in the experiment with 2^{11} datasets. PAC was the only tool to build an index on the 2^{13}, 2^{14}, and 2^{15} datasets.

3.3 Indexing microbial datasets

Besides indexing RNA-seq data, AMQ indexes were also used to index large collections of bacterial genomes. To further highlight the scalability of PAC, we built it on two massive

Table 1. Index construction resource requirements on 2585 RNA-seq[a].

Tool	Max temporary disk (GB)	Preprocessing time (CPU, h)	Index construction time (CPU, h)	Peak RAM (GB)	Final index size (GB)
COBS	4914	7	**1**	111	2458
HowDeSBT	650	16	44	**4.2**	**15**
SeqOthello	1000	19	12	43	25
PAC	**0**	8	**1**	9.6	28

[a] The column "Max temporary disk" reports the maximal external space taken by the method (usually during preprocessing). The "Final Index Size" indicates the final index size when stored on disk. Bold values indicate the best result in each column. The execution times of the methods are reported in CPU hours. All methods were run with 12 threads.

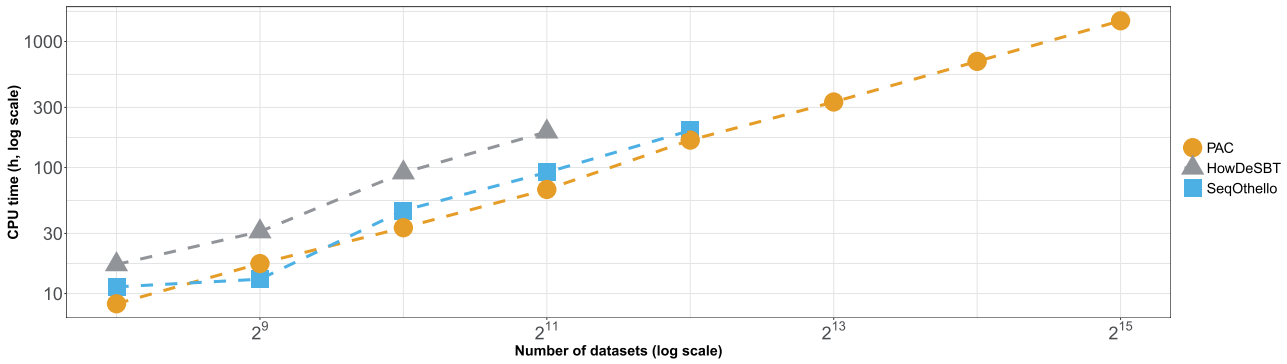

Figure 4. Results on increasing human dataset sizes. We report total CPU hours for constructing the different indexes, including the Bloom filter construction and index constructions. *X*-axis is in log 2 scale and *Y*-axis is in log 10 scale. All methods were run with 12 threads.

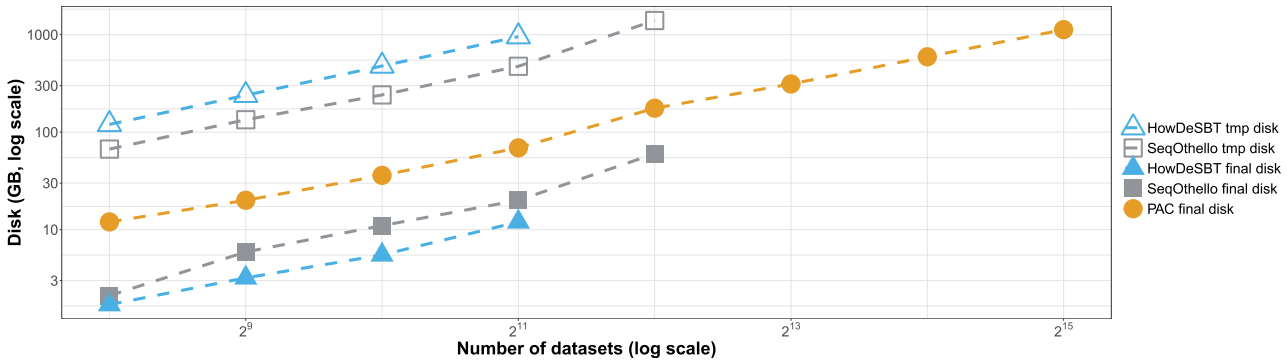

Figure 5. Results on increasing human dataset sizes. We report disk footprints, both temporary and final, for constructing the different indexes. Note that PAC does not use temporary disk. *X*-axis is in log 2 scale and *Y*-axis is in log 10 scale. All methods were run with 12 threads.

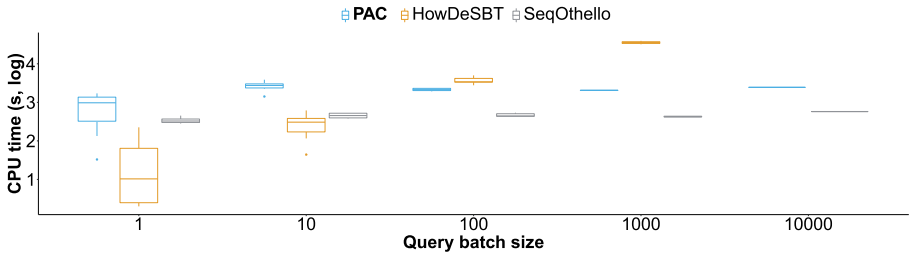

Figure 6. Results on query batches. We present query CPU times computed on batches of 1 to 10k transcript sequences. The *Y*-axis is on a log 10 scale. The queries were computed using 12 threads.

bacterial collections. In a first experiment, we fetched the 661 000 bacterial genomes representing more than three terabases, which were previously collected in a database (Blackwell *et al.* 2021) and built two indexes on this collection. A first index featuring large Bloom filters (2^{29} bits) for low *k*-mer query false-positive rates (below 1% for five megabases genomes) and a second index with smaller Bloom filters (2^{27} bits) when higher false-positive rates can be allowed (below 7% for five megabases genomes). Both indexes were constructed within 24 h using 21 and 6 GB of RAM, respectively, for a total size of 2.4 and 1.4 TB, respectively.

In a second experiment, we downloaded all bacterial assemblies available on GenBank (counting 1 200 575 genomes at the time of the experiment and representing more than five terabases) and built an index with a Bloom filter size of 2^{27} bits that corresponds to a false-positive rate around 4% for five megabases genomes. The construction lasted 24 h for a total of 410 CPU hours using less than 5 GB of RAM for a total index size of 3.5 TB. This is to our knowledge the largest collection ever indexed by an AMQ.

3.4 Query results on human RefSeq

We designed a similar experiment to the one presented in HowDeSBT's paper, with sequence batches of increasing sizes. We repetitively selected random transcripts from human RefSeq [using seqkit (Shen *et al.* 2016)], and gathered 10, 10, 5, 3, and 3 batches of sizes 1, 10, 100, 1000, and 10 000 transcripts. We report HowDeSBT, SeqOthello, and PAC's CPU time on each batch size in Fig. 6. We warmed the cache before profiling the queries. In accordance with the literature (Harris and Medvedev 2020), we observe that SBT structures perform the best on small query sets. In larger instances, other methods can be preferred. SeqOthello shows the best performance and keeps a relatively constant query time over the input size.

Table 2. Results on query times in CPU hours according to the PAC mode used along with the speedup obtained compared to the classical matrix approach (No ABF column)[a].

Query dataset	Double ABF filter	Single ABF filter	No ABF	Overhead
200 Random genomes	0.7 (13.6×)	1.3 (7.3×)	9.5	1.0
200 *E.coli* genomes	1.2 (7×)	2.2 (3.8×)	8.4	1.0
200 *S.enterica* genomes	10.4 (1.2×)	11.6 (1×)	12	1.0

[a] Double, Single, No ABF denotes the number of combs used in the structure. We present query CPU times computed along with the overhead constant across modes that include parsing the query file and loading the index. The random genomes are random nucleotide sequences of length five megabases. *E.coli* and *S.enterica* genomes were randomly selected among the RefSeq genomes.

PAC has the same behavior, while being slightly more CPU time than SeqOthello. We also note that no method required more than 10 GB of RAM to perform the queries. PAC had the lowest RAM footprint, with up to 2 GB for 10 000 transcripts. COBS CPU time usages are not representative of its actual queries' performance, as it incurs almost no CPU times (typically <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 1 h 30 min, COBS lasted 30 min, SeqOthello lasted 10 min, and PAC lasted 5 min. Interestingly, on a larger batch of 10 000 transcripts, SeqOthello and PAC presented very similar results (10 and 5 min, respectively). These results indicate that the query time of these two tools is dominated by index loading. To assess when query time became predominant, we queried a large batch of 100 000 transcripts for which PAC lasted 11 min (two CPU hours) and an even larger batch of 500 000 transcripts handled by PAC in 40 min (6.5 CPU hours).

3.5 Impact of comb structure on query for collections entailing high *k*-mer diversity

One novelty of PAC is the use of lightweight Aggregated Bloom filters to accelerate the queries by skipping subsections of the bit-slices. To assess the efficiency of this strategy, we report in Table 2 the query times of several query batches on an index composed of all complete *Salmonella enterica* genomes from RefSeq (counting 11 993 genomes at the time of the experiment). We report three different query times for each batch, using, respectively, two Aggregated Bloom Comb trees (according to Definition 6), a single one and none (i.e. the structure is a Bloom matrix). Without the use of Aggregated Bloom filters PAC index is conceptually identical to a matrix approach, such as COBS while using one or two Aggregated Bloom filters allow to skip some bitslice of the index. The number of bits that can be skipped depends in practice on the similarity shared between the query and the indexed documents. To highlight this effect, a batch is made up of *S.enterica* that are highly similar to the indexed genomes, a batch is made up of *Escherichia coli* that are very dissimilar to the indexed genomes, and a batch is made up of random sequence to show an extreme example of dissimilarity. The first observation is that Aggregated Bloom filters hardly improve the query time of the *S.enterica* batch. This result is expected because most query *k*-mers should be found in many indexed genomes and the amount of zeros that could be skipped in such slices should be very low. However, we see that using one Aggregated Bloom filter greatly accelerates the query on the two dissimilar batches, increasing the throughput by several fold. Furthermore, using a second Aggregated Bloom filter accelerates the query time even more but is not as beneficial as the first one.

4 Discussion

To our knowledge, PAC is the first AMQ *k*-mer set structure to index the entire GenBank bacterial genome collection and reach 32k human RNA-seq datasets.

PAC combines the simplicity and efficiency of inverted index matrix approaches, such as COBS or BIGSI with a lightweight tree structure.

The novel tree structure has a minimal resource footprint, yet greatly improves the query time when a query is dissimilar to the index content, a scenario possibly met with microbial databases. As *k*-mer sets are designed to efficiently skip unrelevant documents, our Aggregated Bloom filters allow us to efficiently prune our query space.

Using several real datasets, we demonstrate that PAC is practically scalable for its construction. In contrast to other approaches, PAC is simultaneously frugal in RAM, disk, and time requirements for building an index.

We showed PAC's ability to query 500 000 human transcripts in less than an hour, being the fastest in wall-clock 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 reviewed that inverted indexes methods perform $\Theta(1)$ random accesses, but still need to read 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 or absent everywhere), PAC can answer in $\mathcal{O}(1)$.

Acknowledgements

The authors would like to thank Léonid and Zora for a renewed sense of what tiredness, teamwork and joy are. They also thank Daniel Gautheret for providing us with tips on the SRA API, RECOMB-Seq reviewers, and Anatoliy Kuznetsov for his outstanding Bitmagic library and sustained support.

Supplementary data

Supplementary data is available at *Bioinformatics* online.

Conflict of interest

None declared.

Funding

This work was supported by grants from the Agence Nationale de la recherche for the project "full-RNA" [ANR-22-CE45-0007]; and "AGATE" [ANR-21-CE45-0012].

References

Alipanahi B, Kuhnle A, Puglisi SJ *et al.* Succinct dynamic de Bruijn graphs. *Bioinformatics* 2021;**37**:1946–52.

Almodaresi F, Sarkar H, Srivastava A *et al.* A space and time-efficient index for the compacted colored de Bruijn graph. *Bioinformatics* 2018;**34**:i169–77.

Altschul SF, Gish W, Miller W *et al.* Basic local alignment search tool. *J Mol Biol* 1990;**215**:403–10.

Belazzougui D, Gagie T, Mäkinen V *et al.* Bidirectional variable-order de Bruijn graphs. *Int J Found Comput Sci* 2018;**29**:1279–95.

Bingmann T, Bradley P, Gauger F *et al.* COBS: a compact bit-sliced signature index. In: *SPIRE*. 2019. doi: 1905.09624.

Blackwell GA, Hunt M, Malone KM *et al.* Exploring bacterial diversity via a curated and searchable snapshot of archived DNA sequences. *PLoS Biol* 2021;**19**:e3001421.

Bloom BH. Space/time trade-offs in hash coding with allowable errors. *Commun ACM* 1970;**13**:422–6.

Bradley P, den Bakker HC, Rocha EPC *et al.* Ultrafast search of all deposited bacterial and viral genomic data. *Nat Biotechnol* 2019;**37**:152–9. https://doi.org/10.1038/s41587-018-0010-1.

Camacho C, Coulouris G, Avagyan V *et al.* BLAST+: architecture and applications. *BMC Bioinformatics* 2009;**10**:1–9.

Chikhi R, Limasset A, Jackman S *et al.* On the representation of de Bruijn graphs. *J Comput Biol* 2015;**22**:336–52.

Chikhi R, Limasset A, Medvedev P. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics* 2016;**32**:i201–8. https://doi.org/10.1093/bioinformatics/btw279.

Deorowicz S, Debudaj-Grabysz A, Grabowski S. Disk-based k-mer counting on a PC. *BMC Bioinformatics* 2013;**14**:1–12.

Dolle DD, Liu Z, Cotten M *et al.* Using reference-free compressed data structures to analyze sequencing reads from thousands of human genomes. *Genome Res* 2017;**27**:300–9.

European Nucleotide Archive. *ENA Statistics — Reads Growth - Reads Doubling Time*. https://www.ebi.ac.uk/ena/about/statistics (18 January 2023, date last accessed).

Harris RS, Medvedev P. Improved representation of sequence bloom trees. *Bioinformatics* 2020;**36**:721–7.

Holley G, Melsted P. Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs. *Genome Biol* 2020;**21**:1–20.

Holley G, Wittler R, Stoye J. Bloom Filter Trie: an alignment-free and reference-free data structure for pan-genome storage. *Algorithms Mol Biol* 2016;**11**:1–9.

Janin L, Schulz-Trieglaff O, Cox AJ. BEETL-fastq: a searchable compressed archive for DNA reads. *Bioinformatics* 2014;**30**:2796–801.

Lemane T, Medvedev P, Chikhi R *et al.* kmtricks: efficient and flexible construction of Bloom filters for large sequencing data collections. *Bioinform Adv* 2022;**2**:vbac029.

Li H, Durbin R. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics* 2009;**25**:1754–60.

Marçais G, Kingsford C. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics* 2011;**27**:764–70.

Marchet C, Boucher C, Puglisi SJ *et al.* Data structures based on k-mers for querying large collections of sequencing data sets. *Genome Res* 2021a;**31**:1–12.

Marchet C, Kerbiriou M, Limasset A. BLight: efficient exact associative structure for k-mers. *Bioinformatics* 2021b;**37**:2858–65.

Mohamadi H, Khan H, Birol I. ntCard: a streaming algorithm for cardinality estimation in genomics data. *Bioinformatics* 2017;**33**:1324–30.

Muggli MD, Alipanahi B, Boucher C. Building large updatable colored de Bruijn graphs via merging. *Bioinformatics* 2019;**35**:i51–60.

Muggli MD, Bowe A, Noyes NR *et al.* Succinct colored de Bruijn graphs. *Bioinformatics* 2017;**33**:3181–7.

Pibiri GE. Sparse and skew hashing of k-mers. *Bioinformatics* 2022;**38**:i185–94.

Roberts M, Hayes W, Hunt BR *et al.* Reducing storage requirements for biological sequence comparison. *Bioinformatics* 2004;**20**:3363–9.

Shen W, Le S, Li Y *et al.* SeqKit: a cross-platform and ultrafast toolkit for FASTA/Q file manipulation. *PLoS One* 2016;**11**:e0163962.

Solomon B, Kingsford C. Fast search of thousands of short-read sequencing experiments. *Nat Biotechnol* 2016;**34**:300–2.

Solomon B, Kingsford C. Improved search of large transcriptomic sequencing databases using split sequence bloom trees. *J Comput Biol* 2018;**25**:755–65.

Yu Y, Liu J, Liu X *et al.* SeqOthello: querying RNA-seq experiments at scale. *Genome Biol* 2018;**19**:167.