Conway-Bromage-Lyndon (CBL): an exact, dynamic representation of k-mer sets

Igor Martayan[1,✉], **Bastien Cazaux**[1], **Antoine Limasset**[1*], **and Camille Marchet**[1*,✉]

[1]UMR9189 CRIStAL, Univ Lille, CNRS, Centrale, F-59000 Lille, France

In this paper, we introduce the Conway-Bromage-Lyndon (CBL) structure, a compressed, dynamic and exact method for representing k-mer sets. Originating from Conway and Bromage's concept, CBL innovatively employs the smallest cyclic rotations of k-mers, akin to Lyndon words, to leverage lexicographic redundancies. In order to support dynamic operations and set operations, we propose a dynamic bit vector structure that draws a parallel with Elias-Fano's scheme. This structure is encapsulated in a Rust library, demonstrating a balanced blend of construction efficiency, cache locality, and compression. Our findings suggest that CBL outperforms existing k-mer set methods, particularly in dynamic scenarios. Unique to this work, CBL stands out as the only known exact k-mer structure offering in-place set operations. Its different combined abilities position it as a flexible Swiss knife structure for k-mer set management. Availability: https://github.com/imartayan/CBL

k-mers | set | necklaces | data-structures

Correspondence: *igor.martayan@univ-lille.fr, camille.marchet@univ-lille.fr*

1: Introduction

Recent improvements in sequencing significantly advanced assembly methods, which feed many downstream analysis. This progress is reflected in the accumulation of genomes ready to be integrated into analyses. Real-time analysis becomes tangible with current throughputs, for instance in the monitoring of pathogen outbreaks, and will increasingly demand more dynamic data structures. Research in sequence data structures stands to benefit by anticipating that data will soon be amassed so rapidly that we will need to adapt our algorithms accordingly.

K-mer based methods are valued for their scalability and versatility in sequence analysis. A prominent example is assembly, including short and high fidelity long reads (1), and quantification of gene expression, correction (2) or sequence comparison (3) can be cited as other examples. In pangenomics, k-mer based methods are also instrumental in enhancing the scalability (4) and have spurred innovations microbial studies (5, 6). Depending on the application, even a single dataset can reach billions of k-mers. Different solutions for representing k-mer sets have been proposed, focusing on opposed efficiency tradeoffs: speed built on linear algorithms and cache efficiency, versus space efficiency through a form of compression. Current literature mainly presents static k-mer sets, which do not support intensive insertions and deletions of elements. Static methods have been preferred in the case of large data volumes, for their excellent space/time performances, and because it was sufficient to build the k-mer index once. The datasets turnover might drive the problem to a state where an expensive construction time cannot be paid too often and becomes a bottleneck. Therefore, the existence of a generic, dynamic structure for general k-mer set operations is appealing but not yet described. We aim at discussing novel approaches and implementations to reach this goal.

A. New results. We introduce a data-structure for the dynamic, exact representation of k-mer sets, inspired from Conway and Bromage's work. Named Conway-Bromage-Lyndon (CBL), it stores the smallest cyclic rotations of k-mers, akin to Lyndon words. This approach leverages the lexicographic similarities between consecutive k-mers. We also describe an accompanying structure enabling insertions and deletions, showing its competitive edge over current methods. We jointly provide a Rust library, available at https://github.com/imartayan/CBL, which aims at being an ubiquitous tool for manipulating dynamically k-mer sets.

B. Outline of this work. After describing related work, we present two main aspects. First we show the contribution of the k-mer $\rightarrow$ smallest cyclic rotation to the problem of locality-preserved k-mer representation, and second, we describe the data-structures to achieve a dynamic compressed set representation exploiting this transform. Then we provide evaluations with regard to state-of-the art approaches on time and memory for construction, and the different operations. In the discussion, we recall open problems linked to this contribution and identify ways to go forward.

2: Methods

A. Related work. Exact sets of k-mers data-structures include solutions that highly vary in efficiency and implementations. This litterature can be difficult to approach because the interplay of these structures is not trivial. We propose a summary highlighting connections of the different approaches in Table 1. Succinct representations upon the Burrows-Wheeler transform and its variations take advantage of lexicographic redundancies to store and index k-mers in a compressed way. Another direction uses space-efficient hashing combined to k-mer surperstrings (7) to build associative structures for k-mers. Other hashing techniques are sometimes employed, such as quotienting. Earlier lines of work include arborescent structures to store and index k-mer sets; or exploit specificities of de Bruijn graphs to represent them in a lossless way in Bloom filters.

DRAFT

Conway and Bromage (8) is an exception that strikes by its simplicity, as it neither lexicographically transforms, nor hashes the k-mers. A k-mer is represented by a single bit in a sparse bit-vector, using its binary encoding for addressing. This method sets a lower bound for k-mer representation but faces challenges in finding a dynamic, compressed data structure for the bit-vector. It also does not capitalize on redundancy in the initial set for space reduction.

These diverse data structures support different operations. Most allow membership queries of k-mers. Some can propose operations typical of sequence de Bruijn graphs, such as color (output the sources of a k-mer when the graph is built from different datasets), navigate (from a given node, visit the direct neighbours, which allows assembly tasks). Other set operations such as union or intersection are usually not supported, and remarkably, these features are being only discussed in the lossy setting (9, 10). Only a few exact structures allow insertion of novel k-mers, even less allow deletions (see Table 1 for a summary of allowed operations per structure).

B. Representing a k-mer set using smallest cyclic rotations. In this work, we transform k-mers into other strings and insert them in a sparse structure such as Conway and Bromage, with two major advantages over the original Conway and Bromage approach. The first one is that the transform and the data-structure allows us to leverage the k-mers overlaps and locality, and the second is that we present a fully dynamic sparse structure to dynamically insert the set.

B.1. Necklaces as k-mer representatives. K-mers are finite strings of size k on the 4-sized alphabet $\Sigma = \{A, C, G, T\}$. We rely on a k-mer encoding described in (36), that allows to encode canonical k-mers using $2k-1$ bits. An example is given in Figure 1.

Definition 1: **Necklace.** We define the *necklace* of a k-mer as the smallest string among cyclic rotations of the k-mer.

Necklaces are related to Lyndon words (37), the strings that are strictly smaller in lexicographic order than any of their rotations, hence the name of our structure. When encoding a canonical set of k-mers, we map k-mers and their reverse complement to the same necklace, thus reducing the space from $[0, 2^{2k-1}-1]$ to $[0, 2^{2k-1}/2-1]$.

In the following we consider k-mers as binary words of size $2k-1$ (for the sake of clarity, most examples and illustrations will be given using a textual representation of k-mers).

Definition 2: **Binary necklace.** Binary necklaces are the smallest bit string among cyclic rotations on the $2k-1$ bits binary encoding of canonical k-mers.

In the following, we denote binary necklaces simply by necklaces. We show an example of k-mers conversion to their binary necklaces in Figure 1.

We note that the smallest cyclic rotation appears in the literature outside this paper. It was recently used as a k-mer sampling technique (38, 39). But we are not aware of other usage of this technique in the context of exact k-mer set representation, and the most widely used k-mer transform for sequence data-structures remains the Burrows Wheeler transform. As all the substrings of fixed size of k-mers are substrings of fixed size of the circularization of these k-mers, we can find some necklaces as prefixes of all circular rotations of a sequence. Consequently, the first characters of the BWT (and thus the BOSS and SBWT representations) correspond to necklaces. The similarity of the information carried by different BWT approaches with CBL is summed up in Figure 2. Minimizers used in other structures such as SSHash are the smallest substring of a fixed size but they do not follow the lexicographic order in practice.

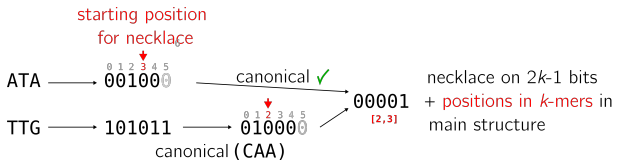

Fig. 1. Example of conversion into necklace for two k-mers of size 3. his operation is used for insertion and query to the structure. We show how k-mers are encoded into $2k-1$ bits. We use a parity definition as the parity of the population count (number of 1's) of the binary encoding of k-mers to define canonical k-mers. Thus, provided bases and their complement have different parity (e.g., $A \to 00$, $C \to 01$, $G \to 11$, $T \to 10$) and k-mers have an odd length, a k-mer and its reverse complement has a different parity each. By choosing one parity as canonical, it is therefore possible to spare a bit for their encoding, i.e. to use $2k-1$ bits instead of $2k$. With a regular encoding, the k-mers' necklaces are different, contrary to our example. ATA's necklace would have been 001100 $\to$ 000011 and TTG's necklace would have been 111110 $\to$ 0111111 (note that necklaces would have been different even if using reverse complements). In our case, we first convert k-mers to their canonical versions (here, the canonical version has an odd number of 1's), the last bit is then safely removed. The necklaces are computed on these $2k$-1 bits, and their starting positions in k-mers are remembered. Necklaces and positions are stored in the main structure.

B.2. Distribution of necklaces in the image space. The initial k-mer set is supposed to be extracted from biological sequences, and therefore sparse in $[0, 2^{2k-1}-1]$. So is the corresponding distribution of necklaces. It is straightforward that k-mers can have the same necklace, but the necklace and the position of the corresponding rotation is sufficient to represent distinctly each k-mer. Hence, the distribution of k-mers' necklaces over $[0, 2^{2k-1}/2-1]$ is skewed towards small values.

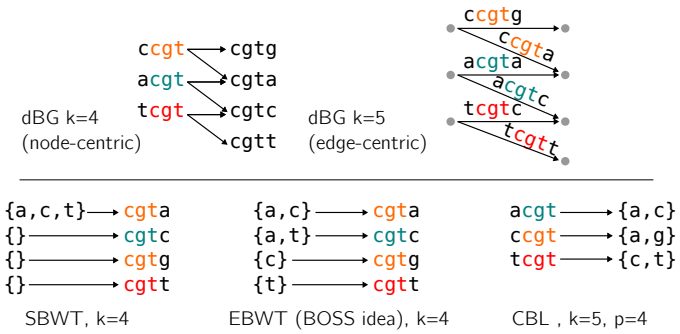

Fig. 2. K-mer information supported in each transform-based data-structure. This information can be stored as the de Bruijn graph of order $k-1$ by using, for example, the extended Burrows-Wheeler Transform (EBWT). Indeed, the arcs in the dBG represent the set of successive $k-1$ mers, which is the set of k mers. This structure stores successively the different labels of the ingoing arcs by lexicographical order of the nodes, which correspond to the $k-1$ mers. Another way is to use the SBWT structures, which use a k-mer to represent all the arcs ingoing in this, or similar, k-mers (all k-mers with the same prefix of size $k-1$) in the previous structure. Even if we lose some information (false positive arcs and thus false positive k-mers), we can recover all $k-1$-mers with this structure.

category	k-mer representation	set data-structure	data-structure name	implementations using the structure (features)
tree/trie	strings	(burst) trie		BFT (11) (MEM,COL), RedOak (12) (MEM,COL)
		suffix tree		splitMEM (13) (COL)
BWT		FM-index	FM-index (14)	DBGFM (15) (NAV)
		dynamic BOSS	BOSS (16)	dynBOSS (17), bufBOSS (18) (MEM,INS,DEL), VARI-Merge (19) (MEM,COL, INS), Metagraph (20) (MEM,COL, INS)
		Wheeler graph	SBWT (21)	Themisto (22) (MEM,COL)
hashing		hash table		Bifrost (23) (MEM, NAV, COL, INS)
		MPHF	BLight (24), SSHash (25)	Fulgor (26) (MEM, COL), Reindeer (27) (MEM, COL) Cuttlefish2 (28) (NAV), Pufferfish2 (29) (MEM)
	integers		BBHash (30)	FDBG (31) (MEM, INS, DEL)
		quotient filter	Squeakr (32)	dynamic Mantis (33) (MEM, COL, INS)
bit-vectors	bits	Bloom Filter +flagged false positives	Cascading Bloom filters (34)	Minia (34) (NAV)
		sparse bit vector	Conway & Bromage (8)	Gossamer (35) (NAV)
		dynamic bit vector	**Conway Bromage Lyndon** (this paper)	**CBL (MEM, NAV, INS, DEL, union, intersection, difference)**

Table 1. K-mer sets data structures for exact representation. The table reads as follows: category denotes one of the four main approaches to represent sets of k-mers. The second column shows the internal k-mer set representation. Arborescent structures, methods based on BWTs and hashing represent k-mers as (sub)strings or integers, while bit-vector based methods represent k-mers as bits. The third column shows the data structure that mainly supports the k-mer set. The fourth column recalls these set data-structure have been named (when there is no specific name we leave a blank). The fifth column details the existing implementations using the structures. For instance, Fulgor implements k-mer membership (MEM) in a set of k-mer sets (COL), and is based on SSHash, an MPHF hashing approach. Indicated features are those actually implemented in the tools. Features abbreviations MEM: k-mer membership, COL: set coloration (distinction between different source datasets), NAV: navigation operations, allowing tasks such as unitig construction, INS: insertion, DEL: deletion.

B.3. Computing necklaces of consecutive k-mers in $\mathcal{O}(\log k)$. Computing the necklace of a k-mer is in $\mathcal{O}(k)$ (since every cyclic rotation must be considered), an operation that starts to show its cost for large k values. In practice, k-mer inserted in sets are extracted from sequences and two consecutive k-mers only differ by a single nucleotide, thus have a $1/4$ chance $(1/|\Sigma|)$ to have the same necklace. We propose a solution to speed-up the computation of necklaces in the case of consecutive k-mers. Our solution is based on the observation that the necklaces of two consecutive k-mers likely share a long common prefix.

What's more, we know that the prefix of size m of a necklace is (lexicographically) smaller than any other substring of size m in the circular word. Thus, in order to find the starting position of the necklace in the k-mer, we only consider the starting positions corresponding to the smallest substrings of size m. Because of circularity, these substrings are either part of the k-mer or overlap both the start and the end of the k-mer (we call these boundary substrings). Boundary substrings change for each new k-mer, so we have to recompute them every time. On the other hand, non-boundary substrings are mostly preserved for consecutive k-mers. By storing them in a monotone queue, we can access the smallest ones in $\mathcal{O}(1)$ and perform updates in $\mathcal{O}(1)$ amortized time. Moreover, we know that if m is large enough ($\Omega(\log k)$) the smallest substring is unique with high probability (see lemma 9 in (40)). Therefore, by choosing $m = \Theta(\log k)$, we only have one substring to consider in the queue w.h.p. and $m-1$ boundary substrings to compute, leading to $\mathcal{O}(\log k)$ time overall. In practice, we found that using substrings of $m = 9$ bits gave the best results for k ranging from 31 to 63. Our implementation of this algorithm can compute 100M necklaces per second on a laptop with a M1 processor.

B.4. Rank of necklaces. Since the number of binary necklaces of length $2k-1$ is approximately $\frac{2^{2k-1}}{2k-1}$ (41), they only represent a fraction $\frac{1}{2k-1}$ of the k-mer space. This means that we could work on a reduced space, $2k-1$ times smaller than the original one, and thus save $\log_2(2k-1)$ bits to represent a necklace. One way to achieve this is to rank necklaces, i.e. associating index i to the i-th smallest necklace. Sawada and Williams proposed an algorithm to compute the rank of a necklace in $\mathcal{O}(k^2)$ (37), and we do not know a faster algorithm as of today. Unfortunately, this quadratic complexity makes it impractical to use as it is two order of magnitude slower than the other steps of the transformation.

C. Data-structures for k-mer set representation based on necklaces.

Similarly to Conway and Bromage, we propose to insert and query necklaces from a vector. However, to retrieve the original k-mers, necklaces must be associated to their positions in k-mers, and to support insertions and deletion, the associative structure must be dynamic. In the following we propose a compressed, dynamic representation of the necklaces vector that exactly represents a set of k-mers. Figure 3 presents a schematized view of the structure.

C.1. Quotienting necklaces. To achieve compression, we employ a quotienting strategy on the necklaces. This involves selecting a prefix size p (the suffix being $k-p$ long). We then store prefixes and suffixes separately. The rationale behind this approach is the observation that many necklaces share prefixes, particularly as their distribution is skewed towards smaller sequences.

C.2. Prefix data-structure. For the storage of these prefixes, we utilize a bitvector. To facilitate rapid access and insertion of prefixes in the bitvector using their rank, a structure

capable of performing efficient rank/select operations is required. Several such structures have been discussed in the literature (42–44). However, we note a sparsity in the literature for dynamic structures. Relying on a theoretical proposition and an implementation (45, 46), we adopted a solution with fast dynamic rank/select operation.

A Fenwick tree, an efficient data structure to compute the prefix sums of a dynamic array, is behind our prefix data-structure. In our context, the Fenwick tree is used for an array of bits, where each tree node stores the cumulative sum of a specific range of elements. Since our elements are bits, the binary rank is equal to the prefix sum (cumulative sum from the left) found in the tree. This arrangement allows for quick computation (in logarithmic time) of a given rank by searching the subtree's nodes corresponding to the interval.

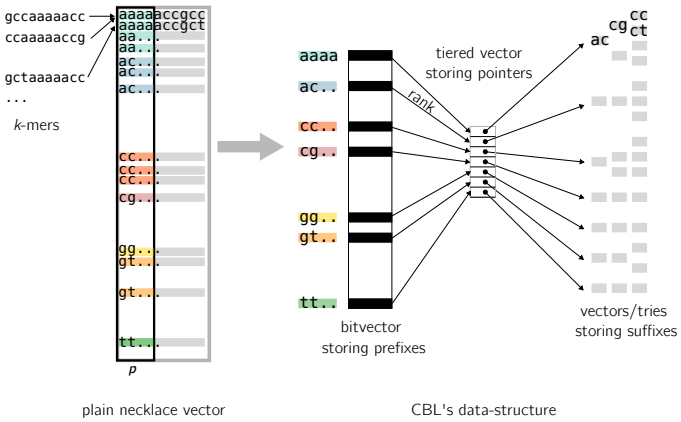

Fig. 3. Schematic view of CBL's data structure. For the sake of simplicity, we show an example where the necklaces are lexicographic and not binary. On the left we show the entire necklace vector filled with input k-mers' necklaces. Two overlapping k-mers are pictured sharing very locally close necklaces. We depicted how our structure quotients necklaces, stores prefixes in a bit vector, and associates them to suffixes.

C.3. Associating prefixes to suffixes. For associating each prefix with its corresponding list of suffixes, we used a tiered vector (47). This structure facilitates the rapid association of a prefix's rank with buckets of suffixes, by storing a pointer to the suffix bucket at the index corresponding the prefix's rank. Insertion and deletion in this structure are supported at any position in $\mathcal{O}(n^{\varepsilon})$ time, where n is the number of elements, and $\varepsilon < 1$, while access is still in $\mathcal{O}(1)$.

Tiered vectors maintain a dynamic array of elements sorted by ranks. The elements are stored in the leaves of a tree, with fast insertion facilitated by inserting elements in available slots. For rapid access, the tree maintains a shallow depth (in our case the internal depth is 4). While the order of elements in the leaves is not strictly preserved due to this insertion process, internal nodes of the tree store the necessary rotations to reorder the leaves.

C.4. Suffixes storage. As indicated in previous studies, the distribution of small-sized genomic words is highly skewed. We anticipate many buckets being populated with a single or a very few suffixes, and a minority containing numerous suffixes. To address this, we propose an adaptive solution: for small buckets, we store the set of suffixes in a vector; for large buckets, we construct a trie over the suffixes. Large buckets, often resulting from genomic repeats or low complexity regions, are expected to have high suffix similarity. Thus, a trie can effectively compress and encapsulate much of this redundancy. Our trie implementation utilizes 1 byte per level. For vector-based storage of small buckets, we similarly divide suffixes into bytes to minimize storage requirements. Finally, some bits in the end of suffixes are reserved to store the necklace's position in k-mers.

C.5. Operations. The main operations on the data structure (membership, insertion, deletion) all share the same major steps:

1. compute the necklace associated to a k-mer and append the associated position in the k-mer at the end of the necklace
2. split the necklace into a prefix q of size p and a suffix r
3. look for the prefix in the bitvector
4. compute the rank of this prefix using the Fenwick tree
5. find the bucket associated to this rank using the tiered vector
6. perform membership/insertion/deletion of the suffix r in this bucket:
   - for small buckets, we scan the vector linearly
   - for large buckets, we navigate in the trie byte by byte

Additionally, when the necklaces of consecutive k-mers share the same prefix q, we only need to perform steps 1-5 once, and we can group the remaining operations on the same bucket. This optimization is especially useful when querying all the k-mers of a sequence.

D. Implementation details. Our structure stores both a necklace and its position in a single primitive integer (with up to 128 bits) to perform fast operations. As a consequence, the number of bits $2k-1+\log_2(2k-1)$ must be smaller than 128, and k values are supported up to 59.

The size p of the prefixes stored can be parameterized at compilation time, and is supported for up to 28 bits. We settled on a default value of $p = 24$ bits as a good compromise between the size of the prefix bitvector (2^p) and the size the suffix buckets. For very large sets, increasing p helps to reduce the load on the buckets, speeding up the operations on the buckets.

By default, buckets store suffixes in a vector, and switch to a trie structure if they contain more than 1024 elements. This threshold was selected empirically as a compromise between the cost of a linear search in a contiguous vector and the cost of a layered search in a trie.

D.1. Informal comparison to Elias-Fano encoding. A comparison can be drawn between CBL and the Elias-Fano encoding for compressing sorted integers. Similar to Elias-Fano encoding, in the CBL approach, the elements are quotiented, with their prefixes and suffixes stored separately using two distinct methods. However, a key distinction lies in the treatment of suffixes. Elias-Fano encoding can omit an associative structure between prefixes and suffixes, and opts to store all suffixes together in a compact way. Although Elias Fano supposedly achieves higher compression rates, it compromises efficient dynamism; since inserting a suffix requires shifting numerous elements. In contrast, CBL may exhibit a lower compression ratio for the same input but inherently accommodates dynamic operations at every stage of the data structure, offering a more flexible solution for scenarios requiring frequent updates.

3: Results

We conducted extensive benchmarks on the CBL library, focusing on its ability to represent and manipulate k-mers in various biological datasets. All experiments were performed on a single cluster node running with Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz with 128GB of RAM and Ubuntu 22.04. The experiment scripts with all competing tools and the code to generate the plots are available at https://github.com/imartayan/CBL_experiments.

A. Index construction. For our initial experiment, we elected to benchmark several state-of-the-art tools using an expanding collection of bacterial genomes from RefSeq. We randomly selected bacterial genomes from RefSeq and constructed a compacted de Bruijn graph on subsets, each doubling in size at every step. The final unitig file comprised 1024 genomes and 1,574,701,184 distinct k-mers. We built several indexes from those unitigs and reported the time and memory usage in Figure 4. Multiple observations can be made.

SSHash appears to be the best overall performer, being very fast to construct and using a low amount of memory. However, it is indeed a static structure and requires non-repeating k-mers. HashSet is similarly fast but more memory-intensive, while CBL is slightly slower but equally efficient in terms of memory usage. SBWT, despite being static, is slower than these tools and consumes an amount of memory comparable to HashSet. Bifrost is significantly slower but as memory-efficient as SSHash and CBL. Lastly, BufBOSS is equally slow but very memory-intensive.

The fact that CBL matches the high throughput of a knowingly high performance hash table, while being memory efficient is noteworthy.

We aim to emphasize that Bifrost, HashSet, and CBL uniquely stand out as the only tools capable of running directly on any sequence set without prerequisite modifications. In contrast, other tools necessitate preliminary processing of the input file to ensure the absence of duplicated k-mers. This preprocessing often involves a k-mer counting step, a resource-intensive operation. For instance, SBWT and BufBOSS employ the KMC3 k-mer counter, which is notably heavy on disk usage, potentially posing challenges in certain environments. SSHash's requirements are even more stringent, relying on overlapping k-mers assembly for efficiency. While such prerequisites might be challenging to estimate, it's important to highlight that Bifrost, HashSet, and CBL do not externalize a portion of their construction time. This fact makes their construction performance all the more noteworthy.

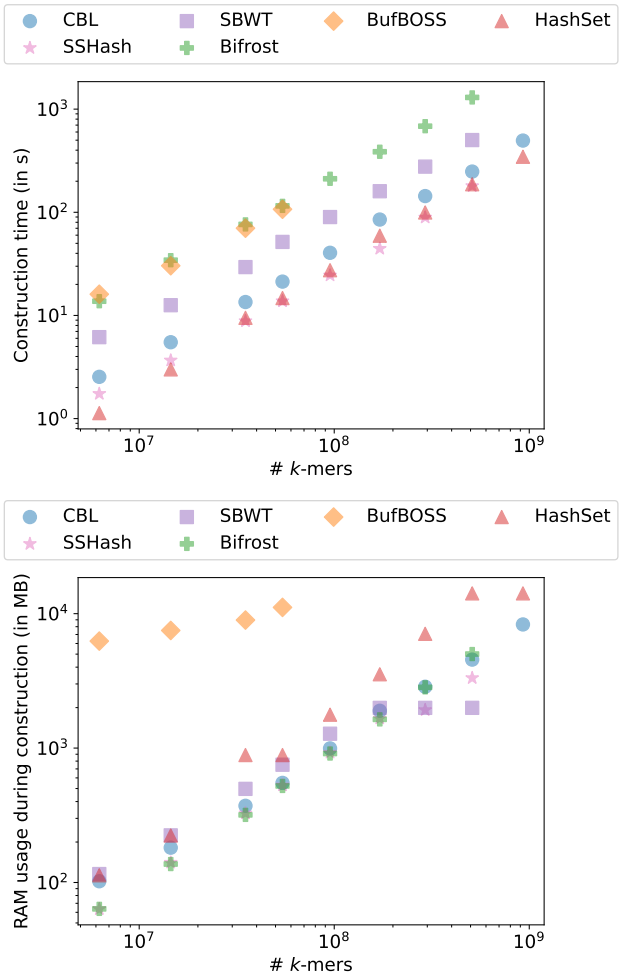

Fig. 4. Time and RAM used when constructing various indexes on growing bacterial genomes collection from Refseq for $k = 31$ and $p = 28$ bits.

B. Queries. To evaluate the query performance of the previously mentioned indexes, we decided to separately benchmark positive queries (k-mers present in the index) and negative ones. For positive queries, we used the input unitigs' k-mers and queried them against their own index. For negative queries, we queried randomly generated k-mers that have an extremely low chance of being present in the index. We present these results in Figure 5: to illustrate the respective time/memory trade-offs, we measured the relative query time and RAM usage with respect to the number of k-mers queried, averaged over a series of experiments.

As anticipated, static structures and BufBOSS consume a very low amount of RAM during query execution, whereas

the other dynamic structures are more memory-intensive. Although SSHash and SBWT provide very fast query responses, BufBOSS exhibits very slow ones. Among the other dynamic structures, CBL and Bifrost are the lightest, while HashSet is more memory-demanding.

Interestingly, their relative speeds vary between positive and negative queries. While Bifrost is the fastest for positive queries, with CBL and HashSet being almost identical, it is the slowest for negative queries. Negative queries seem to be an ideal scenario for HashSet, being the absolute fastest, even surpassing SSHash.

Once again being able to achieve very high throughput while being more memory efficient than other dynamic structures highlight the overall performance of the structure.

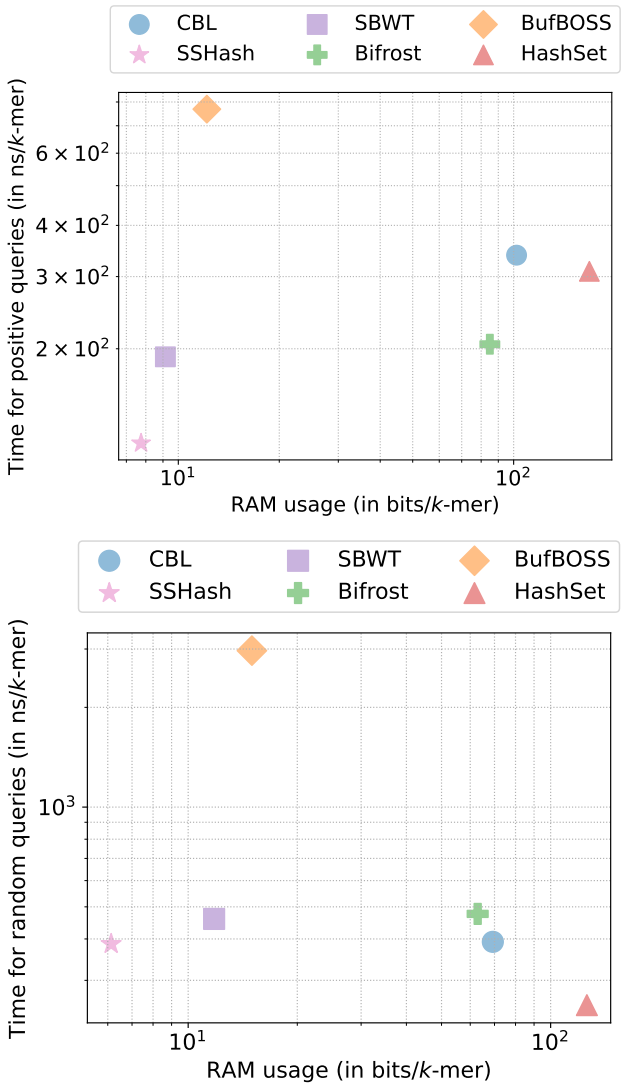

Fig. 5. Time/memory trade-off of various tools when performing queries of present k-mers (up) and absent k-mers (down) for $k = 31$ and $p = 28$ bits, each point is averaged over a series of experiments.

C. Insertion and deletion. To assess index update performance, we analyzed the cost of adding and removing k-mers, with the results presented in Figure 6. Among the four tools capable of insertion, CBL and HashSet stand out as the fastest, processing each k-mer in under a microsecond. In contrast, Bifrost is approximately an order of magnitude slower, with BufBoss trailing at yet another order of magnitude behind. In terms of memory usage, all tools are comparably efficient, falling within the same order of magnitude. BufBoss proves to be the most memory-efficient, followed by Bifrost HashSet and CBL.

For the deletion operation, which Bifrost does not support, the three remaining tools demonstrate similar performance. CBL and HashSet lead in speed, with BufBoss lagging an order of magnitude behind. Memory usage is lowest for BufBoss, followed by CBL, and with HashSet being the most memory-intensive.

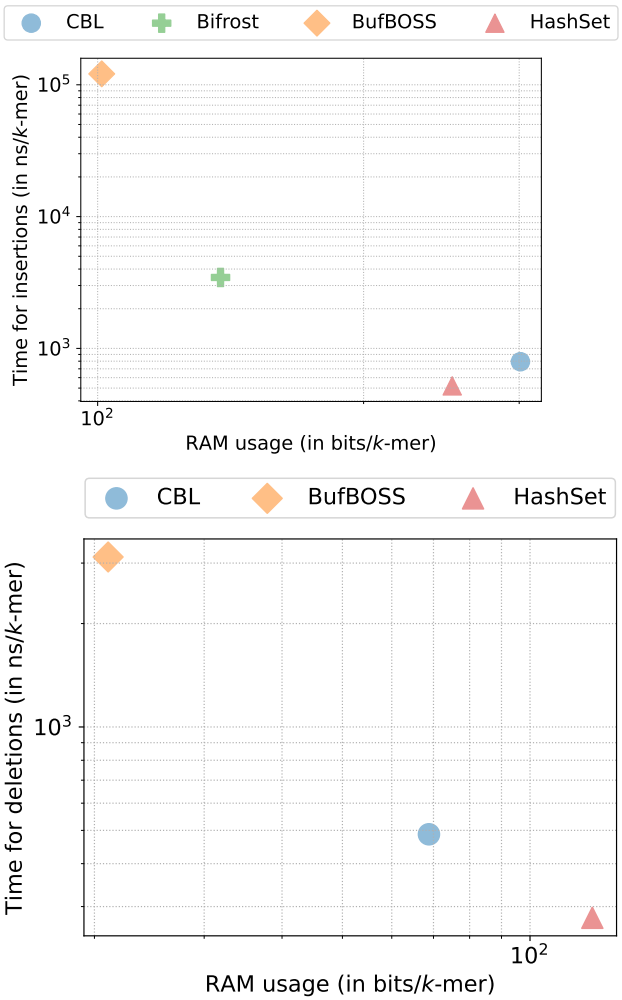

Fig. 6. Time/memory trade-off of various tools when performing k-mer insertions (up) and deletions (down) for $k = 31$ and $p = 28$ bits, each point is averaged over a series of experiments.

D. Set operations. CBL stands out as the sole method offering in-place set operations among the benchmarked approaches. We assessed its performance in terms of runtime and RAM usage for intersection (see Figure 7) and union operations (see Figure 8), comparing it with HashSet, our most comparable dynamic competitor. The results show that both HashSet and CBL efficiently handle intersections and unions of datasets, each containing tens of millions of k-mers with

a resource advantage for CBL. CBL achieves 200 ns per k-mer on average during intersection, and 500 ns per k-mer for union operations, while HashSet requires 400 and 900 ns per k-mers for the same tasks. Comparable trends were observed in operations involving set differences and symmetric differences.

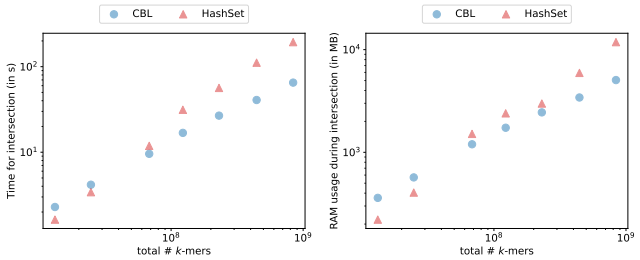

Fig. 7. Time/memory trade-off of various tools when performing set intersections for $k = 31$ and $p = 28$ bits.

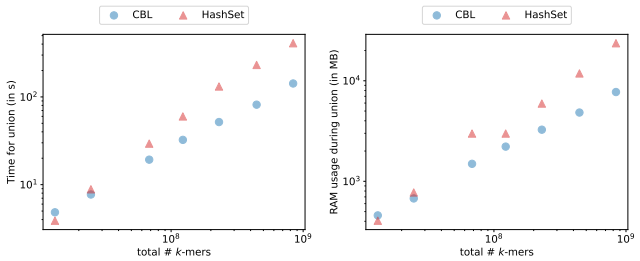

Fig. 8. Time/memory trade-off of various tools when performing set union for $k = 31$ and $p = 28$ bits.

E. Other type of data. While bacterial genome are convenient for benchmark purpose we also try to take into account other type of data. We performed a wide benchmark on human RNAseq data that is described in the Appendix. The same relative behavior and trend are conserved even if we tend to observe a performance degradation due to the unitig fragmentation of such graph due to the datasets properties such as polyA tails, high variability regions, uneven coverage.

To validate our result on eukariotic data we decided to also benchmark CBL and HashSet against human chromosomes from the T2T human reference genome (48). We present those results in Figure 9, we show that both implementations display similar performance results and that indexing large eukaryotic genome is possible on a small node with less than a CPU hour.

As we stress before most existing indexes require data preprocessing to work. On the contrary, Bifrost, CBL and HashSet are able to work in a streaming fashion directly on any FASTA file. To support this claim, we ran build benchmarks directly on raw reads. In Figure 10 we report such benchmarks on a growing amount of long ONT reads from an *E. coli* sequencing (accession SRR26899125) that is a recent Nanopore sequencing with low error rate ($\approx 3\%$) we observe no practical hindrance compared to indexing genomes. Additionally we performed an identical benchmark on a *E. coli* HiFi dataset (accession SRR11434954) with very similar results showed in the appendix.

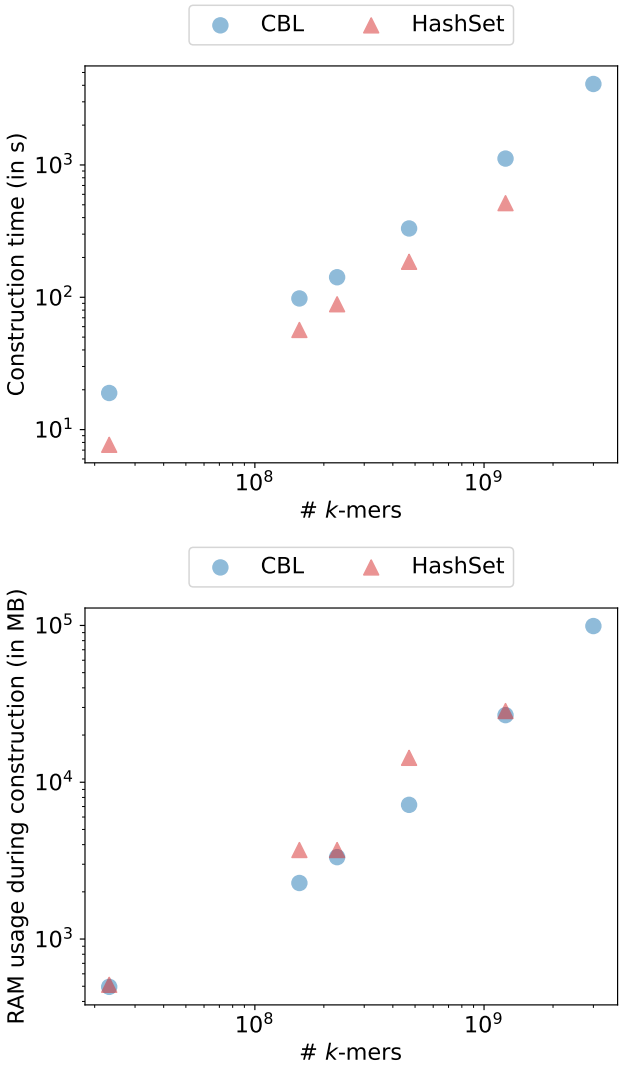

Fig. 9. Time and RAM used when constructing various indexes on growing amounts of chromosomes from the T2T human genome for $k = 31$ and $p = 28$ bits.

4: Discussion

Our experiments establish CBL as a good overall choice, consistently featuring as one of the most rapid and least memory-intensive tools across construction, querying, or updating tasks. Without manifesting any tangible drawbacks, it is also the unique method providing in place set operations.

Dynamicity is rarely completely achieved in state of the art, as many works revolve around static data-structures, with potential rapid rebuilding or buffer systems. These methods do not allow versatile inputs as they work on sequences containing distinct k-mers only. To handle any type of input FASTA, we showed that for moderate amounts of k-mers, a rational choice leads to select modern and efficient hash tables as those implemented in Rust. However, when the amount of input data is sufficient, the data-specificity of k-mer data-structures becomes an asset. Our work with CBL bridges this gap. Static data-structures remain an interesting choice for instance when resources are scarce, or to achieve a highly compressed cold storage.

Future developments for CBL include several aspects.

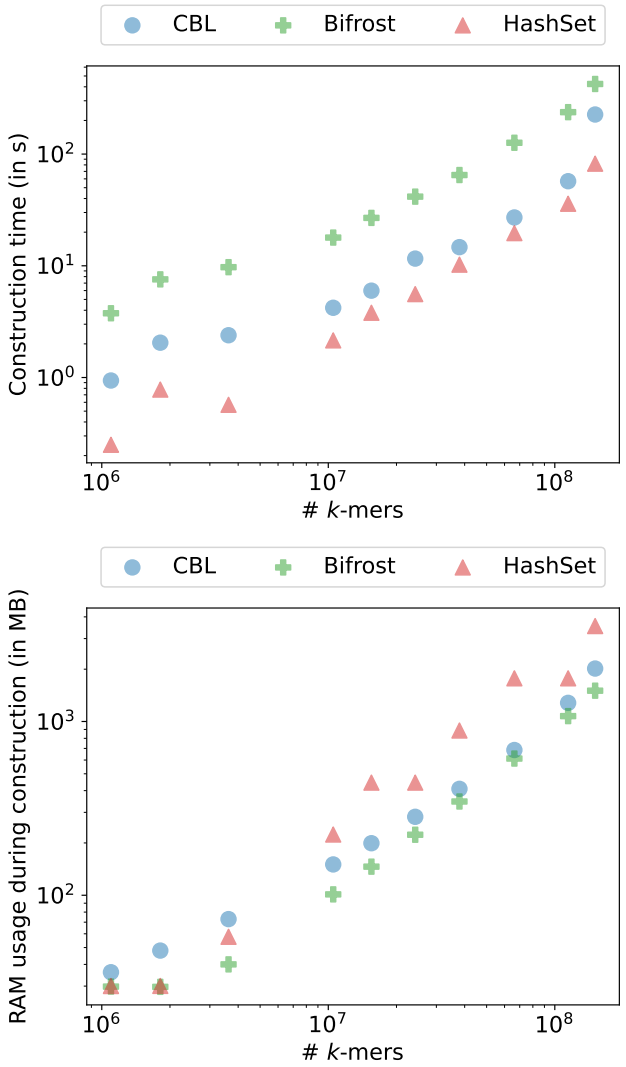

Fig. 10. Time and RAM used when constructing various indexes on a ONT long read dataset for $k = 31$ and $p = 28$ bits.

Adapting it to accept k-mer streams could better meet user requirements. We also aim to refine CBL's serialization, which is expected to improve construction time. CBL's dynamic bit-vector is also a good candidate for a compression scheme, such as Elias Fano. While Elias Fano's implementations are all static, a theoretical solution for dynamicity has been proposed by Pibiri and Venturini (49). CBL's structure will be a good starting point to explore possibilities of a subsequent implementation.

ACKNOWLEDGEMENTS
This work is partly funded by grants from the French National Research Agency: AGATE ANR-21-CE45-0012 and Find-RNA ANR-23-CE45-0003-01. The authors would like to thank Giulio Pibiri for his insights and relevant suggestions.

References

1. Anton Bankevich, Andrey V Bzikadze, Mikhail Kolmogorov, Dmitry Antipov, and Pavel A Pevzner. Multiplex de bruijn graphs enable genome assembly from long, high-fidelity reads. *Nature biotechnology*, 40(7):1075–1081, 2022.
2. Antoine Limasset, Jean-François Flot, and Pierre Peterlongo. Toward perfect reads: self-correction of short reads via mapping on de bruijn graphs. *Bioinformatics*, 36(5):1374–1381, 2020.
3. Camille Marchet and Antoine Limasset. Scalable sequence database search using partitioned aggregated bloom comb trees. *Bioinformatics*, 39(Supplement_1):i252–i259, 2023.
4. Francesco Andreace, Pierre Lechat, Yoann Dufresne, and Rayan Chikhi. Comparing methods for constructing and representing human pangenome graphs. *Genome Biology*, 24(1): 274, 2023.
5. Samuel T Horsfield, Gerry Tonkin-Hill, Nicholas J Croucher, and John A Lees. Accurate and fast graph-based pangenome annotation and clustering with ggcaller. *Genome Research*, 33(9):1622–1637, 2023.
6. Juri Kuronen, Samuel Horsfield, Anna K Pontinen, Sergio Arredondo-Alonso, Harry Thorpe, Rebecca Gladstone, Rob J L Willems, Stephen D Bentley, Nicholas J Croucher, Johan Pensar, et al. Pan-gwes: Pangenome-spanning epistasis and co-selection analysis via de bruijn graphs. *bioRxiv*, pages 2023–09, 2023.
7. Ondřej Sladký, Pavel Veselý, and Karel Břinda. Masked superstrings as a unified framework for textual k-mer set representations. *bioRxiv*, pages 2023–02, 2023.
8. Thomas C Conway and Andrew J Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, 2011.
9. Yoshihiro Shibuya, Djamal Belazzougui, and Gregory Kucherov. Efficient reconciliation of genomic datasets of high similarity. *bioRxiv*, pages 2022–06, 2022.
10. Téo Lemane, Paul Medvedev, Rayan Chikhi, and Pierre Peterlongo. Kmtricks: efficient and flexible construction of bloom filters for large sequencing data collections. *Bioinformatics Advances*, 2(1):vbac029, 2022.
11. Guillaume Holley, Roland Wittler, and Jens Stoye. Bloom filter trie: an alignment-free and reference-free data structure for pan-genome storage. *Algorithms for Molecular Biology*, 11 (1):1–9, 2016.
12. Clément Agret, Annie Chateau, Gaetan Droc, Gautier Sarah, Alban Mancheron, and Manuel Ruiz. Redoak: a reference-free and alignment-free structure for indexing a collection of similar genomes. *bioRxiv*, pages 2020–12, 2020.
13. Shoshana Marcus, Hayan Lee, and Michael C Schatz. Splitmem: a graphical algorithm for pan-genome analysis with suffix skips. *Bioinformatics*, 30(24):3476–3483, 2014.
14. Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st annual symposium on foundations of computer science*, pages 390–398. IEEE, 2000.
15. Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T Simpson, and Paul Medvedev. On the representation of de bruijn graphs. In *Research in Computational Molecular Biology: 18th Annual International Conference, RECOMB 2014, Pittsburgh, PA, USA, April 2-5, 2014, Proceedings 18*, pages 35–55. Springer, 2014.
16. Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de bruijn graphs. In *International workshop on algorithms in bioinformatics*, pages 225–235. Springer, 2012.
17. Bahar Alipanahi, Alan Kuhnle, Simon J Puglisi, Leena Salmela, and Christina Boucher. Succinct dynamic de bruijn graphs. *Bioinformatics*, 37(14):1946–1952, 2021.
18. Jarno Alanko, Bahar Alipanahi, Jonathen Settle, Christina Boucher, and Travis Gagie. Buffering updates enables efficient dynamic de bruijn graphs. *Computational and structural biotechnology journal*, 19:4067–4078, 2021.
19. Martin D Muggli, Bahar Alipanahi, and Christina Boucher. Building large updatable colored de bruijn graphs via merging. *Bioinformatics*, 35(14):i51–i60, 2019.
20. Mikhail Karasikov, Harun Mustafa, Daniel Danciu, Christopher Barber, Marc Zimmermann, Gunnar Rätsch, and André Kahles. Metagraph: Indexing and analysing nucleotide archives at petabase-scale. *BioRxiv*, pages 2020–10, 2020.
21. Jarno N Alanko, Simon J Puglisi, and Jaakko Vuohtoniemi. Small searchable κ-spectra via subset rank queries on the spectral burrows-wheeler transform. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA23)*, pages 225–236. SIAM, 2023.
22. Jarno N Alanko, Jaakko Vuohtoniemi, Tommi Mäklin, and Simon J Puglisi. Themisto: a scalable colored k-mer index for sensitive pseudoalignment against hundreds of thousands of bacterial genomes. *bioRxiv*, pages 2023–02, 2023.
23. Guillaume Holley and Páll Melsted. Bifrost: highly parallel construction and indexing of colored and compacted de bruijn graphs. *Genome biology*, 21(1):1–20, 2020.
24. Camille Marchet, Mael Kerbiriou, and Antoine Limasset. Blight: efficient exact associative structure for k-mers. *Bioinformatics*, 37(18):2858–2865, 2021.
25. Giulio Ermanno Pibiri. Sparse and skew hashing of k-mers. *Bioinformatics*, 38 (Supplement_1):i185–i194, 2022.
26. Jason Fan, Noor Pratap Singh, Jamshed Khan, Giulio Ermanno Pibiri, and Rob Patro. Fulgor: A fast and compact k-mer index for large-scale matching and color queries. *bioRxiv*, pages 2023–05, 2023.
27. Camille Marchet, Zamin Iqbal, Daniel Gautheret, Mikaël Salson, and Rayan Chikhi. Reindeer: efficient indexing of k-mer presence and abundance in sequencing datasets. *Bioinformatics*, 36(Supplement_1):i177–i185, 2020.
28. Jamshed Khan, Marek Kokot, Sebastian Deorowicz, and Rob Patro. Scalable, ultra-fast, and low-memory construction of compacted de bruijn graphs with cuttlefish 2. *Genome biology*, 23(1):190, 2022.
29. Jason Fan, Jamshed Khan, Giulio Ermanno Pibiri, and Rob Patro. Spectrum preserving tilings enable sparse and modular reference indexing. In *International Conference on Research in Computational Molecular Biology*, pages 21–40. Springer, 2023.
30. Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. Fast and scalable minimal perfect hashing for massive key sets. *arXiv preprint arXiv:1702.03154*, 2017.
31. Victoria G Crawford, Alan Kuhnle, Christina Boucher, Rayan Chikhi, and Travis Gagie. Practical dynamic de bruijn graphs. *Bioinformatics*, 34(24):4189–4195, 2018.
32. Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. Squeakr: an exact and approximate k-mer counting system. *Bioinformatics*, 34(4):568–575, 2018.
33. Fatemeh Almodaresi, Jamshed Khan, Sergey Madaminov, Michael Ferdman, Rob Johnson, Prashant Pandey, and Rob Patro. An incrementally updatable and scalable system for large-scale sequence search using the bentley–saxe transformation, 2022.
34. Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de bruijn graph representation based on a bloom filter. *Algorithms for Molecular Biology*, 8:1–9, 2013.
35. Thomas Conway, Jeremy Wazny, Andrew Bromage, Justin Zobel, and Bryan Beresford-Smith. Gossamer—a resource-efficient de novo assembler. *Bioinformatics*, 28(14):1937–1938, 2012.
36. Roland Wittler. General encoding of canonical k-mers. *Peer Community Journal*, 3, 2023.

37. Joe Sawada and Aaron Williams. Practical algorithms to rank necklaces, lyndon words, and de bruijn sequences. *Journal of Discrete Algorithms*, 43:95–110, 2017.
38. Grigorios Loukides and Solon Pissis. Bidirectional string anchors: A new string sampling mechanism. In *ESA 2021-29th Annual European Symposium on Algorithms*, volume 204, pages 1–21, 2021.
39. Lorraine AK Ayad, Grigorios Loukides, and Solon P Pissis. Text indexing for long patterns: Anchors are all you need. *Proceedings of the VLDB Endowment*, 16(9):2117–2131, 2023.
40. Hongyu Zheng, Carl Kingsford, and Guillaume Marçais. Improved design and analysis of practical minimizers. *Bioinformatics*, 36(Supplement_1):i119–i127, 2020.
41. Monsieur Lothaire. *Combinatorics on words*, volume 17. Cambridge university press, 1997.
42. Sebastiano Vigna. Broadword implementation of rank/select queries. In *International Workshop on Experimental and Efficient Algorithms*, pages 154–168. Springer, 2008.
43. Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, 44(11):1287–1314, 2014.
44. Dong Zhou, David G Andersen, and Michael Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *Experimental Algorithms: 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings 12*, pages 151–163. Springer, 2013.
45. Stefano Marchini and Sebastiano Vigna. Compact fenwick trees for dynamic ranking and selection. *Software: Practice and Experience*, 50(7):1184–1202, 2020.
46. Giulio Ermanno Pibiri and Shunsuke Kanda. Rank/select queries over mutable bitmaps. *Information Systems*, 99:101756, 2021.
47. Philip Bille, Anders Roy Christiansen, Mikko Berggren Ettienne, and Inge Li Gørtz. Fast dynamic arrays. *arXiv preprint arXiv:1711.00275*, 2017.
48. Sergey Nurk, Sergey Koren, Arang Rhie, Mikko Rautiainen, Andrey V Bzikadze, Alla Mikheenko, Mitchell R Vollger, Nicolas Altemose, Lev Uralsky, Ariel Gershman, et al. The complete sequence of a human genome. *Science*, 376(6588):44–53, 2022.
49. Giulio Ermanno Pibiri and Rossano Venturini. Dynamic elias-fano representation. In *28th Annual symposium on combinatorial pattern matching (CPM 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

Appendix

Supplementary results.

Parametrization of CBL. Using similar bacterial datasets to those described in the main document, we analyze the impact of the prefix size in CBL's performance.

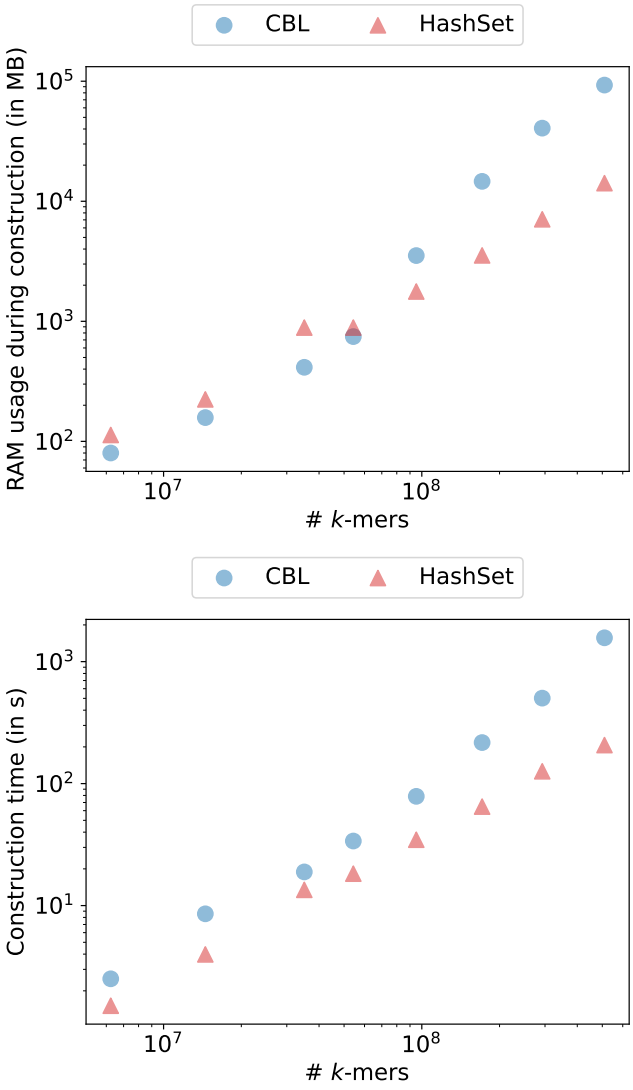

Fig. 11. Time and RAM used during construction on indexes with growing number of k-mers from bacterial genomes, for a prefix size of 22.

Differences in RAM usage between sizes 22 and 28 can be explained by the fact that increasing the prefix size on large instances implies that tries of suffixes are less populated, which decrease their overhead. Conversely, jumps to other suffixes buckets become more frequent during accesses.

Benchmarks on human RNA-seq. Illumina raw reads FASTA files were downloaded from SRA with accessions SRR972708, SRR975415, SRR962597, SRR976738, SRR953494, SRR950080, SRR950083, SRR953488, SRR972717, SRR972716, SRR975416, SRR975412, SRR962604, SRR976749, SRR953495. Unitigs were built using BCALM2, keeping k-mers with a multiplicy greater than 2. Additional files were used for query experiments,

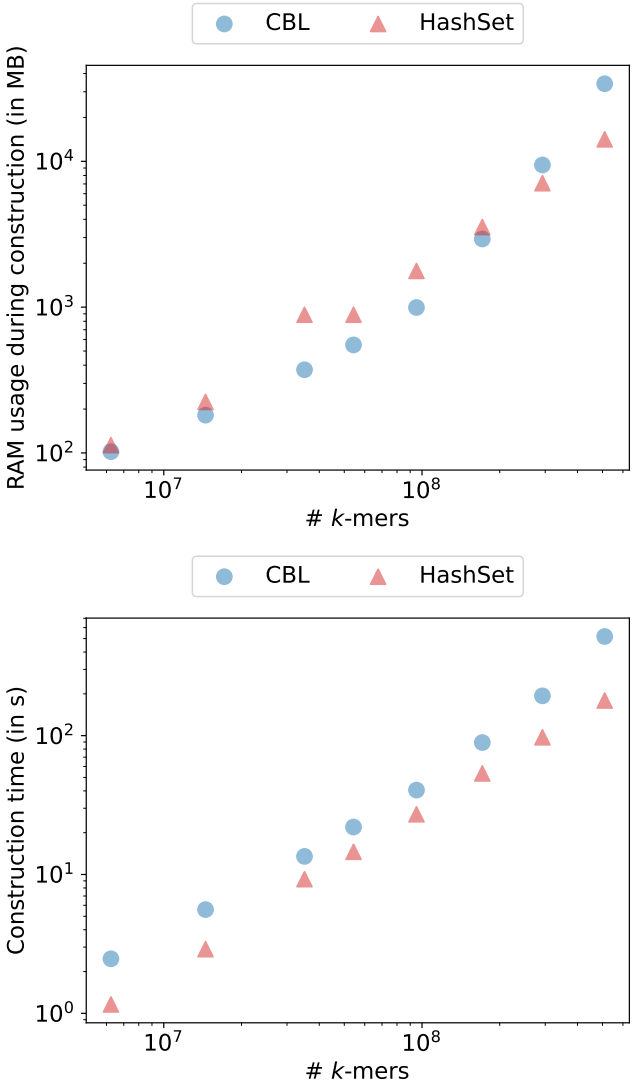

Fig. 12. Time and RAM used during construction on indexes with growing number of k-mers from bacterial genomes, for a prefix size of 24.

obtained in a similar setting: SRR975414, SRR950879, SRR976743, SRR950882, SRR950881, SRR950079, SRR972713, SRR972715, SRR960732, SRR960733, SRR975411, SRR962602, SRR962600, SRR950084.
We display here similar analysis of the main document, benchmarking index building, queries, insertions and deletions.

Benchmarks on HiFi reads. We report a building benchmark on a growing amount of HiFi reads from a *E.coli* sequencing (accession SRR11434954)

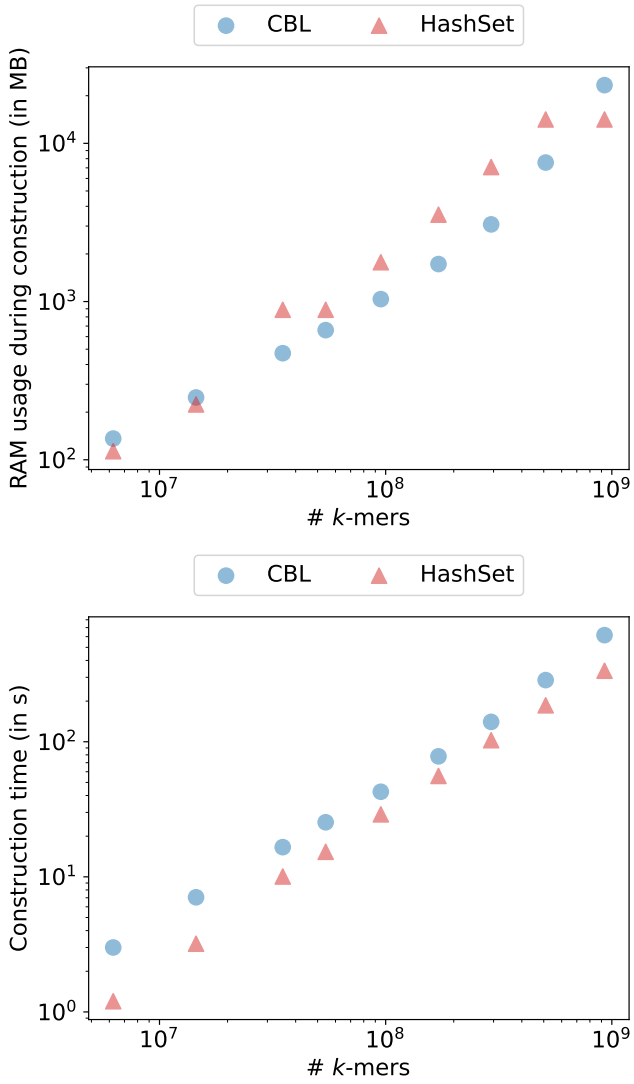

Fig. 13. Time and RAM used during construction on indexes with growing number of k-mers from bacterial genomes, for a prefix size of 26.

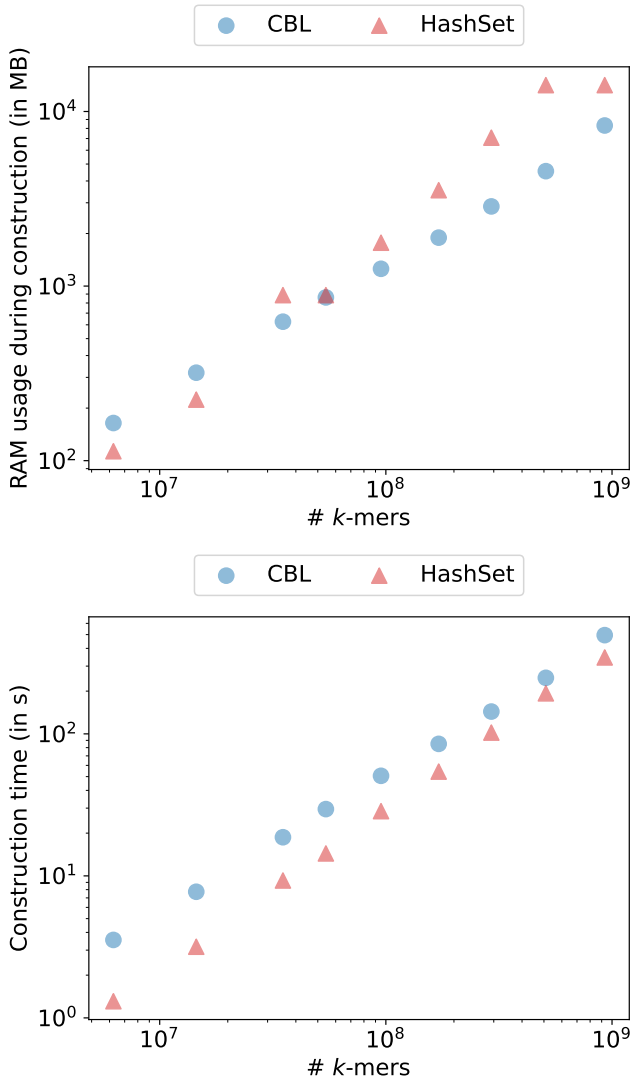

Fig. 14. Time and RAM used during construction on indexes with growing number of k-mers from bacterial genomes, for a prefix size of 28.

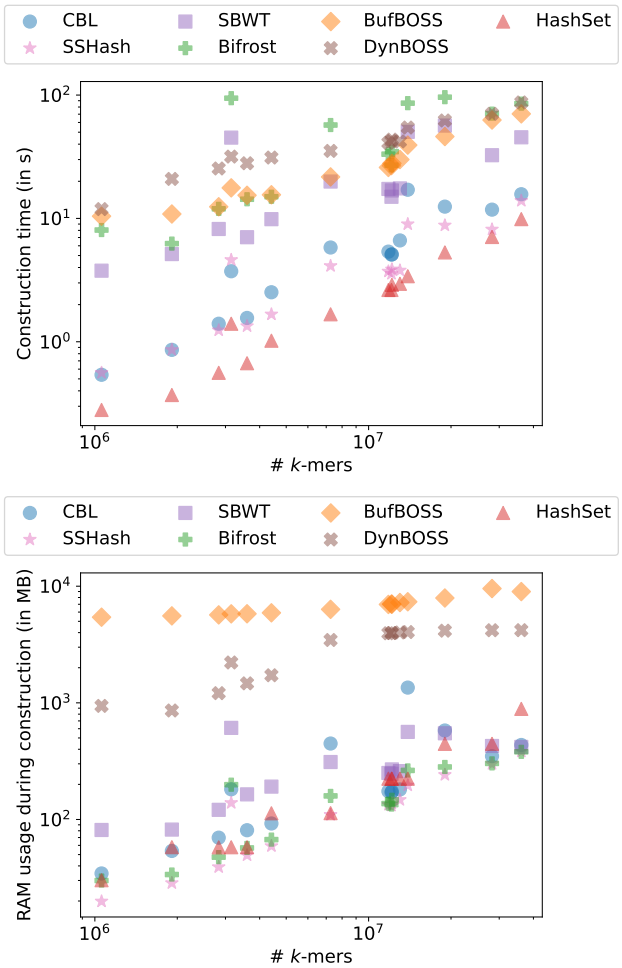

Fig. 15. Time and RAM used when construction various indexes on growing number of k-mers from unitigs built on human RNA-seq.

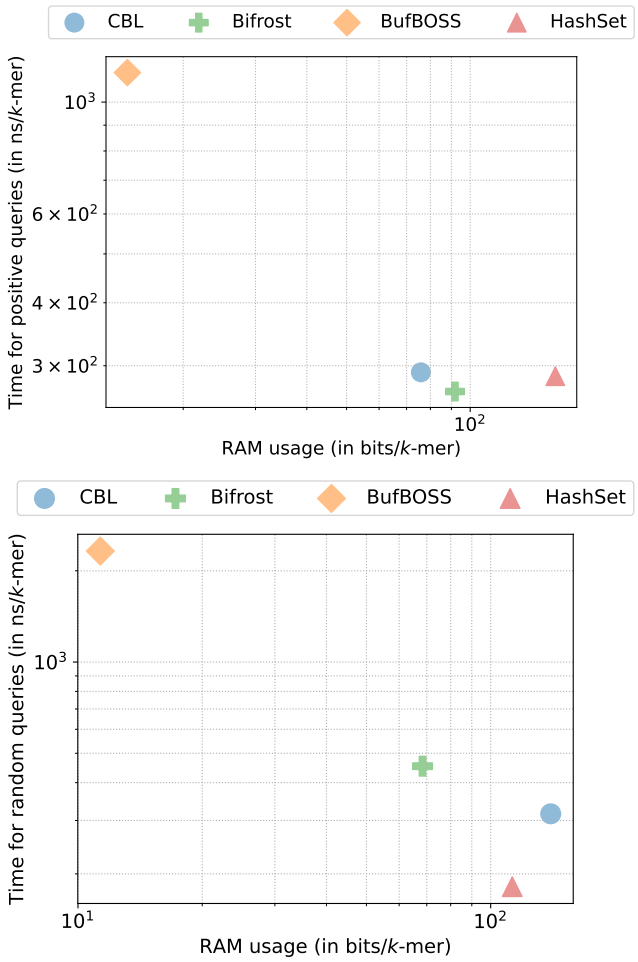

Fig. 16. Time/memory trade-off of various tools when performing positive queries (up) and negative queries (down)

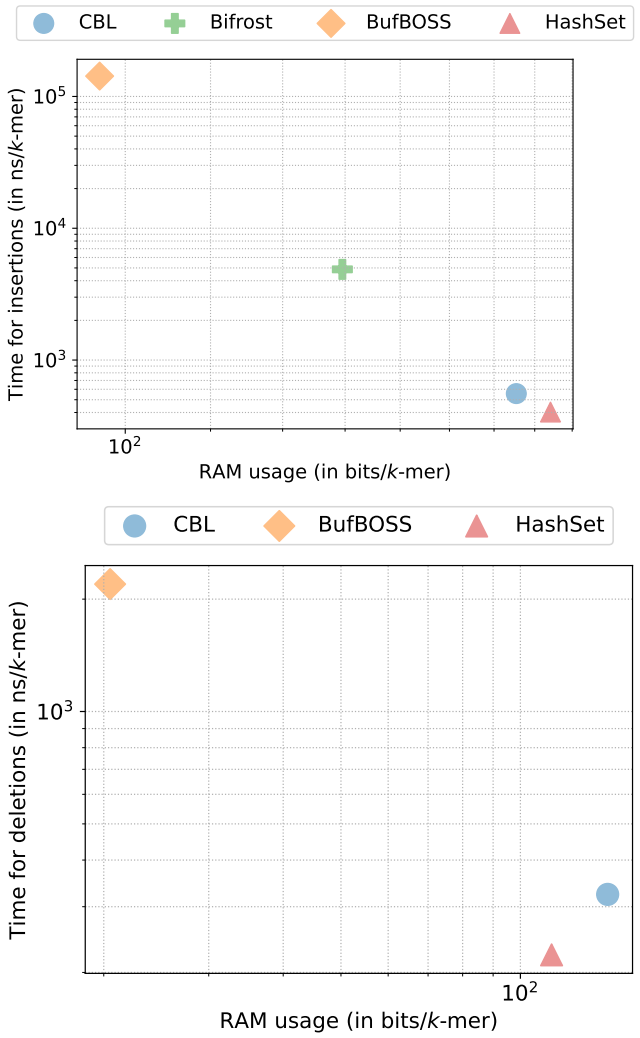

Fig. 17. Time/memory trade-off of various tools when performing positive queries (up) and negative queries (down)

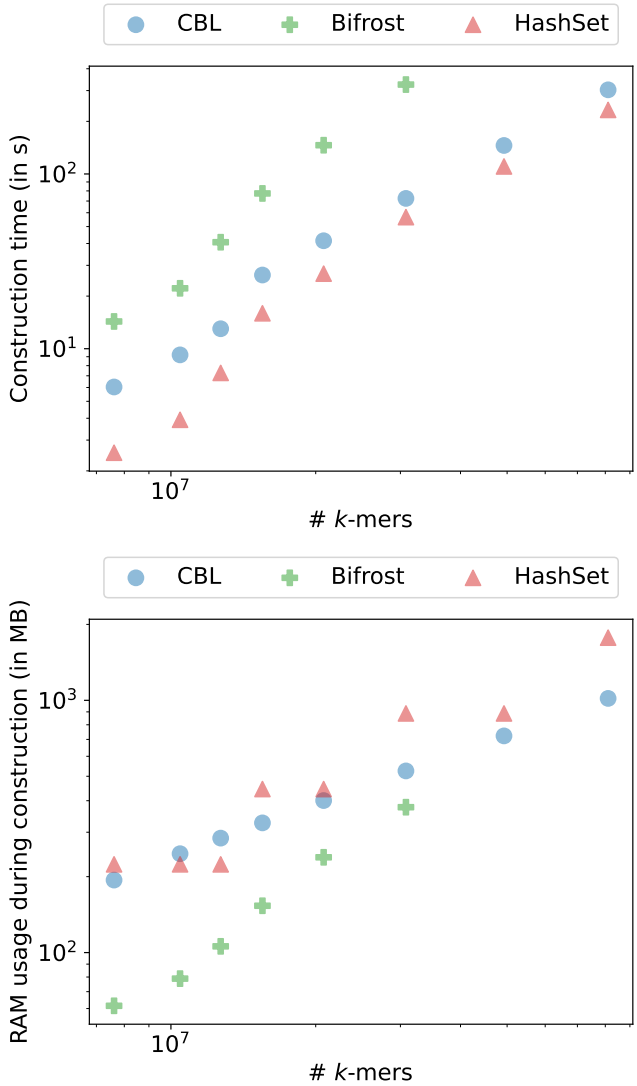

Fig. 18. Time and RAM used when constructing various indexes on a HiFi long read dataset for $k = 31$ and $p = 28$ bits.