bioRxiv, 2025, 1–12
doi: 10.1101/2024.10.30.621029

FroM Superstring to Indexing: a space-efficient index for unconstrained k-mer sets using the Masked Burrows-Wheeler Transform (MBWT)

Ondřej Sladký,[1,2,*] Pavel Veselý[2,†] and Karel Břinda[3,†]

[1]ETH Zurich, Switzerland, [2]Charles University, Czechia and [3]Inria, Irisa, Univ. Rennes, France

[*]Corresponding author: ondra.sladky@gmail.com [†]Joint supervision

Abstract

The exponential growth of DNA sequencing data calls for efficient solutions for storing and querying large-scale k-mer sets. While recent indexing approaches use spectrum-preserving string sets (SPSS), full-text indexes, or hashing, they often impose structural constraints or demand extensive parameter tuning, limiting their usability across different datasets and data types. Here, we propose FMSI, a minimally parametrized, highly space-efficient membership index and compressed dictionary for arbitrary k-mer sets. FMSI combines approximated shortest superstrings with the Masked Burrows-Wheeler Transform (MBWT). Unlike traditional methods, FMSI operates without predefined assumptions on k-mer overlap patterns but exploits them when available. We demonstrate that FMSI offers superior memory efficiency over established indexes such as SSHash, SBWT, and CBL, with space savings up to 2–3× compared to the second-best competitor, depending on the dataset, k-mer size, sampling, and genomic complexity, while supporting fast membership and dictionary queries. Overall, this work establishes superstring-based indexing as a highly general, flexible, and scalable approach for genomic data, with direct applications in pangenomics, metagenomics, and large-scale genomic databases.

Key words: k-mer sets, shortest superstring, Burrows-Wheeler Transform, membership queries, compressed dictionary

Introduction

The exponential growth of DNA sequencing data calls for novel sublinear approaches for their compression and search [46, 74]. To address the growing complexity of data, modern bioinformatics methods are increasingly relying on k-mer tokenization for data storage and analysis. This approach allows for a unified representation of various types of genomic data, such as sequencing reads, transcripts, assemblies, or species pan-genomes, as well as data reduction via techniques such as subsampling [76] and sketching [56]. Notable applications of k-mer-based methods include large-scale data search [7, 4, 13, 37], metagenomic classification [76, 14], infectious disease diagnostics [8, 12], and transcript abundance quantification [9, 58]. As the resulting sets can encompass hundreds of billions of k-mers [37], their efficient storage and associated membership queries have become critical questions in sequence bioinformatics [49].

State-of-the-art data structures for k-mer sets increasingly use textual representations [6, 51, 1, 59]. Originally, specialized methods for exact k-mer set representation drew ideas from information theory [22], which, however, provided unsatisfactory results for the practically encountered k-mer sets. Subsequent approaches have focused on improving space requirements either through inexactness or by exploiting k-mer non-independence [16]. Particularly advantageous has been the observation that genomic k-mer sets are typically generated by a small number of long strings, a property referred to as the *spectrum-like property (SLP)* [17]. If such strings are inferred from a given k-mer set, they can serve as its exact representation.

We refer to such textual representations as *(r)SPSS, (repetitive) Spectrum Preserving String Sets*. The first were *unitigs*, which correspond to the non-branching paths of the associated vertex-centric de Bruijn graphs. They were followed by *simplitigs/SPSS* [11, 10, 62, 61], which permit arbitrary disjoint paths in the de Bruijn graphs, and *matchtigs/rSPSS* [67, 66] which compact along any paths in the de Bruijn graphs, thereby further improving compression under SLP. When combined with full-text indexes or minimal perfect hash functions, (r)SPSS are in the core of modern k-mer data structures such as SShash [59, 60].

Masked superstrings have allowed for further k-mer set compressibility [72], modeling k-mer sets using overlap graphs instead of de Bruijn graphs. Here, k-mer sets are represented using their approximately shortest superstring and an associated mask that encodes the introduced "false positive" k-mers. Masked superstrings

© The Author 2025. Published by Oxford University Press. All rights reserved. For permissions, please e-mail: journals.permissions@oup.com

unify all (r)SPSS representations and further generalize them by allowing overlaps shorter than $k-1$, resulting in further compression, especially for pangenomic or subsampled datasets [72].

However, the state-of-the-art k-mer indexing methods are still based on (r)SPSS representations and thus cannot make use of these benefits. Although masked superstrings have been shown to be well compressible for a variety of k-mer sets [72], their appealing compressibility properties and the potential for simplicity of not having to store a multitude of strings thus remain unexploited by the current k-mer data structures. Moreover, to achieve high performance, current methods typically rely on a manual calibration of one or multiple parameters for individual datasets, thus creating additional level of complexity for the user.

Here, we introduce FMSI, a parameter-free method for space-efficient indexing of arbitrary k-mer sets without any constraints on their structure. FMSI combines the ideas of representing k-mer sets using approximated shortest masked superstrings, with an indexing technique that we term the Masked Burrows-Wheeler Transform (MBWT), resulting in a space-efficient data structure for membership and dictionary queries and support for streaming. In fact, we construct a minimum perfect hash function for the input k-mers with $O(k)$ time evaluation both for the function and its inverse. We show that $2+o(1)$ bits of query memory per indexed k-mer suffice for data sets with SLP, with space requirements growing linearly with the size of the superstring. We implement our index in a program called FMSI (`https://github.com/ondrejsladky/fmsi`) and demonstrate its superior space efficiency compared to the state-of-the-art k-mer set indexing methods on a variety of genomic datasets, including microbial pangenomes, the human genome, Illumina raw reads, and the subsampled Kraken database [76].

Related Work

A substantial body of research has focused on data structures for individual k-mer sets and their collections; we refer to recent surveys [17, 49, 48, 47]. One strategy for individual k-mer sets involves combining the (r)SPSS textual representations [18, 11, 62, 67] with efficient full-text indexes such as the FM-index [30]; this includes, e.g., BWA [44] (as used in [11]), dbgFM [19] or ProPhex [65, 64]. Hashing techniques have proven effective in the design of k-mer data structures. Notably, Bifrost [35] uses hash tables to index colored de Bruijn graphs to represent k-mer set collections. Other prominent data structures, such as BBHash [45], BLight [51], and SSHash [59], employ minimal perfect hash functions and serve as a base for constructing indexes such as FDBG [24], REINDEER [50], Fulgor [28], and pufferfish2 [27]. Another strategy uses BWT-based the BOSS data structure [6], which has been implemented as an index across multiple k-mer sets in VARI [54], VARI-merge [53], and Metagraph [37]. Inspired by BOSS, the Spectral Burrows-Wheeler Transform (SBWT) [1] has been proposed as a compact representation of the de Bruijn graph, alongside various slightly varying indexing method. Themisto [2] further builds upon SBWT to provide an index for collections of k-mer sets. Space reduction can also be achieved by sacrificing the lossless property and allowing for a certain low-probability error. For instance, the counting quotient filter, used for k-mers in Squeakr [57], was later extended to an efficient index called dynamic Mantis [3]. Another line of work employs variants of the Bloom filter to further reduce space requirements [7, 4, 34, 40]. More recently, Invertible Bloom Lookup Tables combined with subsampling have been used to estimate the Jaccard similarity coefficient [68]. The CBL structure [52] is a dynamic and exact k-mer index based on a combination of smallest cyclic rotations of k-mers with sparse bit-vector encodings from [22].

Methods

Problem formulation

Let us consider the nucleotide alphabet ACGT. For a fixed $k > 0$, *k-mers* are strings of length k over this alphabet. We assume the *bidirectional model*, where a k-mer and its reverse complement are considered equal, unless the *unidirectional model* is explicitly indicated. We are interested in designing data structures with the following indexing functionality for a k-mer set K.

P1: **Membership.** Index with a single $O(k)$-time operation:
- ▷ Member(Q): determine whether a given k-mer Q is in K.

P2: **Dictionary.** Index with two $O(k)$-time operations for a bijective mapping between k-mers $Q \in K$ and the set of hashes $H = \{0, \dots, |K|-1\}$:
- ▷ Lookup(Q): returns the hash $h \in H$ of Q, or -1 if $Q \notin K$.
- ▷ Access(h): returns the k-mer Q corresponding to $h \in H$.

P3: **Streaming**. Perform Member/Lookup for Q in $O(1)$ if overlapping k-mer Q' whose $k-1$-long prefix is a suffix of Q has just been queried and both $Q, Q' \in K$.

Note that our formulation of the dictionary index (P2) corresponds to evaluating a *minimal perfect hash function* (MPHF) with rejection and its inverse.

We have three additional requirements on an ideal data structure. First, our motivation is to minimize space requirements while maintaining competitive query speed. This corresponds to analyses in memory-fixed budgets, such as in real-time diagnostics on portable devices from streamed nanopore data [12]. Together with MPHF evaluation, space efficiency yields a so-called *compressed dictionary* [59]. Second, no particular input structure of the k-mer sets should be assumed, such as the existence of many $(k-1)$-long overlaps; however, when such as a structure is available, the method should be able to exploit it. Third, support for efficient streaming queries should be an extension of the data structure for isolated queries, while using a small amount of additional space and retaining the efficiency on isolated queries as well.

Two core ingredients of FMSI

Our approach is based on BWT-based indexing of a particular type of k-mer superstrings, based on the desired queries. The given k-mer set is first represented via an approximately shortest masked superstrings of the k-mers and then indexed using a technique based on the Burrows-Wheeler Transform, which is complemented with supporting rank and select data structures into an exact index termed FMSI. FMSI shares similarities but is different from existing BWT-based approaches such as dbgFM [19], BWA [44, 42], FM-index [30], and SBWT [1] (**Fig. 1**c).

1. Approximately shortest masked superstrings of k-mers
A *masked superstring* representation [72] of a given k-mer set K consists of two parts: a superstring S of the k-mers, i.e., a single string containing all the k-mers in K as substrings, and a binary mask M determining which of the k-mers appearing in S are being represented. A k-mer is considered represented by a masked superstring (S, M) if at least one of its occurrences in S has the corresponding mask symbol set to 1, and the set of represented

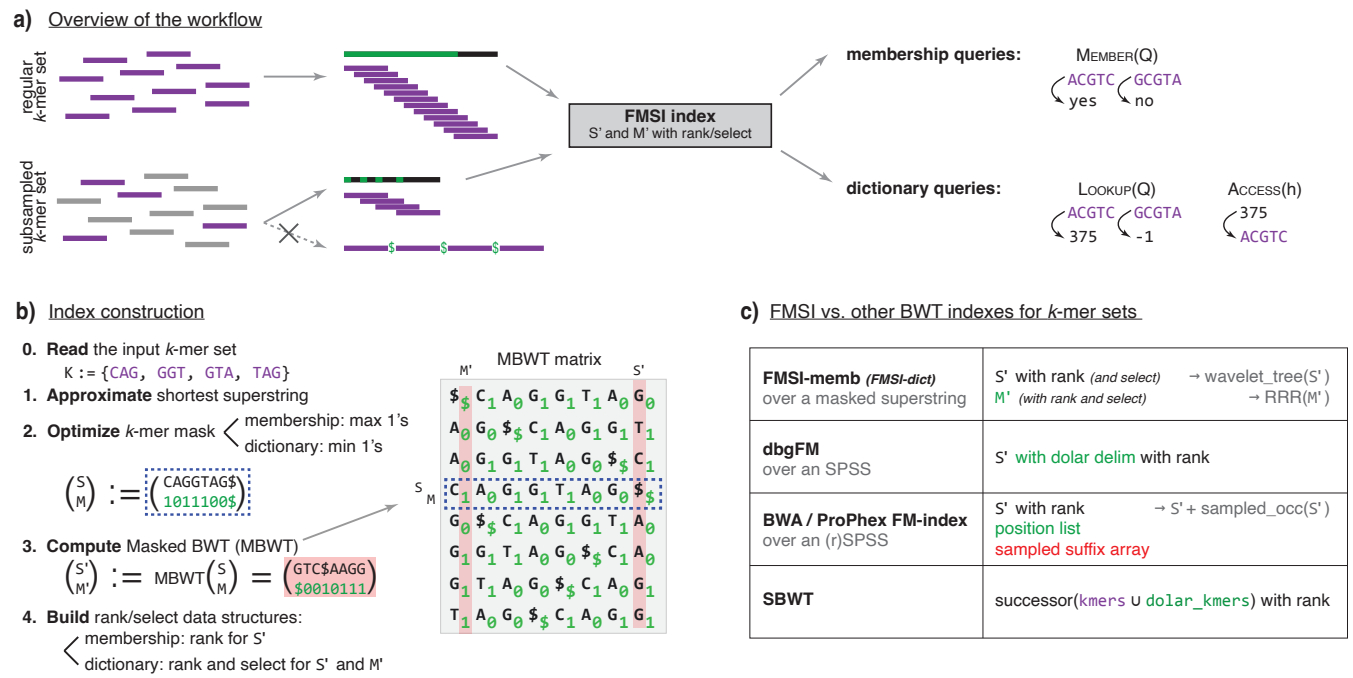

Fig. 1: **Overview of the FMSI k-mer set indexing method. a)** Overview of procesing either regular or subsampled k-mer set by computing a masked superstring building the FMSI index, which can be used for membership or dictionary queries. **b)** For a given k-mer set, shortest superstring is approximated using a greedy algorithm, and mask subsequently optimized to have either maximum or minimum number of ones, based on the desired functionality (P1 or P2). The MBWT of the masked superstring is then computed by an appropriately adjusted BWT algorithm and finally, FMSI index is complete by creating supporting rank and select data structures. The resulting FMSI index consists of S' and M' which we implement via wavelet trees [32] and RRR [63]. **c)** The FMSI index shares similarities, but is distinct from the dbgFM [19], BWA [44], ProPhex [65], FM-index [30], and SBWT [1].

Algorithm 1 MBWT computation.

```
function ComputeMBWT(S, M)
    M = RotateToLeft(M, 1)
    Z = zip(S, M)
    Z' = BWT(Z, compare = x → x.first)
    S', M' = unzip(Z')
    return S', M'
```

	P1: Memb.	P2: Dict.	P3: Stream.	Scalable	Dynamic
BWA [44]	✓ [11]	✗	ProPhex [65]	✓	✗
dbgFM [19, 62]	✓	ABySS [70]	✗	✗	✗
SSHash [59]	✓	✓	✓	✓	✗
CBL [52]	✓	✗	✓	✓	✓
SBWT [1]	✓	✗	✓	✓	✗
FMSI	✓	✓	✓	✓	✓ [73]

Table 1. Comparison of the functionality of FMSI with other state-of-the-art k-mer indexes for single sets.

k-mers should be equal to K. Masked superstrings generalize all (r)SPSS representations and provide more general framework with better compression properties [72].

We will consider masked superstrings obtained as greedily approximated shortest superstrings [72] and, based on the desired functionality, with masks optimized to have the maximum or the minimum number of ones, denoted max-1's and min-1's (Steps 1, 2 in **Fig. 1**b). Such a superstring approximation can be computed by the global greedy algorithm in $O(k \cdot |K|)$ time, as implemented in KmerCamel in [72]. The masked superstring output by KmerCamel already has the min-1's mask. The mask optimization step is thus only needed for max-1's and takes time linear in the superstring size. The max-1's property enables to determine the presence/absence of a k-mer based on its arbitrary occurrence, and thus better optimizing the time for membership queries. However, for dictionary queries, we require min-1's mask to obtain a MPHF.

2. Masked Burrows-Wheeler Transform

To index the resulting masked superstring, we developed a technique called *Masked Burrows Wheeler Transform*, an extension of the traditional BWT [15] to support masks in the suffix array (SA) space (**Fig. 1**b).

Definition 1 (MBWT) Let (S, M) be a masked superstring and let S' be the BWT image of S. The *SA-transformed mask* is a string M' of the same length as M such that for all i, $M'[i] = M[j]$ where j is the starting position of the i-th lexicographically smallest suffix of S. We call the pair (S', M') the *Masked Burrows-Wheeler Transform* (MBWT) of (S, M).

Algorithm 2 k-mer search in FMSI

```
function SingleSearch(Q)                          ▷ Backward search
    i = 0, j = |S'|
    for all c ∈ reversed(Q) do
        i, j = |S|_<c + S'.rank(c, i), |S|_<c + S'.rank(c, j)
        if i = j then return 0, 0
    return i, j
function StreamingSearch(Q, i, j)
    if i = j then return i, j = SingleSearch(Q)
    i = klcp.select(0, klcp.rank(0, i))
    j = klcp.select(0, 1 + klcp.rank(0, j − 1))
    c = Q[0]
    return i, j = |S|_<c + S'.rank(c, i), |S|_<c + S'.rank(c, j)
```

Algorithm 3 Query operations (given SA range $[i, j)$ or hash h)

```
function Member(i, j)                             ▷ requires max-1's mask
    return i ≠ j & M'[i] = 1
function Lookup(i, j)                             ▷ requires min-1's mask
    r_i = M'.rank(1, i), r_j = M'.rank(1, j)
    if r_i = r_j then return −1 else return r_i
function Access(h)
    Q = "", i = M'.select(h)
    for k steps do
        c = max{c ∈ Σ : |S|_≤c < i}
        i = S'.select(c, i − |S|_<c)
        Q = Q + S'[i]
    return Q
```

The SA-transformed mask can be computed alongside the BWT with just a small tweak: a rotation of the mask by one position to the left, followed by "gluing" the mask symbols to the superstring symbols, and computing the BWT of the superstring. Crucially, the lexicographical order is based solely on the superstring characters and the BWT computation is not be affected by the glued mask bits. Therefore, MBWT can be constructed with any method for BWT computation that allows alphabets with eight symbols while considering pairs of different symbols be lexicographically equivalent. We formalize the MBWT computation in Alg 1, and Lemma 1 guarantees the correctness. We note that the MBWT may be applied to any masked string, and thus may find applications out of the scope of this work.

Lemma 1 (SA-transformed mask construction) *Let (S, M) be a masked superstring, M' the SA-transformed mask, and $BWT_S^{-1}(i)$ the original coordinates in S of the i-th character from S'. Then $M'[i] = M[BWT_S^{-1}(i) + 1 \mod |S|]$.*

Proof Consider the i-th lexicographically smallest rotation of S. The last character of this rotation has position $BWT_S^{-1}(i)$ in S and since the first character cyclically succeeds it in S, it has coordinate $BWT_S^{-1}(i) + 1 \mod |S|$ in S. Therefore, $M'[i]$ must be $M[BWT_S^{-1}(i) + 1 \mod |S|]$. □

The FMSI membership and dictionary data structures

The FMSI data structure indexes particular forms of masked superstrings (S, M), to allow for efficient membership or dictionary queries, possibly complemented with support for fast streaming queries. Besides (S', M'), the MBWT image of (S, M), we compute the rank data structure [36] for S', and for dictionary queries additionally also select [21] for S' and rank and select for M' (**Fig. 1**b). M' can be stored in a compressed form with a random-access support such as using RRR [63] or Elias-Fano [26, 29]. We compare the differences between FMSI and other BWT-based indexing structures for k-mer sets in **Fig. 1**c.

Definition 2 (FMSI) Given a k-mer set K, a corresponding masked superstring (S, M) and its associated MBWT image (M', S'), we define the following data structures:

- **FMSI-memb** = (S' with rank, M'), assuming that M *maximizes* the number of 1's in the mask, for membership queries (P1).
- **FMSI-dict** = (S' with rank and select, M' with rank and select), assuming that M *minimizes* the number of 1's in the mask, for dictionary queries (P2).

FMSI index construction

The MBWT of a masked superstring can be computed in linear time in the superstring length using any standard algorithm for BWT construction for an alphabet with 8 symbols and possibility of adjusting the comparisons (Alg. 1 and Lemma **1**). The bottleneck is the computation of the approximately shortest masked superstring, which can be computed in time $O(k \cdot |K|)$ using the Greedy algorithm [75], even in the bidirectional model [72, 71]. All the other steps, including maximizing or minimizing the number of 1's in the mask [72], can be done in time linear with respect to the superstring length, which is bounded by $O(k \cdot |K|)$ but typically significantly shorter [72]. Therefore, we obtain:

Lemma 2 *Given a list of k-mers, both FMSI-memb and FMSI-dict data structures can be constructed from a set of k-mers K in time linear in the size of the k-mers given as a list, i.e., $O(k \cdot |K|)$.*

We now analyze the space requirements of FMSI, using the RRR compression [63] for the SA-transformed mask. To make the bound precise, let $|M|_c$ be the total number of occurrences of a character c in M.

Lemma 3 (FMSI storage requirements) *Both FMSI-memb and FMSI-dict with the SA-transformed mask compressed using RRR [63] require $2|S| + \log \binom{|S|}{|M|_1} + o(|S|)$ bits. If $|S| = |K| + o(|K|)$, the index requires $2 + o(1)$ bits per distinct canonical k-mer.*

Proof S' stored as a plain bit-vector-based wavelet tree [32] requires $2 + o(1)$ bits per character, including the rank and select support. M' stored using RRR [63] requires $\log \binom{|S|}{|M|_0} = \log \binom{|S|}{|M|_1}$ bits [63]. Both rank and select for a RRR-compressed vector M' occupy only sublinear space [63]. Last, suppose that $|S| = |K| + \Delta$ with $\Delta \in o(|K|)$. Letting $\Delta = \frac{|K|}{f(|K|)}$ for $f(|K|) \to \infty$ as $|K| \to \infty$, then the space complexity of M' is $\log \binom{|S|}{|M|_1} \leq \log \binom{|K|+\Delta}{\Delta} \leq \Delta \log(\frac{2e|K|}{\Delta}) = |K| \frac{\log(2e \cdot f(|K|))}{f(|K|)} = o(|K|)$ and thus, the index requires $2 + o(1)$ bits per distinct k-mer. □

FMSI membership and dictionary queries

FMSI can act as two slightly different data structures, a membership index and a dictionary, with membership queries being slightly

more space- and time-efficient. In both cases, the search starts with the standard backward search as used in the FM index, which results in a suffix array (SA) interval corresponding to occurrences of the query k-mer in the superstrings (Alg. 2). The backward search uses the values of $|S|_{<c}$ for characters c, equal to the total number of occurrences of all characters $c' < c$ in S, which can be computed during the FMSI construction in constant space. Given the SA occurrence interval $[i, j)$, we resolve the query using the SA-transformed mask M' (Algorithm **3**). In the bidirectional model, if the forward k-mer is not found in the superstring, we also query its reverse complement (RC).

Membership queries. In FMSI-memb (i.e., with a max-1's mask), the k-mer's presence is determined from the first bit of the SA-transformed mask in the range $[i, j)$ from backward search is set to one, since all the bits in the range have the same value from maximality. Membership may be tested with other masks as well, but at the expense of additional time overhead due to invocations of two rank operations on the mask to determine if there is a 1 in the range from backwards search; namely, we infer the presence by checking $M'.\text{rank}(1, i) < M'.\text{rank}(1, j)$.

Dictionary queries. FMSI-dict supports the Lookup operation, using min-1's masks, with a space and time overhead over the membership queries. Namely, we require additional rank and select data structures and and two additional invocations of M'.rank to compute $r_i = M'.\text{rank}(1, i)$ and $r_j = M'.\text{rank}(1, j)$. Note if $r_i = r_j$, the k-mer is not in K as there is no 1 in the range $[i, j)$ in M', and that due to the min-1's property of the mask, r_i is the number of represented k-mers that appear before the queried k-mer with respect to the SA coordinates and can serve as a unique minimal hash.

To evaluate the Access(h) operation, which is the inverse of Lookup, we use $M'.\text{select}(h)$ to get the position i of the h-th one in M', which corresponds to the BWT matrix row starting with the desired k-mer and obtain the desired k-mer by traversing the mapping between the first and last column of the BWT matrix for k steps (Alg. 3). For each step, we obtain the character in the first column from the values of $|S|_{<c}$ and then use select on S' to get its position in S'.

For other masks, the dictionary operations are possible, but with the loss of surjectivity of Lookup. Namely, with any mask, we would hash k-mers into $\{0, \dots, |S| - 1\}$ instead of $\{0, \dots, |K| - 1\}$. Access can then be defined with an additional 'undefined' value for hashes that do not correspond to k-mers from K.

Streaming acceleration

In many practical scenarious, k-mers are queries in batches of overlapping k-mers from a single string, such as a sequencing read. To support faster processing of positive streamed queries, we leverage the kLCP array [64], namely, the bit-vector of the same length as S such that the i-th bit is set to 1 if and only if the longest common prefix between the i-th and $(i + 1)$-th suffixes of S in the lexicographic order has length at least $k - 1$.

Unidirectional streaming via kLCP. Without taking reverse complements (RCs) into account, we traverse the query sequence in the reverse direction. Starting from the last k-mer, after we have queried a *present* k-mer, we obtain the range for its prefix of size $k - 1$ by extending the range in the SA coordinates in both directions to the first 0 in the kLCP array using two rank and select operations. To get the range for the next k-mer, we perform an update of the range with its first character, i.e., one step of the backward search, which only costs $O(1)$ time in total. Only when the current queried k-mer does not appear in the superstring, we perform the backward search for the next k-mer in time $O(k)$.

Streaming bidirectionality via saturating strand counters. An additional challenge in the bidirectional model is that k-mers in the superstring may be from either stand while extending the SA range for the next queried k-mer assumes that the strand remains the same. To deal with it, we use a new technique of saturating counters, which store whether forward or reverse strand of the query was more often found in the superstring. In a nutshell, we slice the query into blocks and predict whether each block should be first queried forwards or backwards. On the level of individual k-mers, whenever a k-mer is found on the forward strand, we increment the counter, and decrement otherwise. We do it vice versa for the reverse strand. The value of the counter then predicts which strand should be queried first for each block.

Lemma 4 gives theoretical time guarantees. Apart for slightly better guarantees for the unidirectional model, which vanishes as t grows, the speed up in the bidirectional model assumes that consecutive k-mers do not change strand in the superstring too often, which is satisfied for datasets with relatively few unitigs.

Lemma 4 *For an FMSI index over a superstring S for k-mer set K, and a sequence T containing $t = |T| - k + 1$ many k-mers that belong to K, the following holds:*

- *In the unidirectional model, querying T takes time $O(t + k) = O(|T|)$, or $O(1 + \frac{k}{t})$ per k-mer.*
- *In the bidirectional model, if there are at most $O(t/k^2)$ pairs of consecutive k-mers Q_i, Q_{i+1} in T, such that Q_{i+1} does not appear on the same strand as Q_i in S, then querying k-mers in T takes time $O(t + k\sqrt{t})$, or $O(1 + \frac{k}{\sqrt{t}})$ per k-mer.*

Proof sketch. Querying the last k-mer of T takes $O(k)$ time. In the unidirectional model, since all queries are positive, every other k-mer of T can be queried in $O(1)$ time as explained above.

In the bidirectional model, let $B = \min(\sqrt{t}, k)$. We split T into blocks of size $\Theta(B)$. For each of these blocks, we predict whether the k-mer or its RC are in S using the saturating counter with small constant saturation computed on previous blocks, which predicts to start with the strand with which the previous block ended with. We process the first block in time at most $O(B \cdot k)$. For other blocks, we spend time $O(k)$ on the first k-mer. Then, if we predict the strand correctly and the strand does not change during the block, we spend time $O(k + B)$ per block, and otherwise, we spend time $O(k)$ for each k-mer. The total time spend on blocks in the former case is thus $O((t/B) \cdot (B + k)) = O(t + k \cdot \sqrt{t})$. The latter can happen if consecutive k-mers switch strands either inside the block, in the previous block, or between blocks which may cause an incorrect prediction. However, such switching happens at most $O(t/k^2)$ times in total and hence, the total time complexity is bounded by $O(B \cdot k + t + k \cdot \sqrt{t} + (t/k^2) \cdot B \cdot k)$, which is always bounded by $O(t + k\sqrt{t})$. □

Implementation in the FMSI program

We implemented the FMSI index in a tool called FMSI, an abbreviation of 'FroM Superstring to Indexing'. FMSI is developed in C++, provided on Github under the MIT licence (`https://github.com/OndrejSladky/fmsi`), deposited on Zenodo (`https:`

dataset	k	31-mers	CBL		SSHash		SBWT-fast		SBWT-small		FMSI	
sc2-pg	15, 17, ... 43	12.7 M	153.8–654.1	*46.6–78.7*	14.0–25.0	*10.7–19.3*	22.2–56.0	*9.4–10.3*	16.0–50.7	*3.2*–5.0*	5.5*–9.1*	*3.4–3.7**
sp-pg	15, 17, ... 43	9.62 M	150.3–300.9	*47.5–88.9*	8.2–22.6	*5.5–18.0*	30.0–45.9	*9.2–9.8*	24.5–41.4	*3.6–5.1*	6.1*–7.5*	*2.8*–3.4**
ec-pg-hq	15, 17, ... 43	551 M	34.3–173.0	*20.7–80.9*	6.7–22.7	*6.6–22.6*	9.0–9.7	*8.5–8.6*	3.4–7.8	*3.1–6.6*	3.1*–3.5*	*2.8*–3.3**
ec-pg-all	15, 23, ... 31	1.32 G	25.5–69.0	*18.7–48.9*	7.6–22.0	*7.5–21.9*	8.8–9.2	*8.5–8.6*	3.1–7.8	*2.9–7.1*	2.9*–3.0*	*2.7*–2.8**
mtg-ilm	15, 17, ... 31	355 M	37.1–65.5	*21.0–50.9*	7.8–22.7	*7.7–22.6*	9.5–10.8	*8.6–10.1*	3.7–7.3	*3.1–6.1*	3.0*–3.7*	*2.8*–3.4**
rna-ilm	15, 23, ... 31	686 M	39.6–130.4	*20.5–48.8*	14.9–22.8	*14.9–22.6*	9.7–14.4	*8.7–14.0*	4.8–7.8	*4.4–6.7*	3.3*–6.2*	*3.0*–5.9**
hg-ilm	15, 23, ... 31	3.56 G	25.1–341.1	*18.6–48.5*	10.8–20.6	*10.8–20.6*	9.2–10.5	*8.5–10.4*	3.5–8.5	*3.4–7.8*	2.9*–4.1*	*2.6*–3.9**
hg-t2t	15, 17, ... 43	2.51 G	21.1–597.0	*19.0–73.1*	5.9–24.2	*5.9–24.2*	8.6–9.2	*8.5–8.5*	2.4*–8.0	*2.3–7.3*	2.4*–3.0*	*2.2*–2.9**
Overall			21.1–654.1	*18.6–88.9*	5.9–25.0	*5.5–24.2*	8.6–56.0	*8.5–14.0*	2.4*–50.7	*2.3–7.8*	2.4*–9.1*	*2.2*–5.9**

Table 2. Space requirements of membership query data structures. The table displays the ranges of 'bits per canonical k-mer' for individual indexes across all benchmark datasets evaluated across the indicated k-mer values. The space was measured in memory (via GNU time; larger numbers) and on disk (smaller numbers in italics). CBL with hg-t2t could be evaluated only for $k \leq 39$. The best values for each dataset are marked with *.

`//doi.org/10.5281/zenodo.13905020`), and distributed via Bioconda [33]. The version of FMSI used in this paper is 0.4.0. FMSI can calculate an index for user-provided masked superstring (S, M) for an arbitrary k, and then query it with Membership or Lookup queries, possibly streamed. The input masked superstring can be computed by KmerCamel, or alternatively inferred from any (r)SPSS representation.

The index in FMSI consists of three components: S' as a BWT image of S (stored via a wavelet tree over plain bit-vectors), M' as the corresponding SA-transformed mask (an RRR-compressed bit-vector), and an optional kLCP array (a plain bit-vector), all based on the sdsl-lite library [31]. MBWT computation proceeds via suffix array construction using QSufSort [39].

For single-k-mer queries, if a k-mer is not found in the superstring (Algs 2 and 3), its RC is also queried (Algs 2 and 3), starting from a predicted strand (using a counter with saturation 7). FMSI uses two additional counters, to accumulate and predict only if the previous k-mer was found in the forward or reverse direction respectively. Streaming (Alg. 2) additionally simplifies the range extension: the kLCP array is traversed in both directions until 0s are found, avoiding the additional rank data structures. Bidirectionality is supported via splitting each queried sequence of length ℓ into parts of size $\lfloor 2\sqrt{\ell} \rfloor$. The initial query strand of each part is predicted from the saturating counters of the preceding parts, with only the first strand queried entirely and the other one only for the not-yet-found k-mers.

Experimental Evaluation

Evaluation protocol. FMSI was evaluated against three state-of-the-art k-mer sets indexes: SBWT [1] (commit b795178), SSHash [59] (commit d90ad37), and CBL [52] (commit 328bcc6), with configurations advised by the authors. DbgFM [19] was excluded due to its limitations. SSHash was compiled separately for $k \leq 31$ and $k > 31$, and always used with the minimizer size $\min\{\lceil \log_4 |K| \rceil + 1, k - 2\}$. CBL was compiled with SIMD support and 28 prefix bits for each k-value. SBWT was tested in two modes: *small* (with rrr-split, index without RCs) and *fast* (plain-matrix, with RCs), both variants with and without streaming support.

Eight datasets were used for benchmarking (**Tab. 2**): SARS-CoV-2 pan-genome (**sc2-pg**; n=14M; GISAID [69]), *S. pneumoniae* pan-genome (**sp-pg**; n=616; [25]), *E.coli* 661k pan-genomes [5] with (**ec-pg-hq**; n=86K) and without (**ec-pg-all**; n=89K) quality filtering, human T2T assembly (**hg-t2t**; chm13v2.0 [55]), metagenomic reads from a human tongue microbiome (**mtg-ilm**; SRX023459),

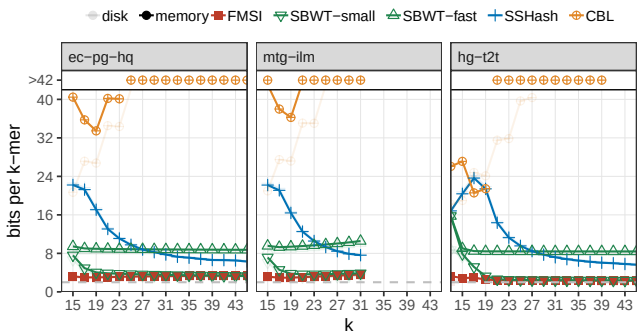

Fig. 2: **Index size in memory (and on disk) as a function of k for selected datasets**. The evaluation of mtg-ilm is limited by $k = 32$ as this was the length for the initial k-mer filtering. For CBL, hg-t2t indexes could not be constructed beyond $k = 39$ due to its excessive memory requirements (>251 GiB RAM).

RNA-seq reads (**rna-ilm**; SRX348811), and a human reads (**hg-ilm**; SRX016231). All eight datasets were initially preprocessed by GGCAT [23], possibly followed by KmerCamel [72] to obtain greedy masked superstrings for FMSI. Illumina (***-ilm**) datasets were also cleaned via k-mer filtering ($k = 32$, min-freq 2).

Each method was assessed across individual datasets and for various k-mer sizes, focusing on memory per distinct k-mer (on disk and in RAM) and time per k-mer query for various configurations (positive/negative queries, with/without streaming, and with/without 10% reference subsampling). Benchmarks were performed on an AMD EPYC 7302 (3 GHz) server with 251 GB RAM and SSD storage, using a single core for each program. Detailed protocols and pre-processed datasets are provided in Supplementary Materials and on Zenodo (`https://doi.org/10.5281/zenodo.13997397`).

Index construction. We first evaluated time and memory requirements of index construction, including MS/SPSS computation from unitigs/assembly if required; detailed results are provided in Supplementary Materials. Focusing on hg-t2t with $k = 31$, CBL, SSHash, and SBWT-fast indexes were constructed in 35–45 minutes and 100 GB (SSHash only 20 GB) and SBWT-small needed only 23 minutes and 50 GB, whereas FMSI index required 187 minutes and 120 GB, since computing the greedy superstring took 170 minutes (FMSI executed on the masked superstring used 16 minutes and 45 GB); we nevertheless have preliminary evidence that superstring computation can be substantially optimized.

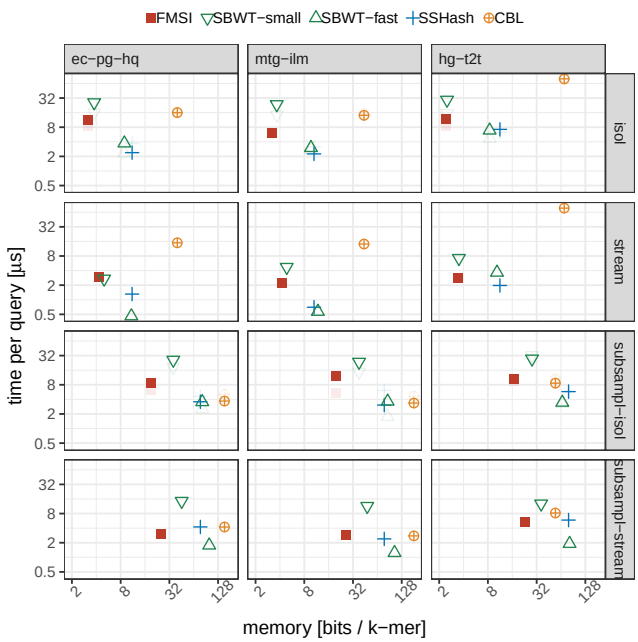

Fig. 3: **Time/memory efficiency of k-mer membership queries for selected datasets with** $k = 23$. The plot compares time per query and memory per k-mer for isolated and streamed queries on the full dataset and 10% subsampled k-mer set. Negative queries are plotted in light.

Results on regular k-mer sets. To evaluate membership queries' memory and time for FMSI-memb, we first focused on "standard" k-mer sets, which are easier to index as they satisfy the SLP. We found that FMSI consistently provided the best memory consumption – across all the scenarios, datasets, and k-mer sizes tested (**Tab. 2** and **Fig. 2** and **3**), requiring only 2.2–3.9 bits per k-mer on disk and sligthly more in memory (except for rna-ilm). On hg-t2t, FMSI used ≤ 2.4 bits of memory per k-mer for $k \geq 21$. At the same time, the speed of FMSI was competitive, typically in the mid range of all the methods. In most cases, FMSI was 2–3× faster than the second best space-efficient tool, SBWT-small, which had similar memory requirements on regular k-mer sets. SBWT-fast and SSHash were 2–4× faster but typically required 2–3× more memory than FMSI; for SSHash, the ratio was decreasing with increasing k beyond 31. Lastly, CBL needed by 1–2 orders of magnitude larger memory and was also significantly slower. The use of kLCP improved streamed queries 2–4×, at the cost of an additional bit per superstring character for FMSI and similarly for SBWT; SSHash and CBL do not require more memory for streaming support. The comparison to competing methods on streamed queries remains roughly the same as for isolated queries. Importantly, unlike all other tools, FMSI provided consistent performance regardless of the value of k.

Membership for subsampled k-mer sets. As genomics increasingly uses sparsification techniques, such as subsampling and sketching, and FMSI directly supports this, we further evaluated the application of our approach in such a setup. We tested indexes on 10% randomly sampled distinct k-mers from selected datasets (**Fig. 3**, bottom half). This resulted in substantial increase of memory per k-mer of all test tools (except for CBL on hg-t2t).

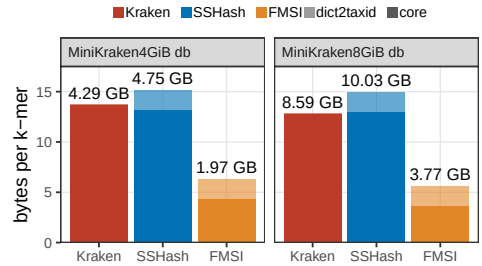

Fig. 4: **MiniKraken DB k-mer LCA indexing.** Comparison of Kraken, FMSI, and SSHash using the subsampled Kraken databases (MiniKraken v20171019, $k = 31$). FMSI and SSHash include an additional two bytes per k-mer to associate hashes with the k-mers' lowest common ancestors in the taxonomic tree.

FMSI typically used at most half the space of the second best tool, SBWT-small, while also being 2× faster. FMSI was ≈ 4× more space-efficient than SBWT-fast and SSHash, while being 2–3× slower on isolated queries. FMSI was particularly efficient on streaming queries, similarly fast as SSHash while using about 4× less space, achieved thanks to the "k-mer interpolation" capabilities of superstrings.

MiniKraken dictionary indexing. Finally, we sought to evaluate whether FMSI may be used to reduce the memory requirements of methods based on sampled k-mer dictionaries, such as Kraken [76] with the MiniKraken databases (4 GiB and 8 GiB). To evaluate this, we built FMSI-dict and SSHash 31-mer dictionaries for the MiniKraken 31-mer sets and compared them to the original MiniKraken indexes (**Fig. 4**). We found that, for both MiniKraken databases, SSHash moderately increased memory requirements, while FMSI provided more than a twofold memory improvement.

Conclusions

k-mer-based methods are central in modern bioinformatics and the efficiency of k-mer set indexing is thus critical for reducing computational resources. Here, we have focused on minimizing the space requirements and provided FMSI, a k-mer-set indexing method that consistently outperforms in space all other state-of-the-art methods across all operational ranges, while maintaining a competitive query speed. This was possible thanks to a combination of ideas from the world of superstrings and BWT indexing, thus building upon our previous work on masked superstring [72] and extending it via the the Masked BWT.

The scalability of the FMSI method to big datasets is currently limited by two bottlenecks, both readily addressable: k-mer superstring computation and MBWT construction. First, the current superstring construction protocol based on KmerCamel proceeds by greedy merges of sequences, starting from the original k-mers. However, instead, such computation may start from unitigs or even arbitrary simplitigs, for which scalable solutions are available [20, 11, 62, 38, 23]. Our preliminary data suggest that, on the human genome, this may reduce superstring computation time by a factor of approximately 5×. Second, modern BWT index construction algorithms are already applicable at the terabyte scale [43] and can be embedded into FMSI indexing.

Despite the long history and the rich palette of available methods, k-mer-set indexing remains an exciting problem with substantial room for improvement. This is especially true when nontraditional k-mer sets come into play, such as those arising from subsampling or sketching, or those accumulating extensive genetic

polymorphisms or technological artifacts, such as seen in computational pan-genomics. Superstrings then provide an elegant solution to bypass the limitations of traditional methods, representing a major weapon in the algorithmic arsenal for future generic, dynamic, parameter-free, and space-efficient k-mer-based data structures, with their possible applications reaching far beyond bioinformatics to areas such as large language models.

Author contributions statement. O.S., P.V., K.B. developed the methodology, designed and conducted the experiment(s), analyzed the results, and wrote the manuscript. O.S. developed the software.

Acknowledgments. This research was supported by the French National Research Agency (ANR) under Grant ANR-24-CE45-1226 for the REALL project (KB), by Czech Science Foundation project 22-22997S (OS, PV), by ERC-CZ project LL2406 of the Ministry of Education of Czech Republic (PV), and by Center for Foundations of Modern Computer Science (Charles Univ. project UNCE 24/SCI/008, PV). We are grateful to Camille Marchet, Igor Martayan, Giulio Pibiri, Jarno Alanko, and Paul Medvedev for their constructive comments.

References

1. J. N. Alanko, S. J. Puglisi, and J. Vuohtoniemi. Small searchable κ-spectra via subset rank queries on the spectral Burrows-Wheeler transform. In *ACDA '23*. SIAM, 2023.
2. J. N. Alanko, J. Vuohtoniemi, T. Mäklin, and S. J. Puglisi. Themisto: a scalable colored k-mer index for sensitive pseudoalignment against hundreds of thousands of bacterial genomes. *Bioinformatics*, 39(Supplement_1), 2023.
3. F. Almodaresi, J. Khan, S. Madaminov, M. Ferdman, R. Johnson, P. Pandey, and R. Patro. An incrementally updatable and scalable system for large-scale sequence search using the Bentley-Saxe transformation. *Bioinformatics*, 38(12), 2022.
4. T. Bingmann, P. Bradley, F. Gauger, and Z. Iqbal. COBS: a compact bit-sliced signature index. In *SPIRE'19*. Springer, 2019.
5. G. A. Blackwell, M. Hunt, K. M. Malone, L. Lima, G. Horesh, B. T. F. Alako, N. R. Thomson, and Z. Iqbal. Exploring bacterial diversity via a curated and searchable snapshot of archived dna sequences. *PLOS Biology*, 19(11), 11 2021.
6. A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya. Succinct de Bruijn graphs. In *WABI '12*, LNCS. Springer, 2012.
7. P. Bradley, H. C. Den Bakker, E. P. Rocha, G. McVean, and Z. Iqbal. Ultrafast search of all deposited bacterial and viral genomic data. *Nature Biotechnology*, 37(2), 2019.
8. P. Bradley, N. C. Gordon, T. M. Walker, L. Dunn, S. Heys, B. Huang, S. Earle, L. J. Pankhurst, L. Anson, M. De Cesare, et al. Rapid antibiotic-resistance predictions from genome sequence data for staphylococcus aureus and mycobacterium tuberculosis. *Nature Communications*, 6(1), 2015.
9. N. L. Bray, H. Pimentel, P. Melsted, and L. Pachter. Near-optimal probabilistic RNA-seq quantification. *Nature Biotechnology*, 34(5), 2016.
10. K. Břinda. Novel computational techniques for mapping and classification of Next-Generation Sequencing data. PhD thesis, Université Paris-Est, 2016.
11. K. Břinda, M. Baym, and G. Kucherov. Simplitigs as an efficient and scalable representation of de Bruijn graphs. *Genome Biology*, 22(96), 2021.
12. K. Břinda, A. Callendrello, K. C. Ma, D. R. MacFadden, T. Charalampous, R. S. Lee, L. Cowley, C. B. Wadsworth, Y. H. Grad, G. Kucherov, et al. Rapid inference of antibiotic resistance and susceptibility by genomic neighbour typing. *Nature Microbiology*, 5(3), 2020.
13. K. Břinda, L. Lima, S. Pignotti, N. Quinones-Olvera, K. Salikhov, R. Chikhi, G. Kucherov, Z. Iqbal, and M. Baym. Efficient and robust search of microbial genomes via phylogenetic compression. *bioRxiv*, 2023.04.15.536996, 2023.
14. K. Břinda, K. Salikhov, S. Pignotti, and G. Kucherov. Prophyle 0.3.1.0. *Zenodo*, 5281, 2017.
15. M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
16. R. Chikhi. K-mer data structures in sequence bioinformatics. HDR thesis, Institut Pasteur Ecole Doctorale "EDITE", 2021.
17. R. Chikhi, J. Holub, and P. Medvedev. Data structures to represent a set of k-long DNA sequences. *ACM Computing Surveys*, 54(1), 2022.
18. R. Chikhi, A. Limasset, S. Jackman, J. T. Simpson, and P. Medvedev. On the representation of de Bruijn graphs. In *RECOMB*, Cham, 2014. Springer International Publishing.
19. R. Chikhi, A. Limasset, S. Jackman, J. T. Simpson, and P. Medvedev. On the representation of de bruijn graphs. *Journal of Computational Biology*, 22(5), May 2015.
20. R. Chikhi, A. Limasset, and P. Medvedev. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12), 2016.
21. D. Clark. *Compact PAT Trees*. Phd thesis, UWSpace, 1997.
22. T. C. Conway and A. J. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4), 2011.
23. A. Cracco and A. I. Tomescu. Extremely fast construction and querying of compacted and colored de bruijn graphs with ggcat. *Genome Research*, May 2023.
24. V. G. Crawford, A. Kuhnle, C. Boucher, R. Chikhi, and T. Gagie. Practical dynamic de Bruijn graphs. *Bioinformatics*, 34(24), 2018.
25. N. J. Croucher, J. A. Finkelstein, S. I. Pelton, J. Parkhill, S. D. Bentley, M. Lipsitch, and W. P. Hanage. Population genomic datasets describing the post-vaccine evolutionary epidemiology of streptococcus pneumoniae. *Scientific Data*, 2, 2015.
26. P. Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2), 1974.
27. J. Fan, J. Khan, G. E. Pibiri, and R. Patro. Spectrum preserving tilings enable sparse and modular reference indexing. In *RECOMB '23*, volume 13976 of *LNCS*. Springer, 2023.
28. J. Fan, J. Khan, N. P. Singh, G. E. Pibiri, and R. Patro. Fulgor: a fast and compact k-mer index for large-scale matching and color queries. *Algorithms for Molecular Biology*, 19(1), 2024.
29. R. M. Fano. *On the number of bits required to implement an associative memory*. MIT, Project MAC, 1971.
30. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS '00*, SFCS-00. IEEE, 2000.
31. S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *SEA '14*, 2014.
32. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *SODA '03*. SIAM, 2003.
33. B. Grüning, R. Dale, A. Sjödin, B. A. Chapman, J. Rowe, C. H. Tomkins-Tinch, R. Valieris, J. Köster, and B. Team. Bioconda: sustainable and comprehensive software distribution for the life sciences. *Nature methods*, 15(7), 2018.

34. G. Gupta, M. Yan, B. Coleman, B. Kille, R. A. L. Elworth, T. Medini, T. Treangen, and A. Shrivastava. Fast processing and querying of 170TB of genomics data via a Repeated And Merged BloOm filter (RAMBO). In *SIGMOD '21*. ACM, 2021.
35. G. Holley and P. Melsted. Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs. *Genome Biology*, 21(1), 2020.
36. G. J. Jacobson. *Succinct static data structures*. PhD thesis, Carnegie Mellon University, USA, 1988. AAI8918056.
37. M. Karasikov, H. Mustafa, D. Danciu, M. Zimmermann, C. Barber, G. Rätsch, and A. Kahles. Indexing all life's known biological sequences. *bioRxiv*, 2020.10.01.322164, 2024.
38. J. Khan, M. Kokot, S. Deorowicz, and R. Patro. Scalable, ultra-fast, and low-memory construction of compacted de bruijn graphs with cuttlefish 2. *Genome Biology*, 23(1), Sept. 2022.
39. N. J. Larsson and K. Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3), 2007.
40. T. Lemane, P. Medvedev, R. Chikhi, and P. Peterlongo. kmtricks: efficient and flexible construction of Bloom filters for large sequencing data collections. *Bioinformatics Advances*, 2(1), 04 2022.
41. H. Li. wgsim. `https://github.com/lh3/wgsim`, 2011.
42. H. Li. Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. *Bioinformatics*, 28(14), 2012.
43. H. Li. BWT construction and search at the terabase scale. *Bioinformatics*, 40(12), Nov. 2024.
44. H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14), 2009.
45. A. Limasset, G. Rizk, R. Chikhi, and P. Peterlongo. Fast and scalable minimal perfect hashing for massive key sets. In *SEA '17*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.
46. P.-R. Loh, M. Baym, and B. Berger. Compressive genomics. *Nature Biotechnology*, 30(7), July 2012.
47. C. Marchet. Advances in colored k-mer sets: essentials for the curious. *arXiv [q-bioGN]*, 2409.05214, 2024.
48. C. Marchet. Advances in practical k-mer sets: essentials for the curious. *arXiv [q-bioGN]*, 2409.05210, 2024.
49. C. Marchet, C. Boucher, S. J. Puglisi, P. Medvedev, M. Salson, and R. Chikhi. Data structures based on k-mers for querying large collections of sequencing data sets. *Genome Research*, 31(1), 2021.
50. C. Marchet, Z. Iqbal, D. Gautheret, M. Salson, and R. Chikhi. REINDEER: efficient indexing of k-mer presence and abundance in sequencing datasets. *Bioinformatics*, 36(Supplement-1), 2020.
51. C. Marchet, M. Kerbiriou, and A. Limasset. Blight: efficient exact associative structure for k-mers. *Bioinformatics*, 37(18), 2021.
52. I. Martayan, B. Cazaux, A. Limasset, and C. Marchet. Conway–Bromage–Lyndon (CBL): an exact, dynamic representation of k-mer sets. *Bioinformatics*, 40:i48–i57, 2024.
53. M. D. Muggli, B. Alipanahi, and C. Boucher. Building large updatable colored de Bruijn graphs via merging. *Bioinformatics*, 35(14), 2019.
54. M. D. Muggli, A. Bowe, N. R. Noyes, P. S. Morley, K. E. Belk, R. Raymond, T. Gagie, S. J. Puglisi, and C. Boucher. Succinct colored de Bruijn graphs. *Bioinformatics*, 33(20), 2017.
55. S. Nurk, S. Koren, A. Rhie, M. Rautiainen, A. V. Bzikadze, A. Mikheenko, M. R. Vollger, N. Altemose, L. Uralsky, A. Gershman, et al. The complete sequence of a human genome. *Science*, 376(6588), 2022.
56. B. D. Ondov, T. J. Treangen, P. Melsted, A. B. Mallonee, N. H. Bergman, S. Koren, and A. M. Phillippy. Mash: fast genome and metagenome distance estimation using minhash. *Genome Biology*, 17(1), June 2016.
57. P. Pandey, M. A. Bender, R. Johnson, and R. Patro. Squeakr: an exact and approximate k-mer counting system. *Bioinformormatics*, 34(4), 2018.
58. R. Patro, G. Duggal, M. I. Love, R. A. Irizarry, and C. Kingsford. Salmon provides fast and bias-aware quantification of transcript expression. *Nature Methods*, 14(4), 2017.
59. G. E. Pibiri. Sparse and skew hashing of K-mers. *Bioinformatics*, 38(Supplement_1), 2022.
60. G. E. Pibiri. On weighted k-mer dictionaries. *Algorithms for Molecular Biology*, 18(1), 2023.
61. A. Rahman. Compression algorithms for de Bruijn graphs and uncovering hidden assembly artifacts. PhD thesis, The Pennsylvania State University, 2023.
62. A. Rahman and P. Medevedev. Representation of k-mer sets using spectrum-preserving string sets. *Journal of Computational Biology*, 28(4), 2021. PMID: 33290137.
63. R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4), nov 2007.
64. K. Salikhov. Efficient algorithms and data structures for indexing dna sequence data. PhD thesis, Université Paris-Est, 2017.
65. K. Salikhov, K. Břinda, S. Pignotti, and G. Kucherov. ProPhex. `https://github.com/prophyle/prophex`, 2018.
66. S. Schmidt. Unitigs are not enough: the advantages of superunitig-based algorithms in bioinformatics. PhD thesis, University of Helsinki, 2023.
67. S. Schmidt, S. Khan, J. N. Alanko, G. E. Pibiri, and A. I. Tomescu. Matchtigs: minimum plain text representation of k-mer sets. *Genome Biology*, 24(1), 2023.
68. Y. Shibuya, D. Belazzougui, and G. Kucherov. Efficient Reconciliation of Genomic Datasets of High Similarity. In *WABI '22*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.
69. Y. Shu and J. McCauley. Gisaid: Global initiative on sharing all influenza data–from vision to reality. *Eurosurveillance*, 22(13), 2017.
70. J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and Birol. Abyss: A parallel assembler for short read sequence data. *Genome Research*, 19(6), Feb. 2009.
71. O. Sladký. Masked superstrings for efficient k-mer set representation and indexing. Bachelor's thesis, Charles University, 2024.
72. O. Sladký, P. Veselý, and K. Břinda. Masked superstrings as a unified framework for textual k-mer set representations. *bioRxiv*, 2023.02.01.526717, 2023.
73. O. Sladký, P. Veselý, and K. Břinda. Towards efficient k-mer set operations via function-assigned masked superstrings. *bioRxiv*, 2024.03.06.583483, 2024.
74. Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson. Big data: Astronomical or genomical? *PLoS Biology*, 13(7), 2015.
75. E. Ukkonen. A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 5(3), 1990.
76. D. E. Wood and S. L. Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome Biology*, 15(3), 2014.

Supplementary material

Experimental evaluation

All scripts, data tables, and additional information are available in the supplementary repository of our paper on `https://github.com/OndrejSladky/fmsi-supplement`. The experimental evaluation was done in the following five steps.

Step 1: Download of the source data. The *SARS-CoV-2* pan-genome **sc2-pg** was downloaded from GISAID `https://gisaid.org/` (access upon registration, version 2023/01/23). The *E. coli* pan-genome was obtained from the phylogenically compressed 661k collection [5] as provided on `https://doi.org/10.5281/zenodo.4602622` [13]; all the genomes from the batches starting by '`escherichia_coli__`' were extracted and used subsequently. Dataset **ec-pg-all** is the whole *E. coli* pan-genome, without quality filtering, while for **ec-pg-hq** we applied high-quality filtering. The *S. pneumoniae* pan-genome was downloaded from the RASE DB *S. pneumoniae* (`https://github.com/c2-d2/rase-db-spneumoniae-sparc/`) Metagenomic sample SRS063932 (Illumina raw reads) of human microbiome with accession SRX023459, denoted **mtg-ilm**, was download from `https://www.hmpdacc.org/hmp/HMASM/`; we converted the fastq files into FASTA files by '`seqtk seq -A -C`'. Human RNA-seq Illumina raw reads with accession SRX348811, **rna-ilm**, and the human genome Illumina raw reads with accession SRX016231, **hg-ilm**, were downloaded using the prefetch tool from the SRA toolkit and then converted into the FASTA format by '`fastq-dump --split-3 --fasta`'. The Human genome assembly chm13v2.0, **hg-t2t**, was downloaded from `https://s3-us-west-2.amazonaws.com/human-pangenomics/T2T/CHM13/assemblies/analysis_set/chm13v2.0.fa.gz`. The MiniKraken datasets (4GiB and 8GiB) were downloaded from `https://ccb.jhu.edu/software/kraken/`

Step 2: k-mer set preparation. The input files for the pan-genomes, mtg-ilm, and hg-ilm reads were converted to unitigs by GGCAT v1.1.0 [23] by '`ggcat build -k {kmer-size} -m 200 -j 5 -s {min-freq} -o {preprocessed} {input_FASTA}`'. We used $k = 128$ and {min-freq}=1 for pan-genomes and $k = 32$ and {min-freq}=2 for mtg-ilm and HG Illu. No preprocessing was done for the HG assembly chm13v2.0 (hg-t2t). For MiniKraken datasets (4GiB and 8GiB), we dumped the 31-mers using Jellyfish 1.1.12. The obtained files with the k-mer sets were deposited on `https://zenodo.org/records/14722244`.

Step 3: Representation tailoring. Before indexing with SShash [59] and FMSI, we used GGCAT [23] (v1.1.1) and KmerCamel🐫 [72] (v1.0.2), respectively, for preprocessing before indexing. The following commands were used (with k matching the k passed to the programs for indexing).

- For SShash, we computed eulertigs, which is the shortest SPSS representation: '`ggcat build -k {kmer-size} --eulertigs --min-multiplicity 1 --memory 200 --threads-count 1 -o {eulertigs} {preprocessed}`'.
- For FMSI with the best available MS: '`kmercamel -p {preprocessed} -o {ms-min1s} -k {kmer-size} -C`' and then, to optimize membership queries, '`kmercamel optimize -p {ms-max1s} -o {ms} -k {kmer-size} -C`'

Step 4: Generating k-mer queries. To generate queries, we simulated Illumina-like reads by WgSim [41] (v0.3.1-r13) with the following command:'`wgsim -1{query_length} -d0 -S42 -s31 -e0 -r{mut_rate} -R0 -N{num_seqs} {wgsim_input} {queries} /dev/null`', where '`query_length`' was equal to k for isolated queries and to 300 for streaming queries, '`mut_rate`' was 0 for positive (isolated and streaming) queries and 0.1 for negative queries, '`num_seqs`' was set so that there are 10^6 k-mer queries overall (up to 300 less for streaming queries), and '`wgsim_input`' was dataset and query-type dependent; namely, for generating negative isolated queries, we used chromosome 1A of *T. aestivum* assembly GCF_018294505.1, downloaded from NCBI. We generated positive isolated and streaming queries from the original dataset or a reference genome in case of pangenomes (details provided in the supplementary repository). For input to the version of SBWT which does not store reverse complementary k-mers, we additionally duplicated every line of all query files and computed the reverse complements of the duplicated lines using '`seqtk seq -r`'.

Step 5: Construction and querying of the indexes We describe the commands used for the construction of indexes and running the queries. Time and memory for index construction are summarized in **Tab. 3**, including MS/SPSS computation of Step 3. The measured performance of the indexes is provided in **Tab. 2** and **Figs. 2-4**. Further information and experimental results can be found at `https://github.com/OndrejSladky/fmsi-supplement`.

- **FMSI** (`https://github.com/OndrejSladky/fmsi/`, v0.4.0, commit '`04c4580`'). FMSIhas two variants, depending on the desired query type:
  - **Variant for membership queries:** index construction by '`fmsi index -k {kmer-size} {ms-max1s}`' and queries using '`fmsi query -k {kmer-size} -O -q {queries} {ms-max1s}`', additionally with '`-S`' in case of streaming queries.
  - **Variant for dictionary queries:** index construction by '`fmsi index -k {kmer-size} {ms-max1s}`' and lookup queries using '`fmsi lookup -k {kmer-size} -q {queries} {ms-min1s}`', additionally with '`-S`' in case of streaming queries.

- **SBWT** [1] (`https://github.com/algbio/SBWT`, commit '`b795178`'). We compiled SBWT separately for $k \leq 32$ and for $k \in (32, 64]$. The specific parameters were tuned based on the results provided in [1]. As the time-efficient variant, we use the default plain-matrix variant, with adding all reverse complements to the index. As the space-efficient variant, we use the rrr-split variant without reverse complements.
  - **Fast variant:** index construction using
    '`sbwt build -k {kmer-size} -m 120 -t 1 --add-reverse-complements \\`
    `-i {prefix} -o {index-path}`', additionally with '`--no-streaming-support`' for isolated queries, and queries executed by '`sbwt search -i {index-path} -q {queries} -o /dev/null`'.

–**Space-efficient variant:** index construction using
'`sbwt build --variant rrr-split -k {kmer-size} -m 120 -t 1 -i {prefix} -o {index-path}`', additionally with `--no-streaming-support` for isolated queries, and queries executed by '`sbwt search -i {index-path} -q {queries_with_RCs} -o /dev/null`', always querying a k-mer and its reverse complement (RC).

•**SSHash** [59, 60] (`https://github.com/jermp/sshash`, commit '`d90ad37`'). A dictionary based on minimal perfect hashing of k-mers. We compiled SSHash separately for $k \leq 31$ and for $k \in [32, 63]$. Indexes were computed using '`sshash build -k {kmer-size} -m {M} -s {seed}` ', where we set the minimizer length to '`M`' $= \min\{\lceil \log_4 \# \ k\text{-mers} \rceil + 1, k - 2\}$ and used a random seed. Whenever the index construction failed, SSHash was executed again with a different random seed.
–**Single variant:** index construction using '`sshash build -i {prefix} -k {kmer-size} -m {M} -o {index-path} -s {seed}`', where `prefix` is the input file with an SPSS of the k-mer set, and queries using '`sshash query -i {index-path} -q {queries}`'

•**CBL** [52] (`https://github.com/imartayan/CBL`, commit '`328bcc6`'), a very recent method based on smallest cyclic rotations of k-mers. The index was computed on canonical k-mers, to handle reverse complements, that is, we ran '`cbl build -c`'. CBL was compiled for every value of k separately, namely, we used '`RUSTFLAGS="-C target-cpu=native" K={kmer-size} PREFIX_BITS={pref_bits} \\ cargo +nightly build ---release ---examples ---target-dir target.k_{kmer-size}`'
where the pref_bits were set to 28 for $k \geq 15$, to 24 for $k = 13$, and to 20 for $k = 11$. We then used the following commands:
–**Single variant:** index construction done using '`cbl build -c -o {index-path} {prefix}`', where `prefix` is the preprocessed dataset, and queries using '`cbl query {index-path} {queries}`'

Supplementary tables

Dataset	Subsampl. rate	Tool	MS/SPSS tool	MS/SPSS time [h]	MS/SPSS mem [GB]	Index constr. time [h]	Index constr. mem [GB]
ec-pg-hq	10%	CBL	–	–	–	0.020	1.017
ec-pg-hq	10%	SSHash	GGCAT (eulertigs)	0.088	36.736	0.011	2.342
ec-pg-hq	10%	SBWT-fast	–	–	–	1.456	58.664
ec-pg-hq	10%	SBWT-small	–	–	–	0.722	30.071
ec-pg-hq	10%	FMSI	KmerCamel's global	0.197	2.674	0.046	8.079
ec-pg-hq	100%	CBL	–	–	–	0.324	5.803
ec-pg-hq	100%	SSHash	GGCAT (eulertigs)	0.248	14.647	0.033	2.949
ec-pg-hq	100%	SBWT-fast	–	–	–	0.352	120.053
ec-pg-hq	100%	SBWT-small	–	–	–	0.165	78.885
ec-pg-hq	100%	FMSI	KmerCamel's global	0.806	26.152	0.066	11.444
hg-t2t	100%	CBL	–	–	–	0.605	101.094
hg-t2t	100%	SSHash	GGCAT (eulertigs)	0.474	19.219	0.242	10.901
hg-t2t	100%	SBWT-fast	–	–	–	0.703	99.811
hg-t2t	100%	SBWT-small	–	–	–	0.378	49.919
hg-t2t	100%	FMSI	KmerCamel's global	2.846	118.014	0.263	44.245
hg-t2t	100%	*FMSI**	*(estimated)*	≈ 0.6	< 50	0.263	44.245
mtg-ilm	10%	CBL	–	–	–	0.016	0.796
mtg-ilm	10%	SSHash	GGCAT (eulertigs)	0.057	23.722	0.009	1.628
mtg-ilm	10%	SBWT-fast	–	–	–	1.196	35.739
mtg-ilm	10%	SBWT-small	–	–	–	0.478	18.690
mtg-ilm	10%	FMSI	KmerCamel's global	0.147	1.684	0.035	5.470
mtg-ilm	100%	CBL	–	–	–	0.055	3.703
mtg-ilm	100%	SSHash	GGCAT (eulertigs)	0.080	6.677	0.020	2.171
mtg-ilm	100%	SBWT-fast	–	–	–	0.185	11.728
mtg-ilm	100%	SBWT-small	–	–	–	0.122	5.883
mtg-ilm	100%	FMSI	KmerCamel's global	0.448	16.529	0.045	7.571

Table 3. Comparison of superstring and index construction time and memory requirements for selected datastes. MS/SPSS computation means either computing a masked superstring (MS, for FMSI) or SPSS (for SSHash). For hg-t2t, the row *FMSI** contains an estimate for a more time efficient superstring computation by the global greedy algorithm executed on SPSS.