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

Ondřej Sladký, Pavel Veselý, Karel Břinda

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[1], and Karel Břinda[3]

[1] Computer Science Institute, Charles University, Malostranské nám., 118 00 Praha, Czechia
[2] ETH Zurich, Rämistrasse, 8092 Zurich, Switzerland
[3] Inria, Irisa, Univ. Rennes, Campus de Beaulieu, 35042 Rennes, France
ondra.sladky@gmail.com, vesely@iuuk.mff.cuni.cz, karel.brinda@inria.fr

Abstract. The exponential growth of DNA sequencing data limits the searchable proportion of the data. In this context, tokenization of genomic data via their k-merization provides a path towards efficient algorithms for their compression and search. However, indexing even single k-mer sets still remains a significant bioinformatics challenge, especially if k-mer sets are sketched or subsampled. Here, we develop the FMSI index, a space-efficient data structure for unconstrained k-mer sets, based on approximated shortest superstrings and the Masked Burrows Wheeler Transform (MBWT), an adaptation of the BWT for masked superstrings. We implement this in a program called FMSI, and via extensive evaluations using prokaryotic pan-genomes, we show FMSI substantially improves space efficiency compared to the state of the art, while maintaining a competitive query time. Overall, our work demonstrates that superstring indexing is a highly general, parameter-free approach for modern k-mer sets, without imposing any constraints on their structure.

Keywords: k-mers, k-mer sets, data structures, superstring, Burrows-Wheeler Transform, FM-index, masked superstrings

1 Introduction

The exponential growth of DNA sequencing data calls for novel sublinear approaches for their compression and search [44,70]. To address the increasing 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 [72] and sketching [53]. Notable applications of k-mer-based methods include large-scale data search [8,4,14,36], metagenomic classification [72,15], infectious disease diagnostics [9,13], and transcript abundance quantification [10,55]. As the resulting sets can encompass hundreds of billions of k-mers [36], their efficient storage and associated membership queries have become critical questions in sequence bioinformatics [47].

State-of-the-art data structures for k-mer sets increasingly use textual representations [7,49,1,56]. Originally, specialized methods for exact k-mer set representation drew ideas from information theory [21], 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 [18]. 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)* [19]. 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* [12,11,59,58], which permit arbitrary disjoint paths in the de Bruijn graphs, and *Matchtigs/rSPSS* [64,63] 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 [56,57].

A more recent improvement of k-mer set compressibility has been achieved via modeling k-mer sets using overlap graphs instead of de Bruijn graphs, resulting in the concept of so-called masked superstrings [68]. Here, k-mer sets are represented using their approximated shortest superstring and an associated mask which identifies the introduced "false positive" k-mers. Masked superstrings unify all (r)SPSS representations and further generalize them by allowing overlaps shorter than $k-1$, resulting in further compression especially for pan-genomic or subsampled datasets [68].

However, the current state-of-the-art approaches 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 [68], 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, these methods often rely on the use of several parameters which need to be adjusted manually, creating additional complexity for the user.

Here, we introduce FMSI, a novel parameter-free method for space-efficient indexing of arbitrary k-mer sets without any constrains on their structure. FMSI combines the ideas of representating k-mer sets using approximated shortest masked superstrings (**Section 2.1**), with an indexing technique that we term the Masked Burrows-Wheeler Transform (MBWT) (**Section 2.2**). We prove that $2 + o(1)$ bits of query memory per indexed k-mer suffice for data sets with SLP and also show that its space requirements grow linearly with the size of the k-mer superstring. We provide a linear-time construction algorithm (**Section 2.4**) and show how to answer isolated queries in $O(k)$ time (**Section 2.5**) and positive streamed queries in $O(1)$ time with an additional bit of memory per k-mer (**Section 2.6**), with accommodation for reverse complements using saturating counter-based k-mer strand prediction (**Section 2.7**). Finally, we implement our index in a program called FMSI (**Section 3**, `https://github.com/ondrejsladky/fmsi`) and demonstrate on prokaryotic pan-genomes its superior space efficiency compared to the state-of-the-art k-mer set indexing methods (**Section 4**).

1.1 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 [19,47,46,45].

One strategy for individual k-mer sets involves combining the (r)SPSS textual representations [20,12,59,64] with efficient full-text indexes such as the FM-index [29]; this includes, e.g., BWA [42] (as used in [12]) or ProPhex [62,61]). Hashing techniques have proven effective in the design of k-mer data structures. Notably, Bifrost [34] uses hash tables to index colored de Bruijn graphs to represent k-mer set collections. Other prominent data structures, such as BBHash [43], BLight [49], and SSHash [56], employ minimal perfect hash functions and serve as a base for constructing indexes such as FDBG [22], REINDEER [48], Fulgor [27], and pufferfish2 [26].

Another strategy uses BWT-based BOSS data structure [7], which has been implemented as an index across multiple k-mer sets in VARI [52], VARI-merge [51], and Metagraph [36]. 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 [54] 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 [8,4,33,39]. More recently, Invertible Bloom Lookup Tables combined with subsampling have been used to estimate the Jaccard similarity coefficient [65].

On the dynamic front, the very recent introduction of the Conway-Bromage-Lyndon (CBL) structure [50], based upon the pioneering work of Conway and Bromage [21], combines smallest cyclic rotations of k-mers with sparse bit-vector encodings, to create a dynamic and exact k-mer index.

2 Methods

Let us consider the nucleotide alphabet ACGT. For a fixed $k > 0$, *k-mers* are strings of length k over this alphabet. We assume a *bi-directional model*, where a k-mer and its reverse complement are considered equal, unless the *uni-directional model* is explicitly indicated. We are interested in the following problem:

Problem: For a given k-mer set K, construct two space-efficient data structures to exactly answer, respectively, isolated and streamed k-mer membership queries:

- MEMBER(Q): Determine whether a given k-mer Q is a member of K, and
- MEMBERS(S): Determine which of the k-long substrings of S are contained in K.

Importantly, the problem does not assume any specific structure of the given k-mer set, such as the existance of particular overlaps between the k-mers in K. However, for our benchmarks, we will focus on k-mer sets constructed from bacterial and viral pan-genomes, possibly subjected to sampling techniques. Our motivation is to minimize space requirements while maintaining competitive query speed compared to state-of-the-art methods. The data structure for streamed queries should be an extension of the one for isolated queries, and should be efficient for isolated queries as well.

We will address the problem through a series of step, ultimately leading us to the resulting data structure – the FMSI index.

2.1 Approximated shortest Masked Superstring as a support k-mer-set representation

A *masked superstring* representation [68] of a given k-mer set K consists of two parts: a superstring of the k-mers, i.e., a single string containing all the k-mers in K as substrings, and a binary mask determining which of the k-mers appearing in the superstring are being represented. A k-mer is considered represented by a masked superstring if at least one of its occurrences has the corresponding mask symbol set to 1, and the set of represented k-mers should be equal to K. Masked superstrings generalize all (r)SPSS representations as they always can be converted to a masked superstring with the same length [68]. The opposite is not always true, and therefore, mask superstrings provide more general framework than (r)SPSS [68].

In this paper, we will uniquely consider masked superstrings obtained as greedily approximated shortest superstrings [68], with masks optimized to have the maximum number of ones [68] (Steps 1, 2 in **Figure 1**). Such a superstring approximation can be done by the global greedy algorithm, as implemented in KmerCamel

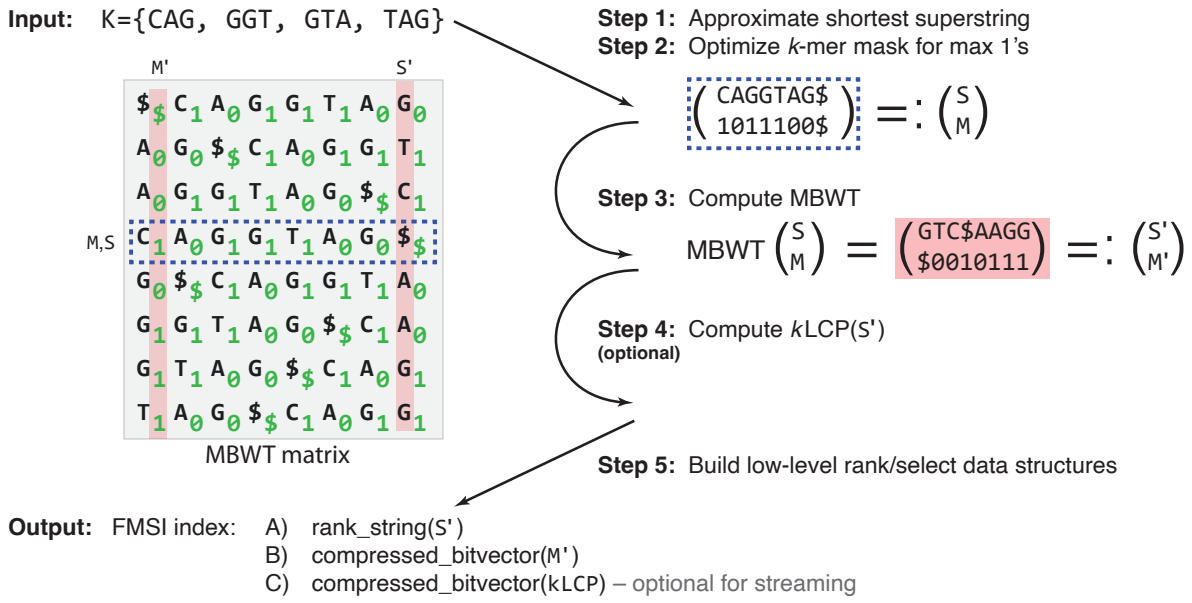

Fig. 1: Computation of an FMSI index using the MBWT. Index construction for an example k-mer set K, including the associated MBWT matrix, with the highlighted computed input superstring S and mask M, alongside the output BWT-transformed [16] superstring S' and the SA-transformed k-mer mask M'. The resulting rank string can be implemented, e.g., via wavelet trees [31] (A), and the compressed bitvectors (B and C) via RRR [60].

in [68], in $O(k|K|)$ time [68]. The mask optimization steps requires linear time, the max-one property guaranties that, since all occurrences of k-mers from K have 1s in the mask, it is possible to determine the presence of a k-mer based on its arbitrary occurrence. This will be important in the subsequent indexing and querying steps.

2.2 Masked Burrows-Wheeler Transform (MBWT) for indexing

The *Burrows-Wheeler Transform* (BWT) [16] of a string S is its permutation such that its i-th character is the last character of the i-th lexicographically smallest rotation of S. Typically, a special \$ symbol, lexicographically smaller than all alphabet letters, is first appended to S, which allows to reverse the BWT.

We design masked Burrows-Wheeler transform (MBWT), a new variant on the BWT, that inputs a string and a binary mask of the same length and outputs the BWT of the string, accompanied by a suitable permutation of the mask, the so-called SA-tranformed mask.

Definition 1 (SA-transformed k-mer mask M'). *Let S be a superstring of a k-mer set and M be a mask with a special \$ symbol appended to both S and M such that \$ is less than all the alphabet letters. Then the SA-transformed mask is a binary string M' of the same length such that for all i: $M'[i] = M[j]$ where j is the starting position of the i-th lexicographically smallest suffix of S.*

Another way how to view the SA-transformed mask is to consider the Burrows-Wheeler matrix of the superstring and attach each mask symbol to its corresponding superstring letter (MBWT matrix in **Figure 1**). The SA-transformed mask is formed by the mask symbols in the first column of the matrix, unlike the Burrows-Wheeler transform itself, which is the last column of the matrix.

Although MBWT cannot be computed directly with an algorithm for BWT, in **Lemma 1** we show that with suitable mask preprocessing, this is possible.

Lemma 1 (SA-transformed mask construction). *Let S be a superstring, M its associated mask, 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 succeeds it in S (with a special case of being the first if the rotation is S), it has coordinate $BWT_S^{-1}(i)+1 \mod |S|$ in S. From **Definition 1**, this implies that $M'[i]$ must be $M[BWT_S^{-1}(i)+1 \mod |S|]$. □

Lemma 1 gives a direct way how to compute the SA-transformed mask alongside the BWT using any method for BWT computation with just a small tweak. We first rotate the mask by one position, then we glue the mask symbols to the superstring symbols, compute the BWT of the superstring while keeping the mask symbols glued. We can view the joint computation of the SA-transformed mask and the BWT of the superstring as applying BWT on pairs of characters, where we order solely based on the superstring characters, disregarding the mask characters in any comparison. As a result, computing MBWT can be done with any algorithm for BWT construction.

2.3 FMSI Index as a novel k-mer-set membership data structure based on MBWT

We design a new index, the FMSI index that leverages the Masked Burrows-Wheeler Transform consisting of the BWT and the SA-transformed mask (**Section 2.2**) computed for masked superstrings representing k-mer sets (**Section 2.1**). The BWT is used to search a k-mer in the suffix-array coordinates, whereas the SA-transformed mask is used to determine which of those k-mers are masked as present in the original masked superstring.

To further accelerate streamed queries, we optionally build the kLCP array [61] which is a binary version of the longest common prefix (LCP) array: the i-th position is set to 1 when the longest common prefix between the lexicographically smallest i-th and $i+1$-th rotations is of length at least $k-1$ characters.

Overall, the FMSI index then consists of three main components (Step 3 in **Figure 1**), the BWT of the superstring, equipped with a rank data structure [35], the SA-transformed mask, and the kLCP array [61] with rank and select data structures for faster positive streamed queries.

We first analyze the storage requirements of the FMSI index (**Lemma 2**), and demonstrate that the FMSI index without the kLCP array requires only $2+o(1)$ bits per k-mer in cases where the superstring length matches the k-mer set size up to a sublinear additive term; we note this can be seen as a mathematical formulation of the SLP property [19].

Lemma 2 (FMSI storage requirements). *Given a k-mer set K, a superstring S of K, and the corresponding mask, the FMSI index without kLCP such that the SA-transformed mask M' is compressed using RRR [60] requires $2|S|+\log\binom{|S|}{O}+o(|S|)$ bits, where O is the total number of occurrences of k-mers from K in S. To additionally store the kLCP array, we require at most $|S|$ additional bits. Moreover, if $|S|=|K|+o(|K|)$, the FMSI index without kLCP requires $2+o(1)$ bits per distinct k-mer.*

Proof. The BWT can be stored as a plain bit-vector-based wavelet tree [31] requiring $2+o(1)$ bits per character, including the rank. M' stored using RRR [60] requires $\log\binom{|S|}{\#0}=\log\binom{|S|}{\#1}$ bits [60], since at each occurrence of $Q\in K$ in S, the mask bit is set to 1 and there are O such occurrences. Both rank and select for RRR-compressed vector M' occupy only sublinear space [60]. kLCP can simply be stored as uncompressed bit-vector, requiring 1 bit per superstring character.

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 $\log\binom{|S|}{O}\leq\log\binom{|K|+\Delta}{\Delta}\leq\Delta\log(\frac{2e|K|}{\Delta})=|K|\frac{\log(2e\cdot f(|K|))}{f(|K|)}=o(|K|)$ and thus, the index without the kLCP array requires $|2|+o(1)$ bits per distinct k-mer. □

2.4 FMSI Index construction in $O(k|K|)$ time

The BWT and the SA-transformed mask can be computed through the Masked BWT (**Section 2.2**) of the masked superstring, which can be done with any algorithm for BWT construction with small modifications described in **Lemma 1**. The kLCP array can then be constructed from the superstring S and its BWT [61, Lemma 12].

We outline the algorithm for constructing the whole FMSI index from a set of k-mers K in **Algorithm 1**; note that the first two steps, superstring computation and mask optimization, are described in [68].

Algorithm 1 FMSI index construction in linear time.

```
function ConstructFMSI(K)
    S = ApproximatedShortestSuperstring(K)          ▷ e.g., global greedy from [68]
    M = MaskMaximizingOnes(S, K)                    ▷ two-pass algorithm from [68]
    M = Rotate(M, 1)                                ▷ rotate the mask by 1 character to the left
    S', M' = BwtWithAttachedMaskSymbols(S, M)       ▷ any BWT algorithm adjusted according to Lemma 1
    klcp = Klcp(BWT, S)                             ▷ algorithm from [61]; optional support for streaming queries
    return S', M', klcp
end function
```

Algorithm 2 k-mer search using FMSI: single k-mer in $O(k)$ vs. streamed overlapping k-mers in $O(1)$

Single k-mer search:

```
function RangeUpdate(i, j, c)
    return cnt(c)+S'.rank(c,i), cnt(c)+S'.rank(c,j)
end function

function BackwardsSearch(Q)
    i = 0, j = |S'|
    for all c ∈ reversed(Q) do
        i, j = RangeUpdate(i, j, c)
        if i = j then
            return −1, −1
        end if
    end for
    return i, j
end function

function InferPresence(i, j)
    return i ≠ j and M'[i]
end function

function QuerySingle(Q)
    i, j = BackwardsSearch(Q)
    return InferPresence(i, j)
end function
```

Streaming k-mer search:

```
function RangeExtend(i, j)
    i = klcp.select(0, klcp.rank(0, i))
    j = klcp.select(0, 1 + klcp.rank(0, j − 1))
    return i, j
end function

function QueryStreaming(T, k)
    present = 0, i = −1, j = −1
    for all pos ∈ reversed(range(len(T) - k + 1)) do
        Q = T[pos : pos + k]
        if i = j then
            i, j = BackwardsSearch(Q)
        else
            i, j = RangeExtend(i, j)
            i, j = RangeUpdate(i, j, Q[0])
        end if
        present = present + InferPresence(i, j)
    end for
    return present
end function
```

All these steps together can be done in time $O(k|K|)$ as given by **Lemma 3**. The bottleneck for this part is the computation of the masked superstring, which requires time $O(k|K|)$ [68], as all the other steps can be done in size linear with respect to the superstring length.

Lemma 3. *The FMSI index can be constructed from a set of k-mers K in time linear to the size of the k-mers given as a list, i.e., $O(k|K|)$.*

Proof. An approximately shortest superstring can be computed in linear time using the Greedy algorithm [71], even in the bi-directional model [68,67]. The associated mask computation that maximizes the number of ones is also linear [68]. We can compute the Masked BWT by first rotating the mask and then computing the BWT, which is known to be computable in linear time, for instance, by first computing the suffix array [37]. Lastly, the kLCP array can be constructed through the LCP array in linear time [37,61]. □

2.5 Efficient FMSI querying of isolated k-mers in $O(k)$ time

The process of answering membership queries on indexed masked superstrings can be split into two steps (see also **Figure 1** for a comparison to the FM-index search). The first step is the same as in the FM-index: We

perform the backwards search for the queried k-mer and obtain the range of occurrences which correspond to prefixes of size k of consecutive rows in the Burrows-Wheeler matrix. To determine presence of a k-mer, it is sufficient to check only a single bit of the SA-transformed mask (**Lemma 4**).

Lemma 4. *Let (S, M) be a masked superstring of a k-mer set K with M maximizing the number of 1s, M' the corresponding SA-tranformed mask, and $[i, j)$ the range of sorted rotations of S starting with a k-mer Q. A k-mer Q is in K if and only if $i \neq j$ and $M'[i] = 1$.*

Proof. If $i = j$, this means that Q does not appear in S thus cannot be in K. Otherwise $M'[i]$ is by **Definition 1** the mask symbol corresponding to an occurrence of Q in S. If $M'[i] = 1$ then by definition of a masked superstring [68], $Q \in K$. Moreover, since M maximizes the number of ones, $M'[i] = 0$ implies that $Q \notin K$. □

We can thus infer the presence of any k-mer based on the value of the first position of the range found by backwards search in the SA-transformed mask. In the bi-directional model, if the forward k-mer is not found in the superstring, we query also with its reverse complement. As a result, querying a single k-mer takes time $O(k)$, both in uni-directional and bi-directional model.

2.6 Accelerated streamed queries via *k*LCP in $O(1)$ time

To support faster processing of positive streamed queries where consecutive queried k-mers share an overlap of $k - 1$ characters, we leverage the kLCP array technique [61]. For simplicity, let us first assume the uni-directional model and postpone the bi-directional model to **Section 2.7**.

We first start with a simplified query procedure which does not take reverse complements into account. Since the query procedure uses the backwards search, we traverse the query sequence in the reverse direction. Starting from the last k-mer, after we have queried a k-mer that is present, we extend the range in the SA coordinates in both directions until we find 0 in the kLCP array, which is done using two rank and two select queries on the kLCP array. This way, we obtain the range in the SA coordinates for the prefix of length $k - 1$, which is the overlap with the next queried k-mer. 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 backwards search, which only costs $O(1)$ operations. Note that we do not need to stop the streamed query when we find a queried k-mer that appears in the superstring but is not represented as its mask bits are set to 0 (such k-mers are called *ghost* in [68]). Only when the current queried k-mer does not appear in the superstring (i.e., the range update returns an empty range), we need to perform the whole backwards search for the next k-mer in time $O(k)$.

Lemma 5. *Given an FMSI index for k-mer set K, querying a sequence T containing $t = |T| - k + 1$ k-mers, such that any k-mer in T is in K takes time $O(t + k) = O(|T|)$, or $O(1 + \frac{k}{t})$ per k-mer, in the uni-directional model.*

Proof. Suppose that we have computed the range of SA-coordinates for a k-mer $P \in K$. We show that querying a k-mer Q such that the $k - 1$-long prefix of P is a suffix of Q takes time $O(1)$. We get the range of the shared $k - 1$-mer by extending the range for P using the kLCP array in constant time [61] and the range of Q is obtained by two queries to the rank over BWT, again in $O(1)$. At the end, we just check whether the range is non-empty and the SA-transformed mask contains a 1 at the initial position of the range. As a consequence, we can query the first query in the stream in $O(k)$ and all the other k-mers using this $O(1)$ update, resulting in $O(t + k)$ time. □

See also **Algorithm 2** for a pseudocode implementation of both streamed and isolated queries.

2.7 Counter-based strand prediction for accelerated bi-directional queries

So far, we have considered the uni-directional model for streamed queries. However, query k-mers can come in two different orientation, forward and reverse, which we do not a priory know. To support both strands in k-mer queries, data structures typically either append reverse complementary strings, compromising space, or query the data structure twice, compromising the query time.

We resolve this trade-off by storing only a single copy of the masked superstring, complemented by a saturating counter. Whenever a k-mer is found on the forward strand, we increment it, 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, which brings a major time benefit for streamed queries.

Lemma 6. *Consider an FMSI index over a superstring of S for a k-mer set K. Let T be a sequence containing $t = |T| - k + 1$ k-mers such that any k-mer in T is in K and additionally assume that on average it happens at most every $\Omega\left(k^2\right)$-th pair of neighbouring 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 it holds that querying k-mers in T takes time $O(t + k\sqrt{t})$, or $O(1 + \frac{k}{\sqrt{t}})$ per k-mer, in the bi-directional model.*

Proof. Let $B = \min(\sqrt{t}, k)$. We split T into blocks of size $\Theta(B)$. For each of those blocks, we predict whether the forward or the reverse complementary k-mers are in S using the saturating counter computed on previous blocks. For a particular block, if we manage to predict the strand correctly and the strand does not change during the block, we spend time $O(1)$ per k-mer as in the proof of **Lemma 5**. Otherwise, we spend time $O(k)$ for each k-mer. This can happen for two reasons. Either, it is the first block, which happens once and so the this amortizes to $O\left(k \cdot \frac{\min(\sqrt{t},k)}{t}\right) = O(\frac{k}{\sqrt{t}})$ per k-mer. Or, it is caused by consecutive k-mers switching strands (either in the block or between blocks which causes incorrect prediction), but since this happens only once per $\Omega(k)$ blocks, the time complexity for this amortizes to $O(1)$. Putting all the cases together, we get time complexity of $O(1 + \frac{k}{\sqrt{t}})$ per k-mer. □

There are two differences in the result of streamed queries in the bi-directional model (**Lemma 6**) compared to the uni-directional model (**Lemma 5**), which is the best that can be hoped for. First, instead of the $\frac{k}{t}$ term, there is the $\frac{k}{\sqrt{t}}$ term. However, once the size of the queried sequence becomes $\Omega(k^2)$, this term vanishes. Second, we impose conditions on the appearances of k-mers in the superstring. Although this is something that in the worst case possible may not hold, for practical scenarios of sets where consecutive k-mer are contained, as otherwise streamed queries are not even possible, the superstring typically contains long segments of the individual genomes, making this assumption realistic.

3 Implementation in the FMSI program

We implemented the FMSI index in a tool called FroM Superstring to Indexing (FMSI), which is developed in C++, available from GitHub under the MIT licence (`https://github.com/OndrejSladky/fmsi`), deposited on Zenodo (`https://doi.org/10.5281/zenodo.13905020`), and distributed via Bioconda [32]. The version of FMSI used in this paper is 0.3.0.

As its input for indexing, FMSI takes a pre-computed masked superstring for a given k (no limitation on the size of k) and computes the corresponding FMSI index; the input masked superstring can be computed, e.g., by KmerCamel, or inferred from an (r)SPSS representation computed by any tool. The computed index can be queried for k-mers from a provided FASTA/FASTQ file, either without kCLP (more memory efficient) or with it (faster for queries with overlapping k-mers).

The index in FMSI consists of three components (**Section 2.3**): the BWT of the superstring, the SA-transformed mask, and optionally the kLCP array. The BWT is stored as a wavelet tree over plain bit-vectors, and contains only the forward strand (i.e., does not add the reverse strand). The SA-transformed mask is stored as a bit-vector, compressed with RRR. Finally, the optional kLCP is stored as a plain bit-vector. All the bit-vectors and the associated ranks for the wavelet tree use implementations from the sdsl-lite library [30] (`https://github.com/simongog/sdsl-lite`). The whole index is constructed directly from a suffix array, obtained using the QSufSort algorithm from [38]; this requires 17 bytes of memory per superstring character.

Isolated queries are computed as described in **Algorithm 2**: first by updating the range with BWT and then a single access to the SA-transformed mask. If a k-mer is not found in the superstring, its reverse complement is queried as well. To predict which of the reverse complementary pair to query first, FMSI uses a saturating counter (**Section 2.7**) with saturation at value 7. In addition to a global counter which accumulates the results of all previous queries, FMSI uses two other counters where the first accumulates results only for k-mers where previously queried k-mer was found as a forward k-mer, and the second if as a reverse k-mer.

Dataset	Nb. of genomes	Cumul. length	#15-mers	#23-mers	#31-mers
S. pneumoniae pan-genome	616	1.27 Gbp	6,634,628	8,260,593	9,627,755
SARS-CoV-2 pan-genome	14,682,066	430 Gbp	4,816,455	8,028,982	11,283,907
E. coli pan-genome	88,749	450 Gbp	369,627,173	1,172,272,910	1,323,794,999

Table 1: Overview of pan-genome datasets used for evaluation, including the original number of genomes, their cumulative length, and numbers of distinct k-mers. The *E. coli* pan-genome was built from all *E.coli* genomes in the 661k collection [5], without any quality filtering. The *S. pneumoniae* pan-genome was computed from 616 high-quality Illumina draft assemblies from a study of children in Massachusetts, USA [23]). The *SARS-CoV-2* pan-genome was obtained as a snapshot of GISAID [66] as of Jan 25, 2023.

Streamed queries are implemented as in **Algorithm 2**, but for better speed in practice, for each range extension, the kLCP array is iterated in both directions until a 0 is found, instead of performing a rank and a select. To support bi-directional model, each queried sequence of length ℓ is split into parts of size $\lfloor 2\sqrt{\ell} \rfloor$. For each, the strand to first perform queries on is predicted using the saturating counters using the knowledge of the result for previous parts. The first strand is queried fully and the other only for k-mers that have not been found.

4 Experimental Evaluation

To evaluate FMSI, we compared it to four state-of-the-art indexes for single k-mer sets: BWA [42], SBWT [1], SSHash [56], and CBL [50]. BWA [42] is a read aligner, based on a highly efficient implementation of the FM-index [29]; when combined with SPSS/simplitigs [59,12], its fastmap command [41] can serve also as an efficient k-mer index [12]. SBWT [1] is an k-mer index based on the spectral Burrows-Wheeler transform and a successor of the BOSS data structure [7]; it comes with tunable compression-related parameters and various modes of execution for different use cases. SSHash [56] is a k-mer index based on minimal perfect hash functions; its parameters need to be adjusted a priori, based on characteristics of the data, such as the number of k-mers from prior k-mer counting. CBL [50] is a marriage of ideas from Conway and Bromage [21] with the smallest cyclic rotations of k-mers, with a focus on dynamicity; the index, however, always needs to be compiled with tuned, data-specific parameters. Here, we used parameters of CBL, SSHash, and SBWT consulted with the authors; all the parameters are provided in **Appendix A.1**.

We used three prokaryotic pan-genomes, each with fundamentally different structural properties (**Table 1**). For each of them and for $k = 15, 23, 31$, we constructed support SPSS/superstring representations, built the individual indexes, and measured the corresponding time and memory requirements (**Table S1**). Then, we evaluated index performance in terms of memory per distinct k-mer and time per k-mer queries (positive/negative, isolated/streamed) (**Figure 2**), and compared the results across the individual k-mer lengths (**Figures S1 to S3**). The evaluation was performed on a server with AMD EPYC 7302 (3 GHz) processor and 251 GB RAM, using an SSD disk and a single core for each program.

We found that FMSI consistently provided the best memory consumption – across all the methods, datasets, and k-mer sizes tested (**Figures 2 and S1 to S3**). At the same time, the speed was of FMSI competitive, typically in the mid range of all the methods. The most space gain was achieved with the subsampled reference, where FMSI used typically twice less space compared to second best tool. With FMSI, we measure impact of transitioning from SPSS to masked superstring (with everything else fixed); the improvement was always noticable, again especially for subsampled reference data (up to a factor of 4). The use of kLCP improved streamed queries by a factor of $2-4$, but at the price of an additional bit per superstring character. Importantly, unlike all other methods, FMSI provided consistent performance regardless of the specific choice of k.

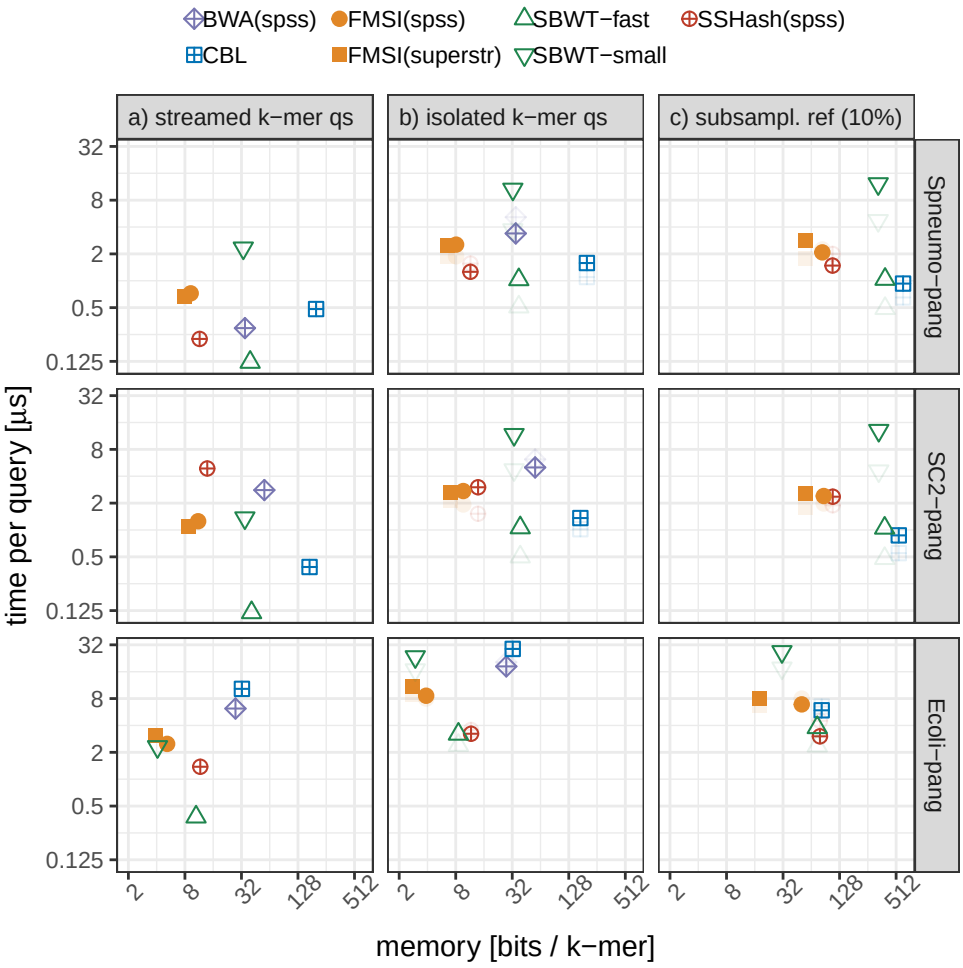

Fig. 2: Time/memory efficiency of k-mer membership queries for the 661k *E.coli*, *SARS-CoV-2*, and *S.pneumoniae* pan-genomes. The plot compares time per query and memory per k-mer queries (isolated (a,c) and streaming (b)), for full (a,b) and subsampled (c) reference k-mer sets. The evaluation was performed using the *E. coli* pan-genome comprising all distinct canonical 23-mers (n=1.17G) of *E. coli* genomes (n=88,749) across the 661k collection [5], the *SARS-CoV-2* pan-genome from GISAID, and the *S. pneumoniae* pan-genome from the RASE DB [23]. The points plotted in light correspond to negative queries (in (b,c)). In (c), BWA is not shown due to excessive space requirements.

5 Discussion and Conclusions

Methods based on k-mer sets have become incredibly popular across bioinformatics, and here we have showed how to query them in a small space, combining ideas from the worlds of k-mer superstrings and BWT indexing. Building upon our previous work on Masked Superstrings [68], we introduced the concept of Masked BWT to design the FMSI index, a data structure analogous to the famous FM-index [29], but substantially simplified to reduce memory. We implemented it in the FMSI tool and demonstrated that indexing directly benefits from the more compact superstring representation of k-mer sets compared to the previous state of the art, SPSS [12,59], especially when the spectrum-like property does not hold. Through our evaluations on prokaryotic pan-genomes, FMSI consistently used the smallest memory across the benchmarked state-of-the-art methods, while providing competitive performance in terms of time per query.

One key outcome of our work is a major space gain, which arises from an original combination of several concepts. The shortest superstring, a famous problem in the theory of approximation algorithms [6,25], gave us a major tool for k-mer data reduction by compactification. In our work, we combined superstrings with another well-known concept, the Burrows-Wheeler Transform (BWT) [16], which had been previously used to obtain efficient k-mer data structures such as BOSS [7] or SBWT [1]. These concepts, together with the mask from [68], which we transform into the SA coordinates, yield the proposed Masked BWT method.

Additionally, the simplicity of the FMSI method – requiring no parameter tuning – starkly contrasts with the other state-of-the-art approaches, such as SSHash [56], SBWT [1], and CBL [50], all of which demand fine-tuned parameters that significantly influence performance. If these existing methods resemble combustion engines with their numerous moving parts, FMSI is akin to electrical vehicle – with fewer components. In its basic form, FMSI operates through just two vectors and simple operations, enabling implementation in just a few lines of code. In contrast, other tools necessitate parameter adjustments by the user during index construction or at compilation time. For our benchmarks, we adhered to parameters recommended by the authors of these tools, based on our preliminary results for the *E. coli* pan-genome, whereas FMSI only requires the specification of k.

The current implementation of FMSI and the associated workflow has several areas of future improvements. First, the FMSI index is constructed from an approximately shortest superstring, whose computation by the global greedy algorithm of KmerCamel [68] currently increases time and memory requirements compared to computing SPSS (see **Table S1**). This also limits the practical usage of k-mer sizes larger than 127, which is possible for FMSI, but those cannot be currently computed with KmerCamel; however, such large values of k are rarely used in practice. Second, due to relying on BWT, our index does not possess good locality properties, unlike CBL [50] or approaches based on super-k-mers such as SSHash [56]. Third, for the evaluation on the *S. pneumoniae* and *SARS-CoV-2* pan-genomes, it may be advantageous to use the unitig flipper tool (`https://github.com/jnalanko/unitig_flipper`) to get better results for SBWT.

Our evaluation was conducted using three prokaryotic pan-genomes with fundamentally different structure and levels of diversity. While the *S. pneumoniae* pan-genome represents high-quality data, the *E. coli* pan-genome is substantially contaminated and thus structurally resembles a metagenome, and the *SARS-CoV-2* pan-genome additionally accumulates significant technological artifacts. As a result, our findings are generalizable, and FMSI is applicable far beyond pan-genomics to individual genomes, transcriptomes, metagenomes, and many other data types. In all these application, our index maintains excellent space efficiency; for instance, for the human genome, our preliminary data indicate memory requirements of only 2.41 bits per distinct 31-mer.

Our findings not only advance the current technology but also open exciting avenues for future research into the applications of k-mer indexing in complex genomic datasets. First, the main question is how to generalize our approach to indexing collections of large k-mer sets. This may be done by computing a single superstring of all k-mers in all the sets and a (compressed) mask for each set; however, the details and an evaluation of this method are left to future work. Second, we address the question of performing set operations on k-mer sets in combination with masked superstrings and indexing in an independent manuscript [69]. Third, FMSI is still a prototype and much additional performance may be gained through better optimization and engineering. For instance, additionally memory savings may come from a better encoding of the SA-transformed mask, e.g., via Elias-Fano [24,28], and compression of superstring BWT and kLCP array by RRR [60] or functionally equivalent methods.

Overall, despite the long history and rich palette of available methods, k-mer set indexing remains an exciting problem with a substantial room for improvement. This is especially true when non-traditional k-mer sets come into the 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 the traditional methods, representing a major weapon in the algorithmic arsenal for future generic, parameter-free, and space-efficient k-based data structures.

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). Portions of this research were conducted at the GenOuest bioinformatics core facility (`https://www.genouest.org`). We are grateful to Camille Marchet, Igor

Martayan, Giulio Pibiri, and Jarno Alanko for their expert advice on parameters for CBL, SSHash, and SBWT. Additionally, we thank Paul Medvedev for his feedback as the reviewer of Ondřej Sladký's BSc. thesis [67], which contained a preliminary version of this work.

Disclosure of Interests. The authors declare no competing interests.

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 *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA23)*, pages 225–236. 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):i260–i269, 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):3155–3163, 2022.
4. T. Bingmann, P. Bradley, F. Gauger, and Z. Iqbal. COBS: a compact bit-sliced signature index. In *String Processing and Information Retrieval: 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7–9, 2019, Proceedings 26*, pages 285–303. 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):1–16, 11 2021.
6. A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis. Linear approximation of shortest superstrings. *J. ACM*, 41(4):630–647, 1994.
7. A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya. Succinct de Bruijn graphs. In B. J. Raphael and J. Tang, editors, *Algorithms in Bioinformatics - 12th International Workshop, WABI 2012, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, volume 7534 of *Lecture Notes in Computer Science*, pages 225–235. Springer, 2012.
8. 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):152–159, 2019.
9. 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):10063, 2015.
10. N. L. Bray, H. Pimentel, P. Melsted, and L. Pachter. Near-optimal probabilistic RNA-seq quantification. *Nature Biotechnology*, 34(5):525–527, 2016.
11. K. Břinda. Novel computational techniques for mapping and classification of Next-Generation Sequencing data. PhD thesis, Université Paris-Est, 2016.
12. K. Břinda, M. Baym, and G. Kucherov. Simplitigs as an efficient and scalable representation of de Bruijn graphs. *Genome Biology*, 22(96), 2021.
13. 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):455–464, 2020.
14. 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.
15. K. Břinda, K. Salikhov, S. Pignotti, and G. Kucherov. Prophyle 0.3.1.0. *Zenodo*, 5281, 2017.
16. M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
17. K. Břinda. Galitime. `https://github.com/karel-brinda/galitime`, 2024.
18. R. Chikhi. K-mer data structures in sequence bioinformatics. HDR thesis, Institut Pasteur Ecole Doctorale "EDITE", 2021.
19. R. Chikhi, J. Holub, and P. Medvedev. Data structures to represent a set of k-long DNA sequences. *ACM Computing Surveys*, 54(1):17:1–17:22, 2022.
20. R. Chikhi, A. Limasset, S. Jackman, J. T. Simpson, and P. Medvedev. On the representation of de Bruijn graphs. In R. Sharan, editor, *Research in Computational Molecular Biology*, pages 35–55, Cham, 2014. Springer International Publishing.
21. T. C. Conway and A. J. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, 2011.
22. V. G. Crawford, A. Kuhnle, C. Boucher, R. Chikhi, and T. Gagie. Practical dynamic de Bruijn graphs. *Bioinformatics*, 34(24):4189–4195, 2018.

23. 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.
24. P. Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974.
25. M. Englert, N. Matsakis, and P. Veselý. Improved approximation guarantees for shortest superstrings using cycle classification by overlap to length ratios. In *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 317–330. ACM, 2022.
26. J. Fan, J. Khan, G. E. Pibiri, and R. Patro. Spectrum preserving tilings enable sparse and modular reference indexing. In H. Tang, editor, *Research in Computational Molecular Biology - 27th Annual International Conference, RECOMB 2023, Istanbul, Turkey, April 16-19, 2023, Proceedings*, volume 13976 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2023.
27. 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):3, 2024.
28. R. M. Fano. *On the number of bits required to implement an associative memory*. Massachusetts Institute of Technology, Project MAC, 1971.
29. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, SFCS-00. IEEE Comput. Soc, 2000.
30. S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.
31. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 841–850, USA, 2003. Society for Industrial and Applied Mathematics.
32. 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):475–476, 2018.
33. 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 *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2226–2234, New York, NY, USA, 2021. Association for Computing Machinery.
34. G. Holley and P. Melsted. Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs. *Genome Biology*, 21(1):1–20, 2020.
35. G. J. Jacobson. *Succinct static data structures*. PhD thesis, Carnegie Mellon University, USA, 1988. AAI8918056.
36. 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.
37. P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3:143–156, 2003.
38. N. J. Larsson and K. Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258–272, 2007.
39. 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):vbac029, 04 2022.
40. H. Li. wgsim. `https://github.com/lh3/wgsim`, 2011.
41. H. Li. Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. *Bioinformatics*, 28(14):1838–1844, 2012.
42. H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.
43. A. Limasset, G. Rizk, R. Chikhi, and P. Peterlongo. Fast and scalable minimal perfect hashing for massive key sets. In C. S. Iliopoulos, S. P. Pissis, S. J. Puglisi, and R. Raman, editors, *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*, volume 75 of *LIPIcs*, pages 25:1–25:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.
44. P.-R. Loh, M. Baym, and B. Berger. Compressive genomics. *Nature Biotechnology*, 30(7):627–630, July 2012.
45. C. Marchet. Advances in colored k-mer sets: essentials for the curious. *arXiv [q-bioGN]*, 2409.05214, 2024.
46. C. Marchet. Advances in practical k-mer sets: essentials for the curious. *arXiv [q-bioGN]*, 2409.05210, 2024.
47. 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):1–12, 2021.
48. 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):i177–i185, 2020.
49. C. Marchet, M. Kerbiriou, and A. Limasset. Blight: efficient exact associative structure for k-mers. *Bioinformatics*, 37(18):2858–2865, 2021.
50. I. Martayan, B. Cazaux, A. Limasset, and C. Marchet. Conway-Bromage-Lyndon (CBL): an exact, dynamic representation of k-mer sets. *bioRxiv*, 2024.01.29.577700, 2024.

51. M. D. Muggli, B. Alipanahi, and C. Boucher. Building large updatable colored de Bruijn graphs via merging. *Bioinformatics*, 35(14):i51–i60, 2019.
52. 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):3181–3187, 2017.
53. 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.
54. P. Pandey, M. A. Bender, R. Johnson, and R. Patro. Squeakr: an exact and approximate k-mer counting system. *Bioinformormatics*, 34(4):568–575, 2018.
55. 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):417–419, 2017.
56. G. E. Pibiri. Sparse and skew hashing of K-mers. *Bioinformatics*, 38(Supplement_1):i185–i194, 2022.
57. G. E. Pibiri. On weighted k-mer dictionaries. *Algorithms for Molecular Biology*, 18(1), 2023.
58. A. Rahman. Compression algorithms for de Bruijn graphs and uncovering hidden assembly artifacts. PhD thesis, The Pennsylvania State University, 2023.
59. A. Rahman and P. Medevedev. Representation of k-mer sets using spectrum-preserving string sets. *Journal of Computational Biology*, 28(4):381–394, 2021. PMID: 33290137.
60. 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):43–es, nov 2007.
61. K. Salikhov. Efficient algorithms and data structures for indexing dna sequence data. PhD thesis, Université Paris-Est, 2017.
62. K. Salikhov, K. Břinda, S. Pignotti, and G. Kucherov. ProPhex. `https://github.com/prophyle/prophex`, 2018.
63. S. Schmidt. Unitigs are not enough: the advantages of superunitig-based algorithms in bioinformatics. PhD thesis, University of Helsinki, 2023.
64. 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):136, 2023.
65. Y. Shibuya, D. Belazzougui, and G. Kucherov. Efficient Reconciliation of Genomic Datasets of High Similarity. In C. Boucher and S. Rahmann, editors, *22nd International Workshop on Algorithms in Bioinformatics (WABI 2022)*, volume 242 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:14, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
66. Y. Shu and J. McCauley. Gisaid: Global initiative on sharing all influenza data–from vision to reality. *Eurosurveillance*, 22(13):30494, 2017.
67. O. Sladký. Masked superstrings for efficient k-mer set representation and indexing. Bachelor's thesis, Charles University, 2024.
68. 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.
69. O. Sladký, P. Veselý, and K. Břinda. Function-assigned masked superstrings as a versatile and compact data type for k-mer sets. *bioRxiv*, 2024.03.06.583483, 2024.
70. 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):e1002195, 2015.
71. E. Ukkonen. A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 5(3):313–323, 1990.
72. D. E. Wood and S. L. Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome Biology*, 15(3):1–12, 2014.

A Appendix

A.1 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 *E. coli* pan-genome was obtained from the phylogenically compressed 661k collection [5] as provided on `https://doi.org/10.5281/zenodo.4602622` [14]; all the genomes from the batches starting by '`escherichia_coli__`' were extracted and used subsequently. The *S. pneumoniae* pan-genome was downloaded from the RASE DB *S. pneumoniae* (`https://github.com/c2-d2/rase-db-spneumoniae-sparc/`) The *SARS-CoV-2* pan-genome was downloaded from GISAID `https://gisaid.org/` (access upon registration) on Jan 25, 2023 (GISAID version 2023/01/23).

Step 2: k-mer set preparation. The input genome files were converted to simplitigs by ProphAsm [12] by '`prophasm -k {k}`'). For *S. pneumoniae* and *SARS-CoV-2* we used ProphAsm v0.1.1 and $k = 32$. For *E. coli* we used ProphAsm v2.0.0 and $k = 63$. The obtained files with the k-mer sets were deposited on `https://doi.org/10.5281/zenodo.13997398`.

Step 3: Representation tailoring. Before indexing with BWA [42], SShash [56] and FMSI, we used ProphAsm [12] (v0.1.1) and KmerCamel [68] (v1.0.2) for preprocessing. The following commands were used (with k matching the k passed to the programs for indexing).

- For BWA/SShash: '`prophasm -i {preprocessed} -o {simplitigs} -k {kmer-size}`'.
- For FMSI with SPSS-like MS: '`kmercamel -p {preprocessed} -o {ms-noopt} -k {kmer-size} -C -a local -d 1`' and then '`kmercamel optimize -p {ms-noopt} -o {ms} -k {kmer-size} -C`'
- For FMSI with best MS: '`kmercamel -p {preprocessed} -o {ms-noopt} -k {kmer-size} -C`' and then '`kmercamel optimize -p {ms-noopt} -o {ms} -k {kmer-size} -C`'

Step 4: Generating k-mer queries. To generate positive streaming queries, we simulated error-free Illumina-like reads by WgSim [40] (v0.3.1-r13) with the following command:
'`wgsim -1300 -d0 -S40 -e0 -r0 -R0 -N10000 {preprocessed} {streaming-queries} /dev/null`'.

To generate isolated queries, we used our adhoc Python 3 script `get_queries.py` (provided in the online supplementary repository), which takes an input FASTA file, reads all k-mers in it, and optionally subtracts k-mers from another FASTA file. It then outputs uniformly random (with repetition) isoalted k-mers as a new FASTA file. The following command was used for positive queries '`get_queries.py -print_header=True -cap=1000000 -k {kmer-size} {preprocessed} > {positive-queries}`'. To get input for the version of SBWT which does not store reverse complementary k-mers, we additionally used '`-print_RC=True`' which for each k-mer also outputs its reverse complement on the next line.

To generate negative (isolated) queries we used: '`get_queries.py -print_header=True -cap=1000000 -k {kmer-size} -e {preprocessed} {human-chromosome} > {positive-queries}`', where '`-e`' flag subtracts the k-mers appearing in the indexed k-mer set from the queries. In particular, we used a 2Mbp prefix of the FASTA file for chromosome 1 of *H. sapiens* genome (GRCh38.p14 Primary Assembly, NC_000001.11), downloaded from NCBI. For input to the version of SBWT which does not store reverse complementary k-mers, we additionally used '`-print_RC=True`' which for each k-mer also outputs its reverse complement on the next line.

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 **Table S1**. The measured performance of the indexes is provided in **Fig. 2** and **Fig. S1 to S3**. Further information can be found in the online repository.

– **FMSI** (`https://github.com/OndrejSladky/fmsi/`, v0.3.0, commit '`2b69a1a`'). Index was always constructed by '`fmsi index -k {kmer-size} -p {prefix}`', where `prefix` is an input file with a masked superstring. Queries considered exists in two modes:
  - **Single variant:** queries using '`fmsi query -k {kmer-size} -p {prefix} -q {queries} -O`', additionally with '`-s`' in case of isolated queries and subsampled reference.

– **BWA** [42,41] (`https://github.com/lh3/bwa`, commit '`79b230d`'). Index was always constructed by '`bwa index {prefix}`', where `prefix` is the input FASTA file with an SPSS of the k-mer set, and queried by '`bwa fastmap -l {kmer-size} -w 999999 {prefix} {queries}`'.

– **SBWT** [1] (`https://github.com/algbio/SBWT`, commit '`8013ade`'). 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 60 -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 60 -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** [56,57] (`https://github.com/jermp/sshash`, commit '`d90ad37`'). An index based on minimal perfect hashing of k-mers, 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 seeds. 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** [50] (`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=28 \\`
`cargo +nightly build --release --examples --target-dir target.k_{kmer-size}`'
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}`'

The time and memory requirements for each individual indexes are provided in **Table S1**, including the time for masked superstring or SPSS computation if required. Memory and time were measured by Galitime [17] (`https://github.com/karel-brinda/galitime`).

A.2 Supplementary figures and tables

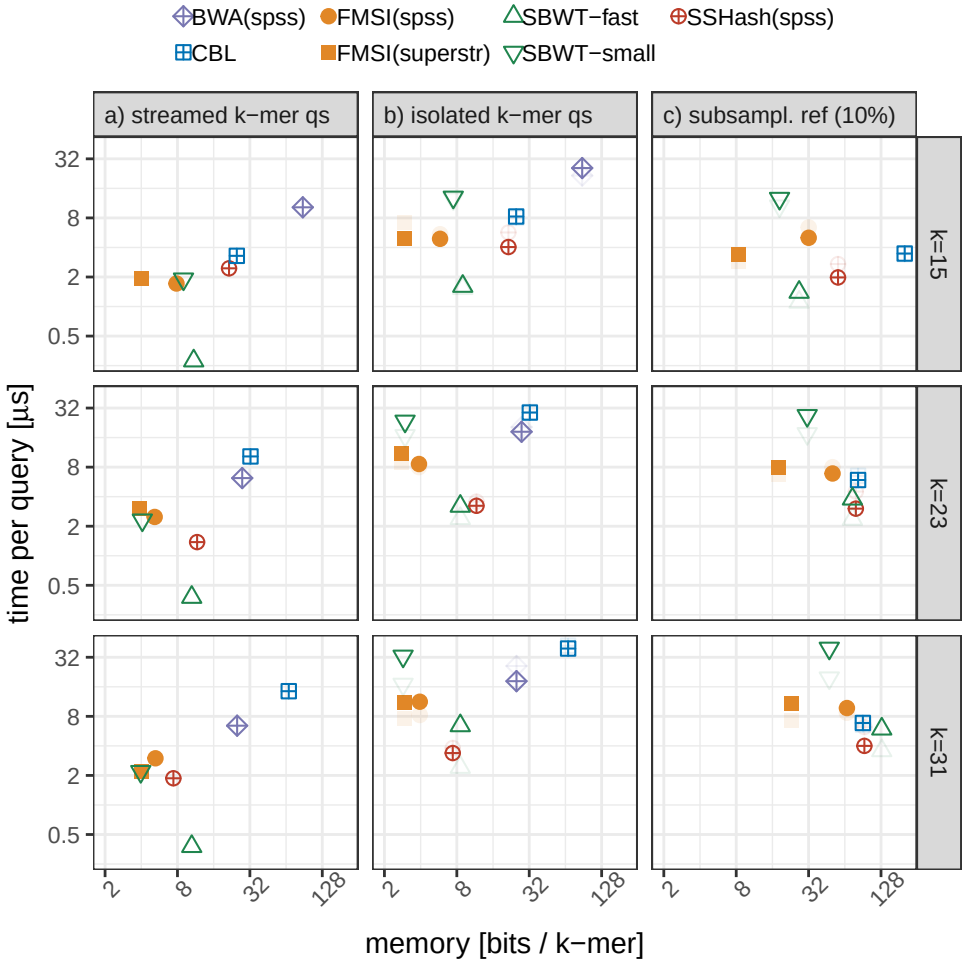

Fig. S1: Time/memory efficiency of k-mer membership queries for the *E.coli* pan-genome and $k = 15, 23, 31$. The plot shows the same information as **Figure 1**.

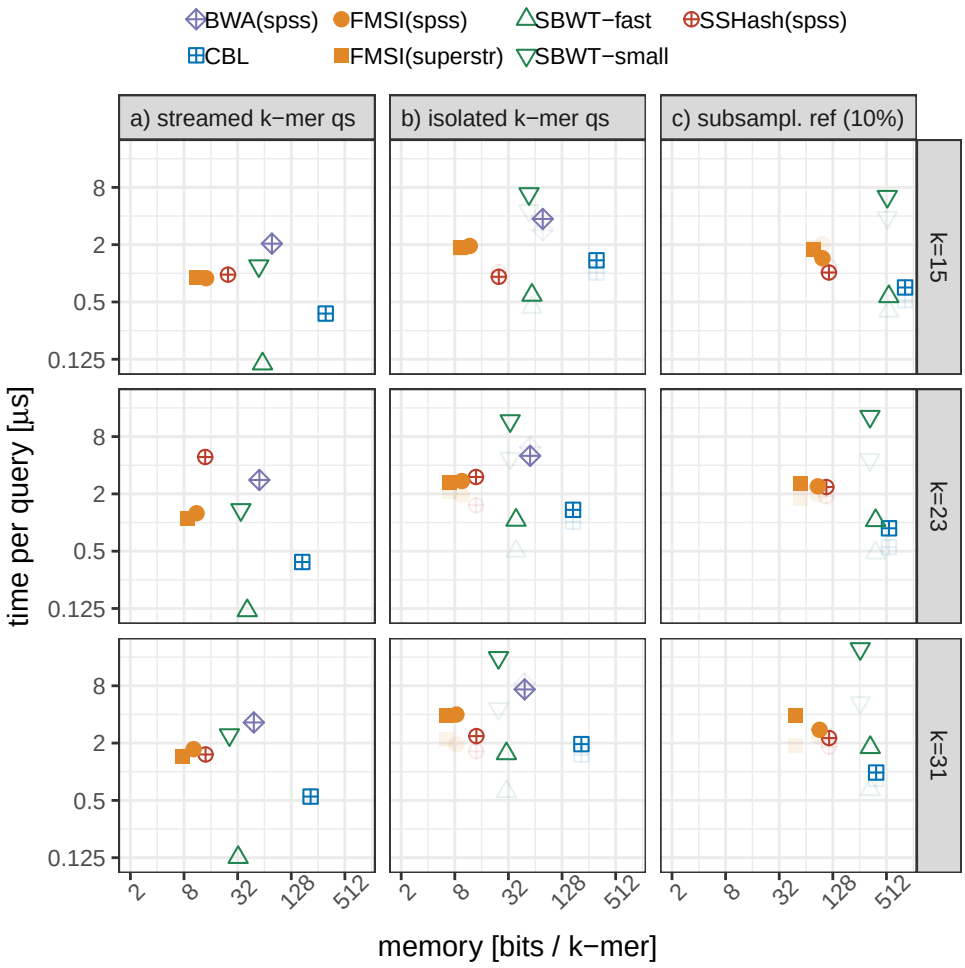

Fig. S2: Time/memory efficiency of k-mer membership queries for the *SARS-CoV-2* pan-genome and $k = 15, 23, 31$. The plot shows the same information as **Figure 1**.

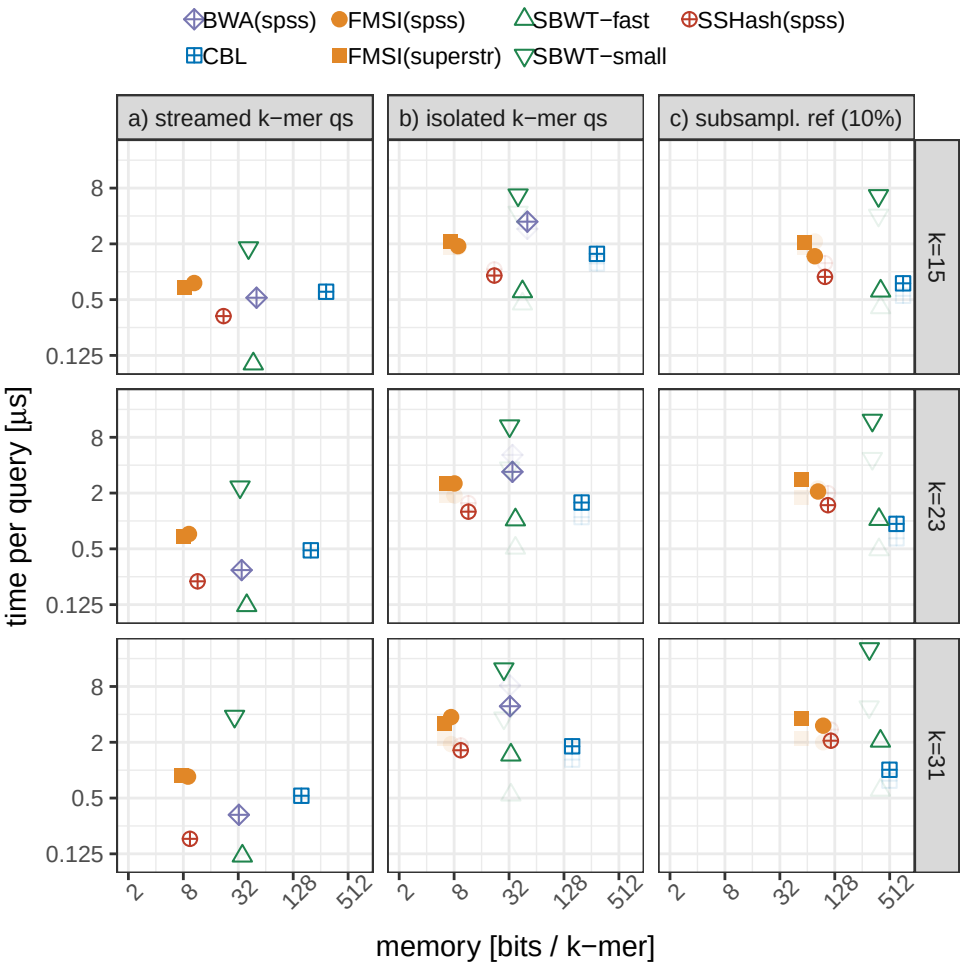

Fig. S3: Time/memory efficiency of k-mer membership queries for the *S.pneumoniae* pan-genome and $k = 15, 23, 31$. The plot shows the same information as **Figure 1**.

Subsampl. rate	k	Tool	MS/SPSS tool	Variant	MS/SPSS time [h]	MS/SPSS mem [GB]	Index constr. time [h]	Index constr. mem [GB]
10%	15	BWA	ProphAsm		0.032	1.569	0.113	3.016
10%	15	CBL	–		–	–	0.013	0.932
10%	15	SBWT	–	fast	–	–	0.333	14.697
10%	15	SBWT	–	small	–	–	0.164	8.162
10%	15	SSHash	ProphAsm		0.032	1.569	0.008	1.503
10%	15	FMSI	KmerCamel's global		0.045	1.707	0.008	1.508
10%	15	FMSI	KmerCamel's local		0.036	0.553	0.051	7.040
10%	23	BWA	ProphAsm		0.119	5.577	0.753	11.884
10%	23	CBL	–		–	–	0.036	1.321
10%	23	SBWT	–	fast	–	–	2.157	69.305
10%	23	SBWT	–	small	–	–	1.177	48.328
10%	23	SSHash	ProphAsm		0.119	5.577	0.030	5.317
10%	23	FMSI	KmerCamel's global		0.360	5.516	0.069	12.477
10%	23	FMSI	KmerCamel's local		0.142	2.199	0.238	40.403
10%	31	BWA	ProphAsm		0.155	5.770	1.184	13.893
10%	31	CBL	–		–	–	0.046	1.746
10%	31	SBWT	–	fast	–	–	4.161	69.304
10%	31	SBWT	–	small	–	–	1.901	69.301
10%	31	SSHash	ProphAsm		0.155	5.770	0.032	5.855
10%	31	FMSI	KmerCamel's global		0.537	6.160	0.107	18.769
10%	31	FMSI	KmerCamel's local		0.181	2.199	0.375	61.453
100%	15	BWA	ProphAsm		0.532	14.843	0.205	3.319
100%	15	CBL	–		–	–	0.299	1.453
100%	15	SBWT	–	fast	–	–	0.148	28.176
100%	15	SBWT	–	small	–	–	0.115	14.106
100%	15	SSHash	ProphAsm		0.532	14.843	0.033	6.362
100%	15	FMSI	KmerCamel's global		0.314	16.771	0.039	7.042
100%	15	FMSI	KmerCamel's local		0.187	4.396	0.129	12.863
100%	23	BWA	ProphAsm		0.770	50.138	0.408	2.532
100%	23	CBL	–		–	–	0.312	6.443
100%	23	SBWT	–	fast	–	–	0.337	18.996
100%	23	SBWT	–	small	–	–	0.249	11.175
100%	23	SSHash	ProphAsm		0.770	50.138	0.124	10.554
100%	23	FMSI	KmerCamel's global		1.114	54.063	0.131	21.589
100%	23	FMSI	KmerCamel's local		0.523	17.569	0.150	25.828
100%	31	BWA	ProphAsm		0.992	54.873	0.444	2.609
100%	31	CBL	–		–	–	0.379	14.486
100%	31	SBWT	–	fast	–	–	0.404	58.634
100%	31	SBWT	–	small	–	–	0.287	35.526
100%	31	SSHash	ProphAsm		0.992	54.873	0.083	7.063
100%	31	FMSI	KmerCamel's global		1.480	60.500	0.149	25.062
100%	31	FMSI	KmerCamel's local		0.376	17.568	0.209	29.538

Table S1: Comparison of superstring and index construction time and memory requirements on the *E. coli* pan-genome dataset. MS/SPSS computation means either computing a masked superstring (MS, for FMSI) or SPSS (for BWA and SSHash).