Optimizing sparse and skew hashing: faster k-mer dictionaries

Giulio Ermanno Pibiri[1] and Rob Patro[2]

[1]DAIS, Ca' Foscari University of Venice, Italy and [2]Dept. of Computer Science, University of Maryland, College Park, MD 20440, USA

Abstract

Motivation: Representing a set of k-mers — strings of length k — in small space under fast lookup queries is a fundamental requirement for several applications in Bioinformatics. A data structure based on *sparse and skew hashing* (SSHash) was recently proposed for this purpose [Pibiri, 2022]: it combines good space effectiveness with fast lookup and streaming queries. It is also *order-preserving*, i.e., consecutive k-mers (sharing a prefix-suffix overlap of length $k-1$) are assigned consecutive hash codes which helps compressing satellite data typically associated with k-mers, like abundances and color sets in colored De Bruijn graphs.
Results: We study the problem of accelerating queries under the sparse and skew hashing indexing paradigm, without compromising its space effectiveness. We propose a refined data structure with less complex lookups and fewer cache misses. We give a simpler and faster algorithm for streaming lookup queries. The refined architecture translates to substantial performance gains, outperforming the original version of SSHash in both index construction speed and query efficiency. Compared to indexes with similar capabilities and based on the Burrows-Wheeler transform, like SBWT and FMSI, SSHash is significantly faster to build and query. SSHash is competitive in space with the fast (and default) modality of SBWT when both k-mer strands are indexed. While larger than FMSI, it is also more than one order of magnitude faster to query.
Availability and Implementation: The SSHash software is available at https://github.com/jermp/sshash, and also distributed via Bioconda. A benchmark of data structures for k-mer sets is available at https://github.com/jermp/kmer_sets_benchmark. The datasets used in this article are described and available at https://zenodo.org/records/17582116.
Contact: giulioermanno.pibiri@unive.it, rob@cs.umd.edu.

Key words: k-mers, minimizers, hashing, associative data structure

1. Introduction

Efficient representation and indexing of large sets of k-mers (substrings of fixed length k appearing in longer strings) is a central primitive in modern Bioinformatics. To tackle this problem, Pibiri [2022] recently introduced a data structure based on *sparse and skew hashing* (SSHash, hereafter). Apart from being compact and fast to query, its distinctive feature is that it is *order-preserving*: k-mers that overlap by $k-1$ characters are likely assigned *consecutive* hash codes. This property makes it particularly well suited for compressing satellite data associated with k-mers, such as abundances and color sets in colored De Bruijn graphs, because values referring to consecutive k-mers tend to be very similar (if not identical), and storing them consecutively significantly improves compression, as well as, locality of reference. Indeed, state-of-the-art indexes for colored [Fan et al., 2024; Campanelli et al., 2024] and positional [Fan et al., 2023] De Bruijn graphs rely on SSHash for this reason. Improving it thus directly impacts on the performance of such indexes.

In this work, we revisit the sparse and skew hashing paradigm with the goal of improving query performance without compromising space effectiveness. We introduce a range of refinements, including: 1. a less complex query logic, 2. a new layout that reduces the number of cache misses per query and leads to consistently better runtime in practice, 3. a simpler and faster algorithm for *streaming* lookup queries. This is a query modality where lookup queries are issued for consecutive k-mers coming from reads or longer sequences. This modality is the one typically used in practice, e.g., to perform pseudo alignment on colored De Bruijn graphs [Bingmann et al., 2019; Fan et al., 2024; Alanko et al., 2023b].

We experimentally evaluate the new SSHash design (which outperforms the previous one in both query and construction time) against the latest state-of-the-art k-mer dictionaries that offer similar functionalities, such as SBWT [Alanko et al., 2023a] and FMSI [Sladký et al., 2025], both based on the Burrows and Wheeler [1994] transform (BWT). Our results indicate that SSHash achieves significantly better query performance and faster construction times. In terms of space usage, SSHash is competitive with SBWT in the bidirectional model (i.e., when both strands of each k-mer are indexed). This is the *de-facto* model used by applications built atop these indexes. The space usage of SSHash is actually better for larger k. It remains, on the other hand, more space-consuming than FMSI, but more than an order-of-magnitude faster to query.

2. Preliminaries

In this section we fix notation and illustrate the basic tools we use throughout the paper to solve the following problem.

Problem 1 (The k-mer dictionary problem) Given a collection $\mathcal{X}$ of strings and an integer $k > 0$, build a data structure that represents all n distinct k-mers of $\mathcal{X}$ so that the following queries are efficient:

- For any k-mer x, $\textsc{Lookup}(x)$ returns a handle $h \in [1..U]$ (where $U \geq n$) if x appears in $\mathcal{X}$, or $\perp$ otherwise. For any $x \neq y$, $\textsc{Lookup}(x) \neq \textsc{Lookup}(y)$ must hold.
- For any $h \in [1..U]$, $\textsc{Access}(h)$ retrieves the k-mer x if $\textsc{Lookup}(x) = h$, or returns the empty string ε otherwise.
- For any string s of length at least k, $\textsc{Lookup}(s)$ answers $\textsc{Lookup}(x)$ for all k-mers x of s.

Basic notation. Given a string s, we indicate with $|s|$ its length and with $s[i..j)$ its substring of length $j - i$ beginning with the symbol in position i and terminating with that in position $j - 1$, for any $1 \leq i < j \leq |s| + 1$. (We use 1-based indexing throughout the paper.) Notation "$x \in s$" means that x appears as a substring of s and we let $\text{pos}(x, s)$ be the start position of x in s.

In this paper we consider strings over the DNA alphabet $\{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$. Each symbol in this alphabet has a complement: the complement of A is T (and vice versa), and the complement of C is G (and vice versa). The *reverse complement* of the string s, indicated with $\overline{s}$, is the string where all the characters of s appear in reverse order and complemented. Since $\overline{s}$ can be obtained from s (and vice versa), we consider them identical.

The k-mer *spectrum* of the string s, $\text{spect}_k(s)$, is the set of k-mers of s. Similarly, we define $\text{spect}_k(\mathcal{X}) = \cup_{s \in \mathcal{X}} \text{spect}_k(s)$. A *spectrum-preserving string set* (or SPSS) for $\mathcal{X}$ is another set of strings $\mathcal{S} = \{s_i\}$ such that $|s_i| \geq k$ for each s_i and $\text{spect}_k(\mathcal{S}) = \text{spect}_k(\mathcal{X})$. In this work we only consider SPSSs that do *not* repeat k-mers, that is $\text{spect}_k(s_i) \cap \text{spect}_k(s_j) = \varnothing$ for any $s_i, s_j \in \mathcal{S}$, $i \neq j$. For the rest of the paper, let $\mathcal{S} = \{s_i\}$ be an SPSS for $\mathcal{X}$ with $N = \sum_i |s_i|$. [1] Given a k-mer x of $\mathcal{X}$, it follows that $\exists!\, s_i \in \mathcal{S}$ such that either $x \in s_i$ or $\overline{x} \in s_i$. Intuitively, $\mathcal{S}$ provides a natural basis for a data structure solving Problem 1 because it contains n distinct k-mers from the input $\mathcal{X}$, without repetitions.

We call $S[1..N]$ the string obtained by concatenating all the strings s_i in some fixed order, and indicate via p_i, the position at which s_i begins in S. Thus, $s_i = S[p_i..p_{i+1})$ for $i = 1..|\mathcal{S}|$ and $p_{|\mathcal{S}|+1} = N + 1$. Let $P = \{p_i\}$ in the following.

Minimizers and super-k-mers. A minimizer sampling scheme is defined by a triple $(m, k, \mathcal{O})$, where $m, k \in \mathbb{N}$, $m < k$, and $\mathcal{O}$ is an order over all m-long strings. Given a k-mer x, the minimizer μ of x is the leftmost m-mer of x such that $\mathcal{O}(\mu) \leq \mathcal{O}(y)$ for any other m-mer y of x. To simplify notation, we indicate the minimizer of the k-mer x with $\textsc{Mini}(x)$ in the following, without specifying the parameters m and $\mathcal{O}$. In practice, the order $\mathcal{O}$ is usually random (e.g., implemented as a pseudo-random hash function).

A string is said to be *sampled* at the positions of the minimizers of its k-mers. For a string composed of N i.i.d. random characters, and when m is sufficiently long, the expected number of *distinct* sampled positions is approximately $2/(k - m + 2) \cdot (N - m + 1)$ (see Theorem 3 by Zheng et al. [2020] for details). In other words, random minimizers are *sparse* along the string; in expectation, the start positions of two consecutive minimizers are $(k - m + 2)/2$ characters apart.

[1] An algorithm that computes $\mathcal{S}$ such that both $|\mathcal{S}|$ and N are minimum is known [Schmidt and Alanko, 2023], improving over previous heuristics [Rahman and Medvedev, 2020; Břinda et al., 2021]. In this case, the strings in $\mathcal{S}$ are called *eulertigs*. This is the form of $\mathcal{S}$ we are going to use in the experimental analysis in Section 8. Furthermore, we note that if duplicate k-mers are allowed in $\mathcal{S}$, the SPSS of minimum total length is composed of *matchtigs* and that SSHash can index matchtigs as well [Schmidt et al., 2023].

We give two more definitions. For any k-mer x, we define its *canonical minimizer* as $\textsc{CMini}(x) := \min\{\textsc{Mini}(x), \textsc{Mini}(\overline{x})\}$.

A *super-k-mer* of a string s is a maximal substring of s where a given property $R(x)$ does not change for all k-mers $x \in s$. We say that s is *parsed* into super-k-mers by property R. As an example, consider the string $s = \mathsf{TCAAGTTGGCCT}$..., and let $k = 7$. Let $R(x)$ be the lexicographic minimizer of length $m = 3$. Then the first super-k-mer of the string would be $\mathsf{TCAAGTTGG}$ because its three k-mers ($\mathsf{TCAAGTT}$, $\mathsf{CAAGTTG}$, and $\mathsf{AAGTTGG}$) all have minimizer AAG according to the lexicographic order of m-mers. Note that this substring is maximal because the fourth k-mer of s does not have AAG as minimizer.

Model of computation: cache misses. To analyze the time complexity of an algorithm, we count the number of cache misses that it spends, as practical performance critically relates to them. We use the following simple model: $C(Q) := \lceil Q/B \rceil$ is the number of cache misses that occur when the algorithm reads Q *consecutive* bits, using a cache with page (or *line*) size equal to B bits [Vitter, 2001]. For most architectures the value B is 512 (64 bytes). That is, B spans several memory words. We assume that $\lceil \log_2(N) \rceil \leq B$ for the rest of the paper[2].

Minimal perfect hash functions. A function $f : X \to \{1, \ldots, |X|\}$ is said to be a minimal perfect hash function for the set X if $f(x) \neq f(y)$ for all $x, y \in X$, $x \neq y$. In other words, a MPHF for X maps its keys into the first $|X|$ natural numbers without collisions. Note that f is *not* defined for a key not in X, thus permitting to implement f with just a constant amount of space per key. Indeed the theoretical space lower bound for representing a MPHF is (less than) $\log_2(e) \approx 1.443$ bits per key [Mehlhorn, 1982], assuming the keys of X are drawn from a large universe and $|X|$ is large as well.

Many practical constructions for MPHFs are known that scale well to large datasets, take little space on top of the lower bound, and retain very fast evaluation time. See the recent survey by Lehmann et al. [2026] for an overview of various techniques. In this paper, we use techniques based on *bucket placement* that specialize in very fast evaluation time, requiring (under proper tuning) 1 cache miss per evaluation [Pibiri and Trani, 2021].

Elias-Fano sequences. Consider a sorted sequence A of n integers $0 \leq A[1] < A[2] < \cdots < A[n] < U$ for some universe size U. The query $\textsc{Successor}(x)$ returns the smallest integer $y \in A$ that is $y \geq x$ (assuming, w.l.o.g., $x \leq A[n]$, so that the result is well-defined). We use the following result, based on Elias-Fano codes (and point the interested reader to the Supplementary Material for details).

Theorem 1 (Elias-Fano) There exists a representation of A that takes at most $\text{EF}(n, U) = n(\ell + 3)$ bits and:

1. For $o(n)$ extra bits, access to the i-th element, can be supported with, at most, $1 + C_{\text{sel}}$ cache misses.
2. For $o(n)$ extra bits, $\textsc{Successor}$ can be supported with, at most, $C_{\text{sel}} + C_{\text{scan}}$ cache misses.
3. For $O(n \log n)$ extra bits, $\textsc{Successor}$ can be supported with, at most, $1 + C_{\text{scan}}$ cache misses.

The quantities C_{sel} and C_{scan} are $C_{\text{sel}} = 2 + C(O(\log^4 n))$ and $C_{\text{scan}} = C(U/n) + C(\ell \cdot (U/n + 1)) + C(\Delta_A)$, where $\Delta_A := \max_{1 \leq i < n} \{\lfloor \frac{A[i+1]}{2^\ell} \rfloor - \lfloor \frac{A[i]}{2^\ell} \rfloor\}$, and $\ell = \lfloor \log_2(U/n) \rfloor$.

[2] We omit ceiling operators when they are not essential.

Algorithm 1 The LOOKUP algorithm for a k-mer x, using the framework illustrated in Section 3.

```
1: function LOOKUP(x)
2:     φ = Φ(x)
3:     t = f(φ)
4:     for j ∈ L[t] do
5:         for q ∈ d(j) do
6:             y = S[q..q + k)
7:             if j = min L[t] and Φ(y) ≠ φ then return ⊥
8:             if x = y then return a unique handle h ∈ [1..U]
9:     return ⊥
```

3. A general indexing framework

With the background and notation fixed in Section 2, the goal of this section is to define a universal, abstract framework that captures the fundamental architecture of any hash-based k-mer index for $\mathcal{S}$. This framework comprises four modular components. By isolating these core components, we can systematically derive and compare both existing tools and our own approach, as different time/space trade-offs can be obtained depending on the choice of each module. While the terminology here is necessarily abstract to maintain generality, we will ground these concepts with concrete examples from the literature.

Definition. The four components of the framework are:

1. A k-mer transformation function Φ. This function takes a k-mer x as input and computes a value $\phi = \Phi(x)$.
2. A table $L[1..M]$, for some $M \leq n$.
3. A function f mapping ϕ to an integer $t \in [1..M]$.
4. The string $S[1..N]$ paired with the start positions $P = \{p_i\}$.

The interaction between these components is described by

$$L[t] = \text{loc}(\phi), \ t = f(\phi), \text{ and } \phi = \Phi(x)$$

where $\text{loc}(\phi)$ is the *locate set* of ϕ, a sorted set of integers for which the following property holds.

Property 1 For each $j \in \text{loc}(\phi)$, there exists a string $s_i = S[p_i..p_{i+1})$ and a k-mer $x \in s_i$ such that $\Phi(x) = \phi$ and $\text{pos}(x, S) \in d(j) \subseteq [p_i..p_{i+1} - k]$.

The function $d(j)$ is a *displacement* (or *decoding*) function that maps the integer j to a set of k-mer positions. (We will add more parameters to the function whenever necessary.)

Intuition. This framework abstracts the query pipeline of a hash-based k-mer index. To achieve space savings, the framework first applies a transformation Φ (e.g., computing a minimizer) to a given k-mer x, deriving a representative signature ϕ. Because the set of these signatures is typically much smaller than the set of all unique input k-mers, the overall memory footprint is reduced. The function f then maps this signature to a specific entry in the table L — the core data structure alongside the string S. To further avoid the memory bottleneck of storing absolute text coordinates for every individual k-mer, the index instead stores a shared locate set $\text{loc}(\phi)$ containing encoded identifiers j. Finally, the displacement function d decodes these identifiers to recover the exact coordinates of x in S.

Queries. Algorithm 1 concretely shows how LOOKUP(x) is implemented following this framework. Again, the idea is to use the value $\phi = \Phi(x)$ to retrieve the set $\text{loc}(\phi)$ and use it to locate the k-mer x in S. Note that all k-mers y read at Line 6 must be such that $\Phi(y) = \phi$ to be present in the input because all the elements $j \in L[t]$ satisfy Property 1. If, instead, $\Phi(y) \neq \phi$, it means that the value ϕ was not observed for any k-mer in the input, hence the search can be terminated directly.

The algorithm for ACCESS(h), instead, depends on the choice of h at Line 8 of Algorithm 1. For example, if the choice $h = q$ is made, then $U = N - k \geq n$ (i.e., LOOKUP does not map k-mers into the minimal range $[1..n]$, unless $|\mathcal{S}| = 1$) and ACCESS(h) is trivial as the k-mer $S[h..h + k)$ is returned directly. Instead, if we opt for $h = q - (i - 1) \cdot (k - 1)$ where i is such that $p_i \leq q \leq p_{i+1} - k$, then we make the mapping minimal, i.e., $h \in [1..n]$. In this case, the value i can be computed using the query SUCCESSOR(q) over P (for example, representing P with the data structure from Theorem 1). Upon ACCESS, however, one has to compute i from h and the sequence P to derive q, and lastly retrieve $S[q..q + k)$.

As the number of cache misses of Algorithm 1 depends on specific choices of data structures for the framework, we defer this analysis to subsequent sections.

Examples. So far in our description, components such as the locate table L, as well as the definitions of Φ, f, and $\text{loc}(\phi)$, are treated abstractly to maintain generality. However, the framework is robust enough to model a wide variety of designs; in the examples below, we provide concrete instantiations of these structures, ranging from simple hash tables to tools like Pufferfish, BLight, and SSHash.

A naïve solution would be to let Φ be a (pseudo) random hash function. Then $\text{loc}(\phi)$ would contain the positions j in S of all the k-mers x such that $\phi = \Phi(x)$. The function f would map the signature ϕ within the bounds of the table L, e.g., $f(\phi) = \phi \bmod M$; the displacement function would just be the identity, i.e., $d(j) = \{j\}$. The key issue of this solution is, obviously, its space usage. First, L stores a position for each k-mer in the input, thus spending $\log_2(N)$ bits per k-mer; second, the number of entries of L (i.e. M) should be chosen large enough, say $M = \Theta(n)$, to achieve fast lookup times.

This issue can be mitigated by letting f be a MPHF for the set of hash codes $\{\Phi(x)\}$ (assuming these are all distinct). That is, $\{\Phi(x)\}$ is mapped bijectively onto the minimal range $[1..n]$ for n input k-mers, so that $M = n$ and $\text{loc}(\phi)$ contains the unique position in S of the k-mer x such that $\phi = \Phi(x)$. To save even more space, $\text{loc}(\phi)$ can store the largest multiple of a chosen quantum $v > 1$ that is smaller than (or equal to) the position of x, effectively spending $\log_2(N/v)$ bits per k-mer instead of $\log_2(N)$ bits. The displacement function d would then indicate the suitable range (of length v, at most). This solution is adopted by the Pufferfish index [Almodaresi et al., 2018].

Minimizers can be used to improve space usage by letting Φ = MINI and the strings in $\mathcal{S}$ be parsed into super-k-mers by Φ. The super-k-mers can then be grouped by minimizer, i.e., all super-k-mers having the same minimizer are concatenated in the string S. Grouping by minimizer naturally leads to a collection of MPHFs, one function per group, to implement the mapping function f. Each MPHF now maps a k-mer appearing in the super-k-mers of the group to its *relative* position in the group. As positions are relative to a group, the space usage improves compared to that of Pufferfish. This is the solution implemented in the BLight index [Marchet et al., 2021].

4. Sparse and skew hashing

We review and analyze the sparse and skew hashing scheme (SSHash) [Pibiri, 2022] in this section, as it lays the foundation for the new development in subsequent sections. Also, we explain how SSHash can be described as a specific instance of the framework from Section 3.

Sparse hashing. The string S is parsed into super-k-mers by Φ, where Φ is either MINI or CMINI, and each $j \in \text{loc}(\phi)$ is the start position of a super-k-mer having minimizer ϕ (red arrows in the example of Figure 1 at page 5). Note that this saves space compared to BLight [Marchet et al., 2021] because the trailing $k-1$ characters of each super-k-mer are encoded once (i.e., the super-k-mers are left where they appear in S).

Let $\mathcal{M} = \{\Phi(x) \mid x \in \text{spect}_k(S)\}$. The function f is a MPHF for $\mathcal{M}$, hence $M = |\mathcal{M}|$. This choice of f also produces a space saving because the MPHF is built over the minimizers rather than the k-mers. As minimizers are *sparse* in the string S, the cost of the MPHF is small (e.g., less than 0.5 bits/k-mer).

A super-k-mer comprises at most $k-m+1$ k-mers by construction[3]. Let j be the starting position of a super-k-mer in S. The displacement function is therefore as follows.

1: **function** $d(j)$
2: let i be such that $p_i \leq j < p_{i+1}$
3: **return** $\left\{j, j+1, \ldots, \min\{j+k-m, p_{i+1}-k\}\right\}$

Since the index i is computed by $d(j)$, the handle returned by SSHash at Line 8 of Algorithm 1 is $h = q-(i-1)\cdot(k-1)$, i.e., the n input k-mers are mapped to the minimal range $[1..n]$.

Skew hashing. Some minimizers ϕ may be very repetitive in S, hence making the set $|\text{loc}(\phi)|$ very large, and LOOKUP slow in the worst case. Fortunately, using a random order and a sufficiently long minimizer length m, the fraction of such minimizers is very small. This is referred to as the *skew* property of minimizer occurrences. We can therefore afford to handle these few minimizers with a more space-consuming data structure that, on the other hand, guarantees a better worst-case runtime. SSHash adopts the following solution.

Let l and r be two integers, such that $0 \leq l < r$. Let $K_i = \{x \in \text{spect}_k(S) \mid 2^i < |\text{loc}(\phi)| \leq \min\{2^{i+1}, max\}, \phi = \Phi(x)\}$ for $l \leq i \leq r$, where $max = \max_{\phi \in \mathcal{M}} |\text{loc}(\phi)|$. By virtue of the skew property of minimizers, we have that $\sum |K_i|$ is a small fraction of the total k-mers. For example, we have that only $\approx$1.3% of the total k-mers belong to such sets, for the whole human genome with $k=31$, $m=21$, and $l=6$. An MPHF f_i is built for each set K_i. For k-mer $x \in K_i$ with $\phi = \Phi(x)$, we store the super-k-mer identifier among $[1..|\text{loc}(\phi)|]$ at position $f_i(x)$ in an array V_i. It follows that an integer in V_i needs $i+1$ bits (or $\log_2(max)$ bits if $i = r$). For the rest of the paper, we refer to a pair (f_i, V_i) for $l \leq i \leq r$ as a *partition* of the skew index. We have $r-l+1$ partitions (at most, when $max \geq 2^r$).

At query time, if $|\text{loc}(\phi)| > 2^l$ then the super-k-mer identifier is retrieved from f_i and V_i and the query k-mer searched *only in one* super-k-mer, instead of $|\text{loc}(\phi)|$ super-k-mers.

Index/query modalities. SSHash operates in two modalities, with different space/time trade-offs. In its *regular* modality, Φ = MINI, hence if LOOKUP$(x) = \perp$ then also LOOKUP$(\overline{x})$ is executed. In the worst case, both loc(MINI(x)) and loc(MINI$(\overline{x})$) are inspected. In the *canonical* modality, Φ = CMINI, so that a k-mer x and its reverse complement $\overline{x}$ are both mapped to the same set loc(CMINI(x)). The LOOKUP algorithm is therefore executed only once (the boolean expression in the **if** at Line 8 of Algorithm 1 becomes: $x = y$ **or** $\overline{x} = y$) for faster query times compared to regular indexing at the price of some more space due to the higher density of canonical minimizers.

Data structures and analysis. The string S is a 2-bit integer vector, as each DNA base can be coded with 2 bits. The array P, indicating the start position of each substring s_i in S, is coded with Elias-Fano and takes at most $\text{EF}(|\mathcal{S}|, N) + o(|\mathcal{S}|)$ bits as per Theorem 1 (Points 1. and 2.). The locate table L is an array of $\log_2(N)$-bit integers, where all locate sets are stored consecutively, one after the other, in the order indicated by the MPHF f. Let $Z = \sum_{\phi \in \mathcal{M}} |\text{loc}(\phi)|$. The space of L is therefore $Z \log_2(N)$ bits. The space for f is $\Theta(M)$ bits. With another array $sizes[1..M+1]$ we keep track of where each $\text{loc}(\phi)$ begins and ends in L. That is, if $t = f(\phi)$, then $\text{loc}(\phi) = L[sizes[t]..sizes[t+1])$, with $|\text{loc}(\phi)| = sizes[t+1] - sizes[t]$. The sorted array $sizes$ is also represented with Elias-Fano and takes at most $\text{EF}(M, Z) + o(M)$ bits (Theorem 1, Point 1.). Letting $\alpha = \sum |K_i|$, we upper bound the cost of the skew index by $\alpha \log_2(N) + \Theta(\alpha)$ bits, because a MPHF is built over each K_i and $\log_2(max) < \log_2(N)$. We thus have the following result.

Theorem 2 (Previous SSHash) The space usage of SSHash is at most $2N + (Z+\alpha)\log_2(N) + \Theta(M) + \Theta(\alpha) + \text{EF}(|\mathcal{S}|, N) + \text{EF}(M, Z)$ bits. The number of cache misses per LOOKUP(x), where $z = |\text{loc}(\text{MINI}(x))|$, is at most

1. $1 + C_{\text{acc}} + C(z \log_2(N)) + z(C_{\text{succ}} + C(4k-2m))$ if $1 \leq z \leq 2^l$,
2. $4 + C_{\text{acc}} + C_{\text{succ}} + C(4k-2m)$ otherwise,

where $C_{\text{acc}} = 3 + C(O(\log^4 M))$, $C_{\text{succ}} = 2 + C(O(\log^4 |\mathcal{S}|)) + C_{\text{scan}}$, and $C_{\text{scan}} = C\left(\frac{N}{|\mathcal{S}|}\right) + C\left(\left(\frac{N}{|\mathcal{S}|}+1\right)\log_2\left(\frac{N}{|\mathcal{S}|}\right)\right) + C(\Delta_P)$.

Figure 2 in the Supplementary Material shows that the space upper bound given in Theorem 2 is quite tight. (Most of the difference between the bound and the measured space comes from overestimating the cost of the skew index.) Table 1 in the Supplementary Material, instead, compares the number of theoretical cache misses in the theorem with the measured ones in practice (and shows that they are very similar).

5. Refining displacements

The first refinement we introduce in the updated SSHash index regards the elements of the set $\text{loc}(\phi)$, where ϕ is a minimizer. As explained in Section 4, in the previous version of SSHash, the elements of $\text{loc}(\phi)$ are the start positions of each super-k-mer whose minimizer is ϕ. While this works seamlessly for both the regular and canonical minimizer, it also involves a linear search of a super-k-mer upon LOOKUP. To avoid the search, we now let $\text{loc}(\phi) := \{\text{pos}(x, S) + \text{pos}(\phi, x) - 1 \mid x \in \text{spect}_k(S) \wedge \Phi(x) = \phi\}$.

Regular displacements. At query time, given $j \in \text{loc}(\phi)$, we compute the putative start position of x in S as $q = j - \text{pos}(\text{MINI}(x), x) + 1$. That is, the displacement function now returns a single position, thus avoiding the need to actually search for the k-mer x in a super-k-mer.

1: **function** $d(j, x)$
2: let i be such that $p_i \leq j < p_{i+1}$

[3] Technically, there can be more than $k-m+1$ consecutive k-mers with the same minimizer. In this case, however, we can split the super-k-mer into chunks of at most $k-m+1$ k-mers.

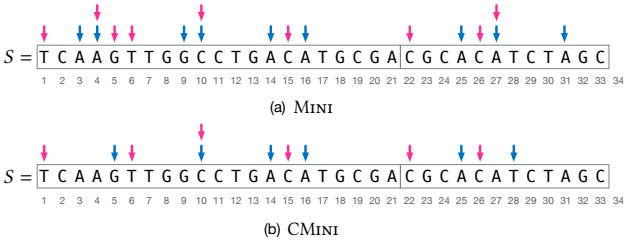

Fig. 1. An example string S of length $N = 33$ resulting from the concatenation of two strings whose start positions in S are $p_1 = 1$ and $p_2 = 22$. There are 21 k-mers for $k = 7$ in the string. The blue arrows indicate the start positions of random minimizers of length $m = 3$, while the red arrows indicate the start positions of the corresponding super-k-mers. In (a), S is parsed into super-k-mers by MINI; in (b), by CMINI. The lexicographic order of m-mers is used to compute minimizers.

3: $p = \text{pos}(\text{MINI}(x), x)$
4: $q = j - p + 1$
5: **if** $p_i \le q \le p_{i+1} - k$ **then return** $\{q\}$
6: **return** $\varnothing$

Refer to the string S shown in Figure 1, panel (a), and consider the query k-mer $x = \mathsf{AACTTGA}$. As explained in Section 4, if $\text{LOOKUP}(x) = \perp$ then $\text{LOOKUP}(\overline{x})$ is executed. We have $\text{MINI}(x) = \mathsf{AAC}$ and $\text{pos}(\text{MINI}(x), x) = 1$. Note that no k-mer of S has minimizer AAC, hence $f(\mathsf{AAC})$ indicates a locate set $\text{loc}(\phi)$ for another minimizer $\phi \neq \mathsf{AAC}$, say ATC which begins in S at position $j = 27$. The position $q = 27 - 1 + 1 = 27$ is computed but $y = S[27..27 + k) \neq x$. Thus, $\text{LOOKUP}(\overline{x})$ is executed, where $\overline{x} = \mathsf{TCAAGTT}$. Now, $\text{MINI}(\overline{x}) = \mathsf{AAG}$ and it begins at position 3 in $\overline{x}$. The candidate position $q = 3 - 3 + 1 = 1$ is computed and $\overline{x}$ is compared against $S[1..1 + k)$ and produces a match.

Canonical displacements. The canonical indexing case, where $\Phi = \text{CMINI}$, is more involved. Recall from Section 2 that we defined the canonical minimizer of a k-mer x as $\text{CMINI}(x) := \min\{\text{MINI}(x), \text{MINI}(\overline{x})\}$. To let the new definition of $\text{loc}(\phi)$ work correctly, we also define

$$\text{pos}(\text{CMINI}(x), x) := \begin{cases} \text{pos}(\text{MINI}(x), x), \text{ if } \text{MINI}(x) \le \text{MINI}(\overline{x}) \\ \text{pos}(\overline{\text{MINI}(\overline{x})}, x), \text{ otherwise} \end{cases}.$$

For example, consider $x = \mathsf{TCAAGTT}$ with $\text{CMINI}(x) = \text{MINI}(\overline{x}) = \mathsf{AAC}$. We have that $\text{pos}(\text{CMINI}(x), x) = 5$ because $\overline{\mathsf{AAC}} = \mathsf{GTT}$ begins at position 5 in x.

Now, given a query k-mer x, we do not know whether it appears in S as x or as $\overline{x}$ (or, if it appears at all). We therefore have to check more than one displacement per position $j \in \text{loc}(\text{CMINI}(x))$ in the worst case:

1. If $\text{MINI}(x) < \text{MINI}(\overline{x})$:
   a. $j - \text{pos}(\text{MINI}(x), x) + 1$ because x can appear in S;
   b. $j - \text{pos}(\overline{\text{MINI}(x)}, \overline{x}) + 1$ because $\overline{x}$ can appear in S and $\text{MINI}(x)$ occurs in $\overline{x}$ as $\overline{\text{MINI}(x)}$.
2. If $\text{MINI}(x) > \text{MINI}(\overline{x})$:
   a. $j - \text{pos}(\text{MINI}(\overline{x}), \overline{x}) + 1$ because $\overline{x}$ can appear in S;
   b. $j - \text{pos}(\overline{\text{MINI}(\overline{x})}, x) + 1$ because x can appear in S and $\text{MINI}(\overline{x})$ occurs in x as $\overline{\text{MINI}(\overline{x})}$.
3. If $\text{MINI}(x) = \text{MINI}(\overline{x})$: all the four displacements are checked.

Observation 1 Let ϕ be a substring of length m of a k-mer x. Then $\text{pos}(\overline{\phi}, \overline{x}) = k - m - \text{pos}(\phi, x) + 2$.

From the previous case analysis and Observation 1, we derive the following displacement function.

1: **function** $d(j, x)$
2: let i be such that $p_i \le j < p_{i+1}$
3: $\mathcal{Q} = \varnothing$
4: **if** $\text{MINI}(x) \le \text{MINI}(\overline{x})$ **then**
5: $p = \text{pos}(\text{MINI}(x), x)$
6: $\mathcal{Q} = \mathcal{Q} \cup \{j - p + 1, j - k + m + p - 1\}$
7: **if** $\text{MINI}(x) \ge \text{MINI}(\overline{x})$ **then**
8: $p = \text{pos}(\text{MINI}(\overline{x}), \overline{x})$
9: $\mathcal{Q} = \mathcal{Q} \cup \{j - p + 1, j - k + m + p - 1\}$
10: **return** $\{q \in \mathcal{Q} \mid p_i \le q \le p_{i+1} - k\}$

Note that for Case 1. and 2., just two displacements are computed. For Case 3., four displacements are computed. Some of them might not be valid and not returned at all (Line 10).

Consider again the string S in Figure 1, panel (b). Suppose we query for $x = \mathsf{TTGGCCT}$. We have $\text{MINI}(\overline{x}) < \text{MINI}(x)$ and $\text{MINI}(\overline{x}) = \mathsf{AGG}$, which begins at position 1 in $\overline{x} = \mathsf{AGGCCAA}$. The minimizer AGG appears in S as $\overline{\mathsf{AGG}} = \mathsf{CCT}$ at position $j = 10$. The two positions to check, returned by $d(x, j)$, are therefore $\{10, 6\}$ and x is found at position 6 in S.

Lastly, the following observation will be useful for Theorem 3 in the analysis from Section 6.

Observation 2 The k-mers starting at positions $d(j, x)$ occur in the same substring of S and this substring is at most $2k - m$ characters long. Hence, accessing S at positions in $d(j, x)$ in ascending order costs $C(4k - 2m)$ cache-misses.

6. Cache-efficient sparse hashing

In this section, we present a second refinement: a new data structure for sparse hashing that reduces the number of cache misses involved during LOOKUP.

We distinguish three *types* of minimizers based on their number of occurrences in S: we call a minimizer *singleton* if it appears once, *light* if it appears more than once but at most 2^l times for a small value of $l \ge 1$, and *heavy* otherwise (it appears more than 2^l times). To reduce cache misses during LOOKUP queries, the idea is to reorganize the storage of the locate sets $\text{loc}(\phi)$ into three arrays according to the type of each minimizer. Let $T[1..M]$ be an array of $(\log_2(N) + 1)$-bit integers, referred to as *tags* in the following, indexed by f. We use this array of tags to indicate the type of the minimizer and therefore retrieve its locate set from a dedicated array. This design allows LOOKUP to follow type-specific execution paths, thereby reducing cache misses. Retrieving the tag itself involves computing f and accessing $T[f(\phi)]$: these two operations cost 2 cache misses.

- *Singleton minimizers.* If the minimizer appears once, the least significant bit of the tag is $\mathsf{0}$ and the remaining $\log_2(N)$ bits encode the single position where the minimizer appears in S. At query time, whenever we read status bit $\mathsf{0}$, we know the minimizer is singleton and the only position in the locate set is readily available in the tag without any further memory access.
- *Light minimizers.* All locate sets of size $2 \le z \le 2^l$ are stored contiguously in an array L, grouped by increasing size. We maintain a small auxiliary array $G[1..2^l]$, where $G[z]$ stores the starting position in L of the group containing sets of size z. For a light minimizer ϕ such that $z = |\text{loc}(\phi)|$,

the tag consists of (from least to most significant bits): a 2-bit status field equal to 01; l bits encoding the value $z-2$; $\log_2(N)-l-1$ bits encoding the position, i, of $\text{loc}(\phi)$ in the group of all sets of size z[4]. At query time, we decode z and i from the tag and access $\text{loc}(\phi) = L[G[z]+iz..G[z]+(i+1)z)$.

Retrieving $G[z]$ costs 1 cache miss. The number of cache misses involved during a scan of $\text{loc}(\phi)$ are $C(|\text{loc}(\phi)|\log_2(N)) \leq C(2^l \log_2(N))$.

- *Heavy minimizers.* Minimizers occurring more than 2^l times are assigned status 11 and their locate sets are stored in another array H. These minimizers are handled with a skew index. Recall from Section 4 that a skew index comprises a collection of (at most) $r-l+1$ pairs (f_i, V_i), where f_i is a MPHF and V_i is an vector of $(i+1)$-bit integers. Here, we choose r so that $r-l+1 \leq 8$ and a skew index partition identifier i can be coded in 3 bits. For a heavy minimizer ϕ, the corresponding tag stores: a 2-bit status equal to 11; 3 bits encoding the skew partition identifier i; the remaining $\log_2(N)-4$ bits encoding the absolute offset o of $\text{loc}(\phi)$ in H. Assume $\phi = \Phi(x)$ is a heavy minimizer. After recovering i and o from its tag, the desired position $j \in \text{loc}(\phi)$ is obtained as $j = H[o + V_i[f_i(x)]]$, involving 3 cache misses.

Figure 2 illustrates how this layout differs from the previous one described in Section 4.

Analysis. Let $\beta \in [0,1]$ be the fraction of minimizers that are not singleton (the number of singleton minimizers is $(1-\beta)M$). The array of tags takes $M(\log_2(N)+1)$ bits, whereas the arrays L and H take $(Z-(1-\beta)M)\log_2(N)$ bits (the array G is a global redundancy, which is negligible for small l, even if stored uncompressed as we do). The other costs are those for S and f, which are the same as those in Theorem 2, and P that we now represent with the data structure from Theorem 1, Point 3. Importantly, we eliminate the *sizes* array altogether. While this does not affect space too much, it is rather relevant to reduce the number of cache misses (see the next theorem). Considering the refined displacements from Section 5 and the new layout in this section, we obtain the following result.

Theorem 3 (Current SSHash) The space usage of SSHash is at most $2N + (Z+\alpha)\log_2(N) + M(1+\beta\log_2(N)) + \Theta(\alpha) + \Theta(M) + O(|\mathcal{S}|\log|\mathcal{S}|) + \text{EF}(|\mathcal{S}|, N) + \text{EF}(M, Z)$ bits. The number of cache misses per Lookup(x), where $z = |\text{loc}(\text{Mini}(x))|$, is at most

1. $2 + C_{\text{succ}} + C(2k)$ if $z = 1$,
2. $3 + C(z\log_2(N)) + z(C_{\text{succ}} + C(2k))$ if $2 \leq z \leq 2^l$,
3. $5 + C_{\text{succ}} + C(2k)$ otherwise,

where $C_{\text{succ}} = 1 + C_{\text{scan}}$ and C_{scan} is the same as that in Theorem 2. When $\Phi = \text{CMini}$, the cost $C(2k)$ in Case 2. becomes $C(4k-2m)$ due to Observation 2.

Assuming the same constants hidden in the Θ terms of Theorem 2 and Theorem 3, e.g., by using the same MPHF data structure, the space increase of Theorem 3 over Theorem 2 is less than $M(\beta\log_2(N) - \log_2(Z/M) - 1) + O(|\mathcal{S}|\log|\mathcal{S}|)$ bits. This space is small when β decreases as m increases.

[4] Technically speaking, we should choose l so that the number of minimizers with two occurrences is less than $2^{\log_2(N)-l-1}$. Due to the skew distribution of such occurrences for sufficiently long m, this is always the case for $l = 6$ across all of our experiments.

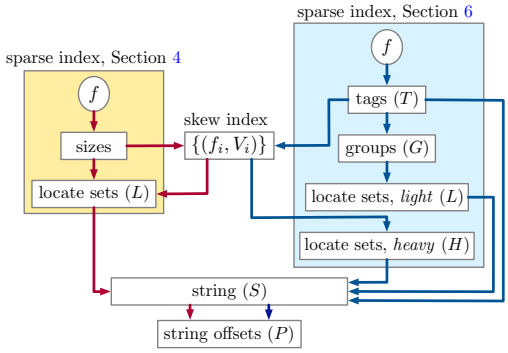

Fig. 2. A graphical comparison between the components of the two versions of SSHash described in Section 4 and Section 6 (symbolic names used in the text are reported in parentheses). The skew index, S, and P, are common to both versions. In this (multi) graph, a path from f to P corresponds to the flow of execution upon Lookup (with the constraint that edges along the path do not change color). Note, in fact, that there are two possible red paths for the previous index (corresponding to the cases of Theorem 2), and three blue paths for the current one (cases of Theorem 3).

For example, β is on average 0.053 for $k = 31$ and $m \geq 19$, $\Phi = \text{Mini}$, on the datasets used in our analysis in Section 8 (see also Figure 1 and Figure 2 in the Supplementary Material.)

For the cache miss analysis in practice, refer again to Section 2 and Table 1 in the Supplementary Material.

7. Streaming query algorithm

Finally, we describe the simplified streaming query algorithm we adopt in the improved SSHash index. Let s be a string with $|s| \geq k$. We consider the query Lookup(s), from Problem 1, that answers Lookup(x) for all k-mers $x \in s$.

Given that two consecutive k-mers of s, say w and x, share a $(k-1)$-long suffix-prefix overlap, answering Lookup(x) *after* Lookup(w) should be performed faster than issuing the queries for two k-mers of s picked in any order. That is, a good implementation of Lookup(s) should *stream* through the k-mers of s to exploit the overlap information and, hence, accelerate the query.

Algorithm 2 shows the solution we use in SSHash. The general idea is to maintain the *state* of the last match w and use this information to perform faster pattern matching. For example, if the last match was found at position q in S, the next query, say for k-mer x, will compare x (or $\overline{x}$) to $S[q+1..q+1+k)$. The state of the last match w, is made up of several variables: its handle h, the identifier i of the comprising string, its orientation $o \in \{-1, +1\}$ in S (under the convention that $+1$ indicates the forward orientation, and -1 indicates the backward orientation), whether its minimizer was *found* in S, its position q in S, and its minimizers Mini(w) and Mini($\overline{w}$). (Note that the tuple $(h, i, o, \textit{found})$, computed at Line 30, can be returned by the Lookup algorithm with a straightforward modification of Algorithm 1.)

The other two variables that are part of the state are the boolean flag *start* and the integer *budget*. The *start* variable is used to let the function Minimizers (Line 14) compute correctly the minimizers of x *knowing* those of the last match w: that is, if *start* = **true** then w is not defined and the pair (Mini(x), Mini($\overline{x}$)) is computed from scratch; otherwise, the pair is computed in amortized $O(1)$ time (using, e.g., the folklore *re-scan* method; see, the discussion in [Groot Koerkamp and Martayan, 2025; Zheng et al., 2025]). The integer *budget* defines

Algorithm 2 The LOOKUP(s) algorithm for a query string s. In the pseudo code, we assume an "iterator-like" interface for the string S such that: S.AT(q) instantiates the iterator at position q of S, and S.NEXT( ) moves the iterator from the current position to the next and returns the k-mer at such position (taking into account the orientation o of the last match).

```
1: function LOOKUP(s)
2:     INIT( )
3:     H = ∅
4:     for k-mer x ∈ s do
5:         h = S-LOOKUP(x)
6:         H = H ∪ {h}
7:     return H
8: function INIT( )
9:     h = ⊥
10:    budget = 0
11:    start = found = true
12:    MINI(w) = MINI(w̄) = ε
13: function S-LOOKUP(x)
14:    (MINI(x), MINI(x̄)) = MINIMIZERS(x)
15:    if budget = 0 then
16:        SEED( )
17:    else
18:        y = S.NEXT( )
19:        if x = y or x̄ = y then
20:            h = h + o
21:            budget = budget − 1
22:        else
23:            SEED( )
24:    MINI(w) = MINI(x), MINI(w̄) = MINI(x̄)
25:    start = false
26:    return h
27: function SEED( )
28:    budget = 0
29:    if MINI(x) = MINI(w) and MINI(x̄) = MINI(w̄) and
       found = false then return
30:    (h, i, o, found) = LOOKUP(x, x̄, MINI(x), MINI(x̄))
31:    if h = ⊥ then return
32:    q = h + (i − 1) · (k − 1)
33:    budget = p_{i+1} − k − q
34:    if o = −1 then
35:        q = q + k
36:        budget = q − p_i
37:    S.AT(q)
```

the maximum allowable number of extensions starting from position q in S. As we extend, we decrease the budget (Line 18-21). When the budget is exhausted, we update the state of the algorithm with the function SEED.

This logic is, *essentially*, the same as that described in the prior SSHash work (Section 4.3 of [Pibiri, 2022]) but *simpler*, thanks to the *budget* "trick". The previous algorithm attempts to extend only if the minimizers of the last match w are the same as those of the current query x, and calls SEED whenever they change. While this already grants a fair deal of extension (because consecutive k-mers are likely to have the same minimizer), in the logic presented here, we attempt an extension whenever $budget > 0$ — even when minimizers change through the stream. We show in the Supplementary Material (Figure 4) that the extension rate of this refined logic is consistently higher than the previous one, granting faster query times. The two refinements presented in Section 5 and Section 6 have an important impact on the runtime of this streaming query algorithm as well.

8. Experimental analysis

In this section, we compare the new SSHash design against the two state-of-the-art k-mer dictionaries, SBWT [Alanko et al., 2023a] and FMSI [Sladký et al., 2025]. (The Supplementary Material also reports on the comparison against the previous published version of SSHash [Pibiri, 2022], using the same methodology and datasets described in this section.)

In particular, we report on the results of the experiments that were collected during November 2025 with the help of the authors of both the SBWT and FMSI. We maintain the benchmark at https://github.com/jermp/kmer_sets_benchmark, to encourage reproducibility of results. The scripts available at the repository list the precise options we used for the tools; we just report some details here.

The SBWT was always built by indexing both k-mer strands as to accelerate query processing, using the so-called "plain-matrix" variant. This is the recommended usage by the authors (and the one used in their tool Themisto [Alanko et al., 2023b] — an index for colored De Bruijn graphs based on the SBWT). Both SBWT and FMSI indexes make use of the *longest common prefix* (LCP) array to speed up queries. The space for this additional array is also included in the reported space usage.

Hardware and compiler. All experiments were executed on a machine equipped with a AMD Ryzen Threadripper PRO 7985WX CPU, 250 GB of RAM, and a Seagate IronWolf 12 TB NAS HDD, running Ubuntu 24.04.3 LTS. All software is written in `C++` and was compiled with gcc 13.3.0 using the highest optimization setting (compiler options: `-O3 -march=native`.)

Datasets. For the experiments reported here we used two types of datasets: whole genomes and pangenomes. For the former type and for consistency with prior published work, we used the whole genomes of *Gadus morhua*, *Falco tinnunculus*, and *Homo sapiens* that are named Cod, Kestrel, and Human, respectively in the following (containing approx. 0.5, 1.5, and 2.5 billion distinct k-mers). For the latter type, we used: NCBI-v — a collection of 18,836 virus assemblies downloaded from RefSeq in October 2025 (approx. 0.4 billion k-mers); SE — a pangenome containing all the 534,751 *Salmonella enterica* genomes from Release 0.2 of the "All The Bacteria" collection [Hunt et al., 2025] (approx. 0.9 and 1.5 billion k-mers for $k = 31$ and $k = 63$ respectively); HPRC — a human pangenome available at https://zenodo.org/records/14854401 (approx. 3.7 and 5.9 billion k-mers for $k = 31$ and $k = 63$ respectively). From these input collections, we computed spectrum-preserving string sets in the form of eulertigs [Schmidt and Alanko, 2023] using the GGCAT algorithm [Cracco and Tomescu, 2023]. These eulertigs, along with detailed instructions about how we computed them, are available at https://zenodo.org/records/17582116, so that it is easy to reproduce our results.

Table 2 in the Supplementary Material reports the exact number of unique k-mers for the two values of k we use in this analysis, and the minimizer lengths used by SSHash.

Index space. Plots (a) and (b) of Figure 3 report index space in avg. bits/k-mer. FMSI is consistently the smallest index, taking between 3.3 and 5.0 bits/k-mer, whereas the SBWT has a steady usage of 10.5 bits/k-mer. SSHash is competitive with the space usage of SBWT, and its space lowers substantially for larger k as minimizers become sparser. For this reason, in some cases like Kestrel and NCBI-v for $k = 63$, it is less than 1 bit per k-mer away from the effectiveness of FMSI.

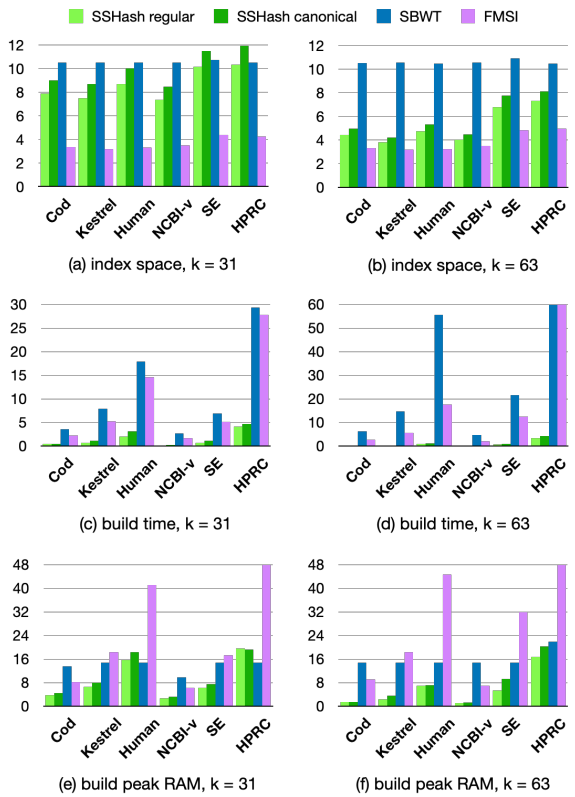

Fig. 3. Comparison of index space in avg. bits/k-mer (plots (a)-(b)), build time in minutes (plots (c)-(d)), and build peak RAM usage (resident set size, or RSS) in GiB (plots (e)-(f)). In plot (d), we cut the bars for SBWT at 60 minutes for ease of visualization because it took three hours to build on HPRC. (FMSI was built in 62 minutes.) Similarly, we cut the bars for FMSI in plots (e)-(f) at 48 GiB because it required 70 and 125 GiB of RAM on HPRC for $k = 31$ and $k = 63$ respectively.

Construction efficiency. All construction algorithms read the input files from the (mechanical) disk of the testing machine. SSHash used a maximum of 16 GiB of RAM during construction and 64 threads. The same configuration was used for SBWT, whereas FMSI's construction does not accept such parameters.

We explicitly clarify that the build times reported here strictly measure the *construction of the indexes*. They do not encompass the time required for upstream preprocessing steps, such as the construction of masked super-strings for FMSI or the generation of eulertigs for SSHash. It should be therefore clear that our plots do not represent the total end-to-end operational cost starting from raw datasets.

Plots (c) and (d) of Figure 3 report the time to build the indexes. Even on the largest collections, SSHash completed within 5 minutes, whereas the other tools took up to 1 hour. (For FMSI, we do not include the time it takes to pre-process the input to obtain the so-called *masked super-string* that it indexes.) SBWT is slower than FMSI especially for larger k because it enumerates and sorts k-mers co-lexicographically on disk (for both strands).

Lastly, plots (e) and (f) display the peak RAM usage (in GiB) during index construction. Both SSHash and SBWT generally remain within the allocated 16 GiB memory budget, except when processing the HPRC dataset. For SSHash, this exception occurs because the final HPRC index occupies an additional 5.6 GiB of memory on top of the 16 GiB construction buffer. In contrast, the construction phase of FMSI does not currently support bounding RAM usage.

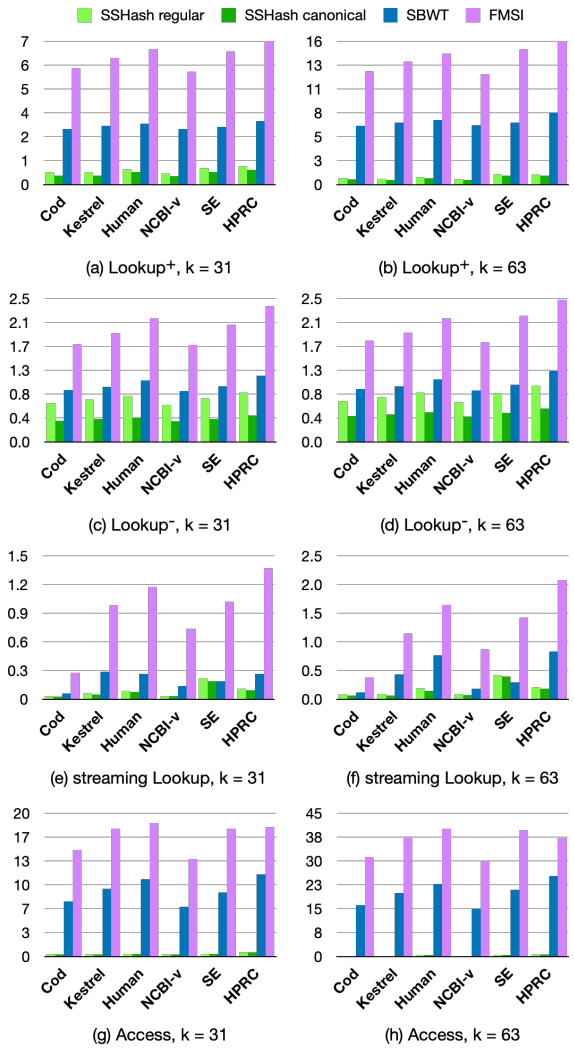

Fig. 4. Comparison of query times in avg. μs/k-mer.

Query efficiency. We say LOOKUP(x) is *positive* if k-mer x is found in the dictionary (indicated with LOOKUP$^+$ in the plots) and *negative* otherwise (indicated with LOOKUP$^-$). To benchmark positive LOOKUP, 10^6 k-mers were sampled uniformly at random from the collections and used as input. Importantly, half of them were reverse complemented to test the LOOKUP algorithm in the most general case. To benchmark negative LOOKUP instead, 10^6 synthetic k-mers were created, having characters selected uniformly at random from the alphabet $\{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$. For ACCESS, we generated 10^6 random handles and retrieved the k-mers corresponding to those handles. The reported times are the averages among 5 runs of the same experiment.

To benchmark streaming LOOKUP queries, we take as input all the reads from FASTQ files, also available in the data repository at https://zenodo.org/records/17582116. These files, that contain millions of reads and are compressed with `gzip`, are decompressed on the fly while performing the queries. The reported time therefore also includes the incremental decompression overhead which, however, is marginal and paid by all the tested tools. The reads were chosen for each collection as to simulate a "high-hit" workload, i.e., where most k-mers are present in the index (say, more than 75% for $k = 31$).

All query algorithms were run using a single processing thread. All indexes were loaded from disk to RAM after construction to perform the queries.

Figure 4 illustrates the comparison between query times. SSHash is generally the fastest index for all queries, by a wide margin. FMSI is the smallest index on disk as said before, but also the slowest to query. For example, the SBWT is 2× faster than FMSI for random Lookup queries and much faster at streaming queries. Remarkably, SBWT is even faster than SSHash on the SE dataset for streaming queries, $k = 63$. The Access query is problematic for indexes based on the BWT, as it requires tens of microseconds whereas SSHash takes a fraction of a microsecond.

In summary, SSHash regular is on average 4.2, 1.3, and 2.9 faster than the next fastest index (SBWT) for positive, negative, and streaming queries respectively, for $k = 31$; and 8.7, 1.2, 2.9 respectively for $k = 63$. (These factors are higher for SSHash canonical: 5.5, 2.5, 3.5 for $k = 31$; 10, 2, 3.6 for $k = 63$).

9. Conclusions and future work

We presented an improved sparse and skew hashing design for the k-mer dictionary problem, featuring a cache-efficient layout, simpler queries, and faster streaming lookups. These speed improvements are expected to accelerate downstream processing, including indexing weighted/colored De Bruijn graphs and spectrum-preserving tilings.

Compared to other indexes based on the celebrated Burrows-Wheeler transform, we found SSHash to be faster to query and build, but to generally consume more space. Batch processing mitigates memory latency in compressed indexes. For instance, batched k-mer lookups significantly improve SBWT throughput [Alanko et al., 2025]. While SSHash would also benefit, BWT-based indexes likely gain a greater relative advantage.

Future work will study the applicability of different string sampling schemes for SSHash. For example, preliminary experiments show that *mod-minimizers* [Groot Koerkamp et al., 2025] have the potential to reduce space consumption without hurting lookup time. Another direction could replace the locally-consistent sampling of minimizers with a *sequence-specific* one. The latter has the promise of achieving optimal density but its impact on query time, on the other hand, has yet to be analyzed.

Acknowledgments. The authors thank Oleksandr Kulkov for his code contributions and Paul Medvedev for useful discussions. The first author also thanks Gianluca Caiazza who gave him access to the hardware used for the experiments in this work.

Fundings. R.P.: This work is supported by the NIH under grant award numbers R01HG009937. Also, this project has been made possible in part by grants DAF2024-342821, DAF2022-252586 from the Chan Zuckerberg Initiative DAF, an advised fund of Silicon Valley Community Foundation.

Competing interests. R.P. is a co-founder of Ocean Genomics Inc.

References

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*, pages 225–236, 2023a.

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, 2023b.

J. N. Alanko, E. Biagi, J. Mackenzie, and S. J. Puglisi. Batched-mer lookup on the spectral burrows-wheeler transform. In *ALENEX*, pages 95–106, 2025.

F. Almodaresi, H. Sarkar, A. Srivastava, and R. Patro. A space and time-efficient index for the compacted colored de Bruijn graph. *Bioinformatics*, 34(13):i169–i177, 2018.

T. Bingmann, P. Bradley, F. Gauger, and Z. Iqbal. COBS: a compact bit-sliced signature index. In *SPIRE*, pages 285–303, 2019.

K. Břinda, M. Baym, and G. Kucherov. Simplitigs as an efficient and scalable representation of de Bruijn graphs. *Genome biology*, 22(1):1–24, 2021.

M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. In *Digital SRC Research Report*, 1994.

A. Campanelli, G. E. Pibiri, J. Fan, and R. Patro. Where the patterns are: repetition-aware compression for colored de Bruijn graphs. *Journal of Computational Biology*, 31(10):1022–1044, 2024.

A. Cracco and A. I. Tomescu. Extremely fast construction and querying of compacted and colored de Bruijn graphs with GGCAT. *Genome Research*, 33(7):1198–1207, 2023.

J. Fan, J. Khan, G. E. Pibiri, and R. Patro. Spectrum preserving tilings enable sparse and modular reference indexing. In *RECOMB*, pages 21–40, 2023.

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.

R. Groot Koerkamp and I. Martayan. SimdMinimizers: Computing Random Minimizers, fast. In *SEA*, pages 20:1–20:19, 2025.

R. Groot Koerkamp, D. Liu, and G. E. Pibiri. The open-closed mod-minimizer algorithm. *Algorithms for Molecular Biology*, 20 (1):4, 2025.

M. Hunt, L. Lima, D. Anderson, G. Bouras, M. Hall, J. Hawkey, O. Schwengers, W. Shen, J. A. Lees, and Z. Iqbal. AllTheBacteria – all bacterial genomes assembled, available, and searchable. *bioRxiv*, 2025.

H.-P. Lehmann, T. Mueller, R. Pagh, G. E. Pibiri, P. Sanders, S. Vigna, and S. Walzer. Modern minimal perfect hashing: A survey. *ACM Computing Surveys*, 58(10), 2026.

C. Marchet, M. Kerbiriou, and A. Limasset. BLight: Efficient exact associative structure for k-mers. *Bioinformatics*, 37(18): 2858–2865, 2021.

K. Mehlhorn. On the program size of perfect and universal hash functions. In *FOCS*, pages 170–175, 1982.

G. E. Pibiri. Sparse and skew hashing of k-mers. *Bioinformatics*, 38 (Supplement_1):i185–i194, 2022.

G. E. Pibiri and R. Trani. PTHash: Revisiting FCH minimal perfect hashing. In *SIGIR*, pages 1339–1348, 2021.

A. Rahman and P. Medvedev. Representation of k-mer sets using spectrum-preserving string sets. In *RECOMB*, pages 152–168, 2020.

S. Schmidt and J. N. Alanko. Eulertigs: minimum plain text representation of k-mer sets without repetitions in linear time. *Algorithms for Molecular Biology*, 18(1):5, 2023.

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.

O. Sladký, P. Veselý, and K. Břinda. From superstring to indexing: a space-efficient index for unconstrained k-mer sets using the masked burrows-wheeler transform (mbwt). *Bioinformatics Advances*, page 10, 2025.

J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing surveys*, 33(2): 209–271, 2001.

A. Zheng, I. Lee, V. S. Shivakumar, O. Y. Ahmed, and B. Langmead. Fast and flexible minimizer digestion with digest. *Bioinformatics*, 41(7), 2025.

H. Zheng, C. Kingsford, and G. Marçais. Improved design and analysis of practical minimizers. *Bioinformatics*, 36 (Supplement_1):i119–i127, 2020.

SUPPLEMENTARY MATERIAL

Optimizing sparse and skew hashing: faster k-mer dictionaries

Giulio Ermanno Pibiri[1] and Rob Patro[2]

[1]DAIS, Ca' Foscari University of Venice, Italy and [2]Dept. of Computer Science, University of Maryland, College Park, MD 20440, USA

Abstract

Supplementary Material for the paper "Optimizing sparse and skew hashing: faster k-mer dictionaries".
Contact: giulioermanno.pibiri@unive.it, rob@cs.umd.edu.

1. Elias-Fano sequences

Consider a sorted sequence A of n integers

$$0 \leq A[1] < A[2] < \cdots < A[n] < U$$

for some universe size U. The query $\textsc{Successor}(x)$ returns the smallest integer $y \in A$ that is $y \geq x$ (assuming, w.l.o.g., $x \leq A[n]$, so that the result is well-defined). We prove the following result based on Elias-Fano codes [Elias, 1974, Fano, 1971].

Theorem 1 (Elias-Fano) There exists a representation of A that takes at most $\mathrm{EF}(n, U) = n(\ell + 3)$ bits and:

1. For $o(n)$ extra bits, access to the i-th element, can be supported with, at most, $1 + C_{\text{sel}}$ cache misses.
2. For $o(n)$ extra bits, $\textsc{Successor}$ can be supported with, at most, $C_{\text{sel}} + C_{\text{scan}}$ cache misses.
3. For $O(n \log n)$ extra bits, $\textsc{Successor}$ can be supported with, at most, $1 + C_{\text{scan}}$ cache misses.

where

- $C_{\text{sel}} = 2 + C(O(\log^4 n))$,
- $C_{\text{scan}} = C(U/n) + C(\ell \cdot (U/n + 1)) + C(\Delta_A)$,
- $\Delta_A := \max_{1 \leq i < n} \{\lfloor \frac{A[i+1]}{2^\ell} \rfloor - \lfloor \frac{A[i]}{2^\ell} \rfloor\}$,
- and $\ell = \lfloor \log_2(U/n) \rfloor$.

Representation. The binary $\lceil \log_2(U+1) \rceil$-bit representation of each integer of A is split into two parts: its ℓ least significant bits and the remaining $h = \lceil \log_2(U+1) \rceil - \ell$ most significant bits. We call these parts the *low* and *high* parts respectively. All the n low parts are written explicitly in a vector of ℓ-bit integers, whereas the high parts are coded using a bitvector A_{high} of length $n + \lfloor U/2^\ell \rfloor + 1$ bits, which is at most $3n$ bits because $n \leq \lfloor U/2^\ell \rfloor < 2n$. The main space bound, $\mathrm{EF}(n, U) = n\lfloor \log_2(U/n) \rfloor + 3n$, follows.

The elements of A can be viewed as logically clustered into $\lfloor U/2^\ell \rfloor + 1$ clusters, $A_0, \ldots, A_{\lfloor U/2^\ell \rfloor}$, such that A_j contains the consecutive elements of A that have their high bits equal to j. The bitvector A_{high} is then

$$1^{|A_0|}0\ 1^{|A_1|}0\ 1^{|A_2|}0 \ldots 1^{|A_{\lfloor U/2^\ell \rfloor}|}0.$$

That is, it writes the cardinalities of the clusters in unary code. (Note that $|A_j|$ could be 0, i.e., no element in A has high bits equal to j. In this case, the unary code is a single 0 bit. Runs of zeros might be present in A_{high}. The length of the longest such run is Δ_A.) It follows that A_{high} has exactly n bits set.

Random access. To decode the i-th value, say $A[i] = x$, from the representation, the two parts must be re-linked together. Let x_ℓ and x_h be the low and high parts of x respectively, so that $x = x_h \cdot 2^\ell + x_\ell$. The low bits x_ℓ are read directly from the corresponding vector, spending one cache miss. The high bits x_h are computed by searching A_{high}, which can be done efficiently using $\textsc{Select}_1$ queries. A $\textsc{Select}_1(i)$ query over A_{high} returns the position of the i-th one, for $1 \leq i \leq n$. It follows that $x_h = \textsc{Select}_1(i) - i$.

The extra bits used for $\textsc{Select}_1$ as well as the number of cache misses claimed in Point 1. of Theorem 1 follow by using Theorem 2 for A_{high} ($u < 3n$ and $z = n$). We explain Theorem 2 below.

Select queries. Clark [1997] shows that $\textsc{Select}_1$ can be supported in $O(1)$ time but the data structure requires a non-trivial space usage in practice and many distinct memory accesses, resulting in several cache misses.

We describe the solution by Okanohara and Sadakane [2007], the *DArray* index, which is inspired by Clark's solution and we use in practice. (We assume $\textsc{Select}_1$ queries throughout the presentation although one can obviously flip the ones into zeros to support $\textsc{Select}_0$ as well.)

Theorem 2 (DArray) Consider a bitvector of u bits with z ones. There exists an index that takes $o(z)$ bits and supports $\textsc{Select}_1$ queries in at most $2 + C(O(\log^4 u))$ cache misses.

Let L, L_2, and L_3 be integer quantities to be fixed later. The bitvector is split into variable-length blocks, each containing L ones (except for, possibly, the last block). A block is called *sparse* if its length is larger than L_2, *dense* otherwise. Sparse blocks are represented verbatim, i.e., the positions of the L ones are coded using $\log_2(u)$-bit integers. A dense block, instead, is sparsified: we keep one 1-bit position every L_3 such positions. The positions are coded relatively to the beginning of each block, hence taking $\log_2(L_2)$ bits per position. The data structure therefore stores three arrays, I, S, D. The *inventory* array $I[1..z/L]$ is such that $I[i] := \textsc{Select}_1(iL)$ if block i is dense; otherwise, $I[i] = -p - 1$ where p is the start position in S of the 1-bit positions of block i. The space for I is therefore

$z/L \cdot \log_2(u)$ bits. The array S holds the positions of the L ones in sparse blocks. As we have at most z/L_2 sparse blocks, its space is $z/L_2 \cdot L \cdot \log_2(u)$ bits at most. Lastly, the array $D[1..z/L_3]$ is such that $D[i]$ is the position of the iL_3-th one, relative to the start position of the comprising block. Its space is $z/L_3 \cdot \log_2(L_2)$ bits.

A $\textsc{Select}_1(i)$ query, $1 \leq i \leq z$, first checks $p = I[i/L]$: if $p < 0$, then the block is sparse and the query is answered as $S[-p - 1 + (i \bmod L)]$; otherwise the position $D[i/L_3]$ is retrieved and a sequential scan of at most L_2 bits is executed starting from position $p + D[i/L_3]$. It follows that the number of cache misses per query is: 2, if i belongs to a sparse block; $2 + C(L_2)$, if i belongs to a dense block.

Choosing $L = O(\log^2 u)$, $L_2 = O(\log^4 u)$, and $L_3 = O(\log u)$, all the three arrays I, S, and D take $o(z)$ bits and the number of cache misses per query is at most $C_{\text{sel}} = 2 + C(O(\log^4 u))$. (In practice, our implementation of the Darray uses $L = 2^{10}$, $L_2 = 2^{16}$, and $L_3 = 2^5$.)

Successor. Using $\textsc{Select}_0$ queries on A_{high}, it is also possible to support the query $\textsc{Successor}(x)$. From x, we compute $x_h = \lfloor x/2^\ell \rfloor$ and $i = p - x_h$ with $p = \textsc{Select}_0(x_h)$. For $x_h > 0$, this indicates that there are i values whose high parts are *less* than x_h (when $x_h = 0$, we let $i = 0$). On the other hand, $j = \textsc{Select}_0(x_h + 1) - x_h$ gives us the position of the first element having high bits larger than x_h. Since a cluster contains at most $2^\ell \leq U/n$ elements, we have that $j - i \leq 2^\ell \leq U/n$ elements, and the successor could be determined by binary searching in the range $A[i..j]$ for a total of $O(\log(U/n)(1 + C_{\text{sel}}))$ cache misses. This algorithm is not, however, cache-efficient. It is better in practice to answer the query by scanning A from the $(i+1)$-th element. We follow this latter approach as it matches our implementation. (It relies on the fact that Δ_A is small for practical applications of Elias-Fano, like SSHash, albeit $\Delta_A = O(n)$ in the worst case.) When scanning from the $(i + 1)$-th element, the following cases can happen:

1. The bit in position $p+1$ of A_{high} is 0: then cluster $x_h + 1$ is empty and the successor of x is $A[i + 1]$ (minimum element in the next non-empty cluster). The low bits of $A[i + 1]$ are retrieved with 1 cache miss, whereas the high bits are computed by scanning A_{high} from position $p + 1$ until the next bit set. Since the longest run of zeros in A_{high} has length Δ_A, $C(\Delta_A)$ cache misses are issued during the scan.
2. The bit in position $p + 1$ of A_{high} is 1, so the cluster is not empty. The elements in the cluster all have the same high bits x_h. Now, two cases can happen:
   a. The successor is not larger than the largest element in the cluster, so it belongs to the cluster. Scanning up to U/n elements therefore costs $C(U/n) + C(\ell \cdot U/n)$ cache misses.
   b. The successor is larger than the largest element in the cluster, so it is the minimum in the next non-empty cluster. The cost is at most $C(U/n) + C(\ell \cdot (U/n + 1)) + C(\Delta_A)$.

Using again Theorem 2 on the zeros of A_{high} ($u < 3n$ and $z = 2n$) to implement $\textsc{Select}_0$, the extra bits and number of cache misses claimed in Point 2. of Theorem 1 follow.

Lastly, Point 3. of Theorem 1 illustrates a more space-consuming alternative that, on the other hand, supports faster $\textsc{Successor}$. The idea is to use an extra array $hints[1..\lfloor U/2^\ell \rfloor]$ such that $hints[i] = \textsc{Select}_0(i)$, for $i = 1..\lfloor U/2^\ell \rfloor$. As $n \leq \lfloor U/2^\ell \rfloor < 2n - 1$ and $|A_{\text{high}}| < 3n$, the space bound follows. Instead of computing $\textsc{Select}_0(x_h)$, this value is readily available as $hints[x_h]$ (in the general case when $x_h > 0$).

Table 1. Number of cache misses: theory and practice. These results are for $k = 31$ and SSHash regular, for random positive $\textsc{Lookup}$ queries.

(a) Theorem 2

	Cod	Human	HPRC
Case 1, theory	43.9	58.1	149.9
Case 1, practice	32.8	49.2	136.2
Case 2, theory	20.0	23.5	23.5
Case 2, practice	16.1	22.5	21.5

(b) Theorem 3

	Cod	Human	HPRC
Case 1, theory	7	7	7
Case 1, practice	7.05	6.1	6.8
Case 2, theory	152.2	152.9	135.5
Case 2, practice	143.6	147.1	132.2
Case 3, theory	10	10	10
Case 3, practice	10.5	8.7	10

2. Cache miss analysis: theory and practice

As discussed in Section 2 of the main paper, we model the number of cache misses involved during a read of Q bits from main memory to the cache with $C(Q) := \lceil Q/B \rceil$, where B is the cache line size. In this section we validate this model and show that it is accurate under proper tuning. In particular, we compare the number of theoretical cache misses of $\textsc{Lookup}$ claimed in Theorem 2 and Theorem 3 of the main paper with the *actual* number of cache misses measured using the Linux `perf` tool (command: `perf stat -B -e cache-misses`).

For ease of presentation, we report again below the number of theoretical cache misses from Theorem 2 and Theorem 3 of the main paper. Both are valid for a $\textsc{Lookup}(x)$ query, with $z = |\text{loc}(\textsc{Mini}(x))|$.

Theorem 2: previous SSHash. The number of cache misses is at most

1. $1 + C_{\text{acc}} + C(z \log_2(N)) + z(C_{\text{succ}} + C(4k - 2m))$ if $1 \leq z \leq 2^l$;
2. $4 + C_{\text{acc}} + C_{\text{succ}} + C(4k - 2m)$ otherwise.

Theorem 3: current SSHash. The number of cache misses is at most

1. $2 + C'_{\text{succ}} + C(2k)$ if $z = 1$;
2. $3 + C(z \log_2(N)) + z(C'_{\text{succ}} + C(2k))$ if $2 \leq z \leq 2^l$;
3. $5 + C'_{\text{succ}} + C(2k)$ otherwise.

The values of C_{acc}, C_{succ}, C'_{succ}, and C_{scan} are:

- $C_{\text{acc}} = 3 + C(O(\log^4 M))$,
- $C_{\text{succ}} = 2 + C(O(\log^4 |\mathcal{S}|)) + C_{\text{scan}}$,
- $C'_{\text{succ}} = 1 + C_{\text{scan}}$,
- $C_{\text{scan}} = C(N/|\mathcal{S}|) + C((N/|\mathcal{S}| + 1) \log_2(N/|\mathcal{S}|)) + C(\Delta_P)$.

Fixing the parameters and result. We fix $B = 512$, which is a very common cache line size and, indeed, that of our testing machine. For the choice of k and m in our experimental analysis (Section 8 of the main paper), we have $C(2k) = C(4k - 2m) = 1$ for $k = 31$ and $m < k$. Although the longest run of zeros on the high bitvector of the Elias-Fano can can be as large as $O(n)$ in the worst case, Δ_P is actually small on tested datasets. For example, it is 140 on the whole human genome. So, we let $C_{\text{scan}} = 3$. We let $C(O(\log^4 M)) = 6 \cdot C(O(\log^4 |\mathcal{S}|))$ for Theorem 2 because, in practice, the *sizes* array is approximately 6 times larger than the P array. We use $C(O(\log^4 |\mathcal{S}|)) = 1$ for Cod but $C(O(\log^4 |\mathcal{S}|)) = 1.5$ for both Human and HPRC as their respective indexes are much larger than that for Cod. For the value of z we use the average number of positions j inspected by 10^6 random positive LOOKUP queries. Lastly, we have $\log_2(N)$ equal to 30, 32, and 34 for Cod, Human, and HPRC respectively.

Table 1 reports the result of the comparison: for every case and dataset, the model closely matches the actual number of cache misses.

3. Construction

We describe a multi-threaded construction algorithm for the new layout of SSHash described in Section 6 of the main paper, designed to scale to large collections using external memory and a fixed RAM budget. The construction takes as input a compressed collection of strings (in FASTA format and compressed, for example with `gzip`) and proceeds as a pipeline of streaming and sorting phases. In short: minimizers are first generated by streaming through the strings, in parallel; then, sorted in external memory; and finally laid out contiguously. The construction steps are as follows.

1. Input encoding. Each string in the input is decompressed incrementally, 2-bit encoded using SIMD instructions, and concatenated in S.

2. Parallel minimizer computation. The obtained string S is split into chunks, one chunk per thread. The RAM dedicated to the construction is split evenly among threads. Each thread computes the minimizers in its chunk in a *streaming* fashion. We use the folklore *re-scan* method which performs better in practice than the monotone-queue approach we used before (see the discussion in [Zheng et al., 2025]). For each minimizer occurrence, the thread emits the tuple (ϕ, j, p, v) into a thread-local in-memory buffer. The tuple comprises: the minimizer itself ϕ (as a 2-bit encoded string), its occurrence j in S, the offset p indicating that the super-k-mer of minimizer ϕ starts in S at position $j - p + 1$, and lastly v, the number of k-mers in its super-k-mer. (These last two quantities, p and v, are used to build efficiently the skew index in the last step.) When the buffer reaches its dedicated capacity, the tuples in the buffer are sorted by the components (ϕ, j) and the buffer is flushed to disk as a sorted run.

3. External merge using a winner tree. The sorted runs on disk are merged into a single run using a classic multi-way external merge algorithm. However, instead of a min-heap, a *winner tree* is used to select the minimum element at each step of the merge. A winner tree is a complete binary-tree, like a min-heap, but it performs only one comparison per tree level when updating the minimum (compared to two, as spent by a min-heap), yielding a ≈30% speedup in our experiments. Knuth [1998] (Section 5.4.1) gives a description of tournament trees.

Table 2. Distinct k-mers and minimizer lengths (m) for SSHash.

Collection	$k = 31$; m	$k = 63$; m
Cod	502,465,200; 20	556,585,658; 24
Kestrel	1,150,399,205; 20	1,155,250,667; 24
Human	2,505,678,680; 21	2,771,316,093; 25
NCBI-v	376,205,185; 19	412,515,880; 23
SE	894,310,084; 21	1,524,904,156; 31
HPRC	3,718,120,949; 21	5,926,785,469; 31

4. MPHF construction. From the merged minimizer stream on disk, we build the MPHF f using external memory and multiple threads. Our implementation uses PTHash as choice of MPHF [Pibiri and Trani, 2021, 2023].

5. Resorting tuples in MPHF order. The minimizer tuples on disk are sorted again according to the identifier assigned to minimizers by f. As a result, all occurrences of the same minimizer become contiguous in a file on disk. This process is implemented, again, with a parallel external-memory merge sort.

6. Locate sets construction. Since minimizer tuples are now laid out consecutively on disk, the arrays T, G, L, and H are all computed by scanning the tuples sequentially.

7. Skew index construction. During the scan, minimizers occurring more than 2^l times are detected and the $r - l + 1$ partitions are built one after the other. Consider a tuple (ϕ, j, p, v) such that $2^i < |\text{loc}(\phi)| \leq 2^{i+1}$ for some $i \geq l$. All the k-mers $x \in S[j - p - 1..j - p + v + k)$ are added to the set K_i under formation. As soon as the next processed minimizer has a locate set larger than 2^{i+1}, the MPHF f_i is built (in parallel) for K_i, V_i laid out consequently, and the process continues with the next partition.

4. Parameter sweep

The performance of SSHash is dependent on the selection of its structural parameters. The space-time tradeoff is governed by two main parameters: the minimizer length m and the skew index threshold l.

- **Minimizer length, m.** This parameter dictates the fundamental space-time tradeoff of the index. A smaller m optimizes index space but results in larger locate sets (since more k-mers share the same minimizer), which in turn makes query resolution slower. Conversely, a larger m makes queries significantly faster by reducing the size of the locate sets, but consumes more space due to indexing a larger number of distinct minimizers.
- **Skew index threshold, l.** This parameter handles highly abundant minimizers. Any k-mer whose minimizer appears more than 2^l times is routed to the skew index. Choosing a small l accelerates queries because a larger fraction of difficult, high-frequency k-mers are solved quickly via the skew index. However, populating the skew index more heavily naturally consumes more memory space.

Varying the minimizer length. Based on the m-sweep benchmark plots in Figure 1, the impact of the minimizer length across the Human and SE datasets is highly visible. For $k = 31$ (represented by circles), we observe a distinct downward-sloping curve. As m increases from 17 to 25 (or up to 31 for SE), the lookup time drops drastically while the index space grows steadily. For $k = 63$ (squares), increasing m yields

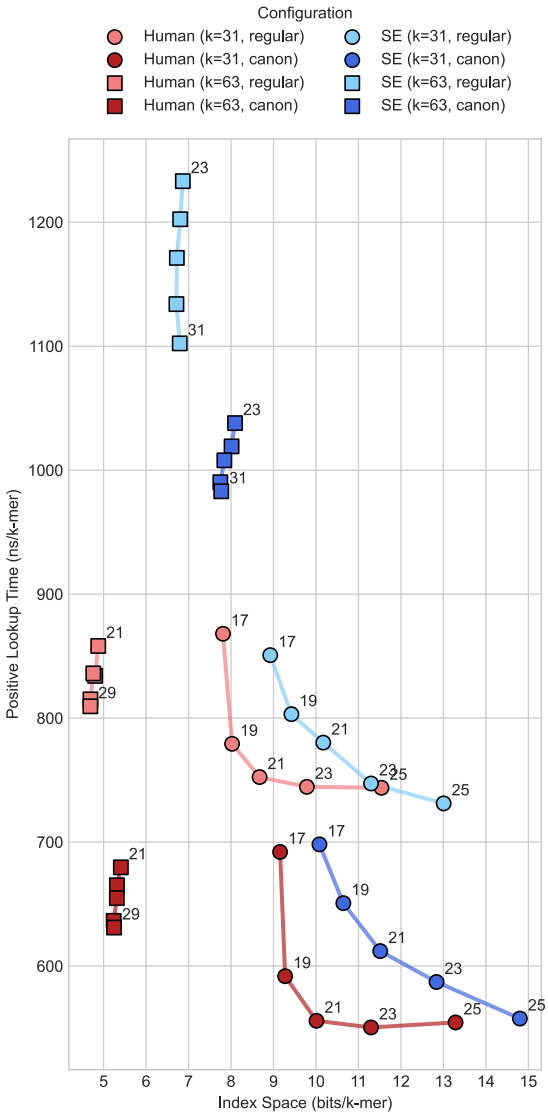

Fig. 1. Space-time tradeoff of SSHash when varying the minimizer length m across Human and SE datasets.

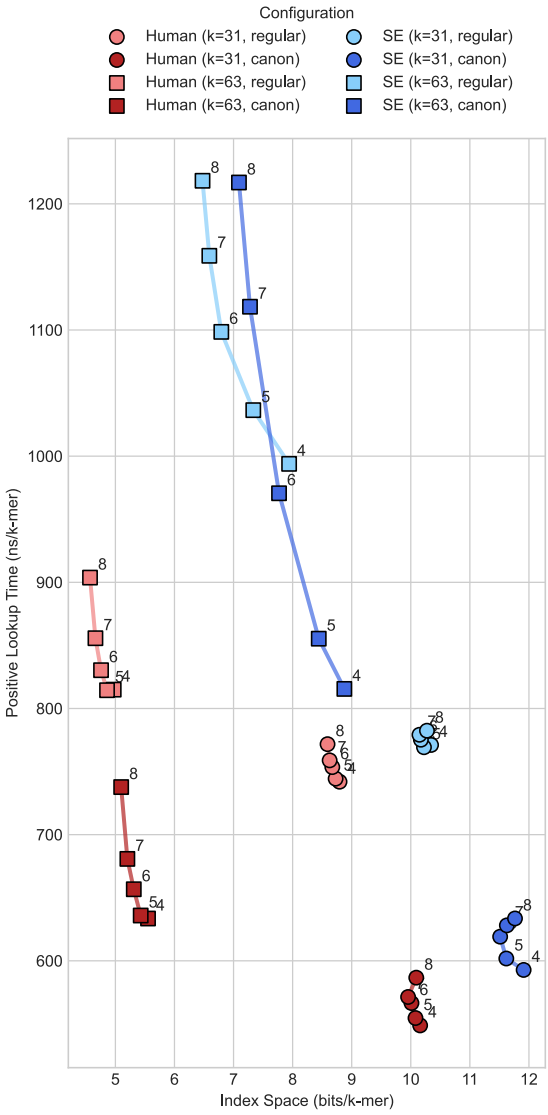

Fig. 2. Space-time tradeoff of SSHash when varying the skew index threshold l across Human and SE datasets.

a steep drop in query time with only a marginal penalty in space, making larger m values highly attractive.

Varying the skew index threshold. The l-sweep plots in Figure 2 isolate the effect of the skew index threshold, revealing how managing high-frequency minimizers impacts performance. As expected, when l decreases from 8 down to 4, there is a clear visual migration down and to the right for every single configuration. Query time improves as more k-mers are offloaded to the fast skew index, at the direct cost of increased bits/k-mer. For some configurations however, notably the Human dataset at $k = 31$ (canonical), dropping l from 5 to 4 yields almost no improvement in query time but still incurs a space penalty. This indicates that a good threshold is $l = 5$ or $l = 6$ for this specific case. The SE dataset with $k = 63$ (regular and canonical) shows a massive vertical spread. For these configurations, relying more heavily on the skew index (smaller l) is highly effective at driving down query times from over 1200 to under 1000 ns/k-mer.

5. Streaming queries

Figure 3 shows the performance for streaming Lookup queries on three different scenarios: "high-hit" (most k-mers are found in the indexes), "low-hit" (most k-mers are *not* found), and a "mixed" workload where the previous query workloads are mixed 50-50%. As explained in Section 8 of the main paper, the indexes stream through FASTQ reads and each read set contains several million reads. All reads are available at https://zenodo.org/records/17582116 for reproducibility.

High-hit workloads are relevant to assess the capability of the indexes to take advantage of consecutive query k-mers, whereas low-hit workloads mostly test the ability to reject alien k-mers. Since both cases are relevant in practice, the mixed-hit workload alternates between the two cases.

Overall, all evaluated indexes exhibit robust query times, delivering consistent and predictable performance across different workloads without significant variation. Across all benchmarks, SSHash offers the fastest query time though, being

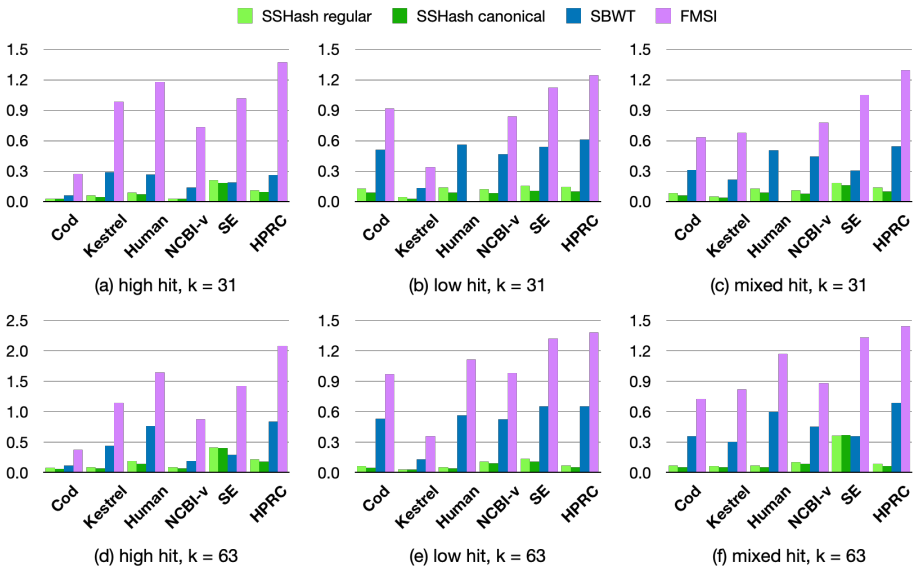

Fig. 3. Streaming query efficiency in avg. $\mu s/k$-mer. (In plot (b), FMSI is not reported for the Human dataset because the corresponding implementation generated a segmentation fault.)

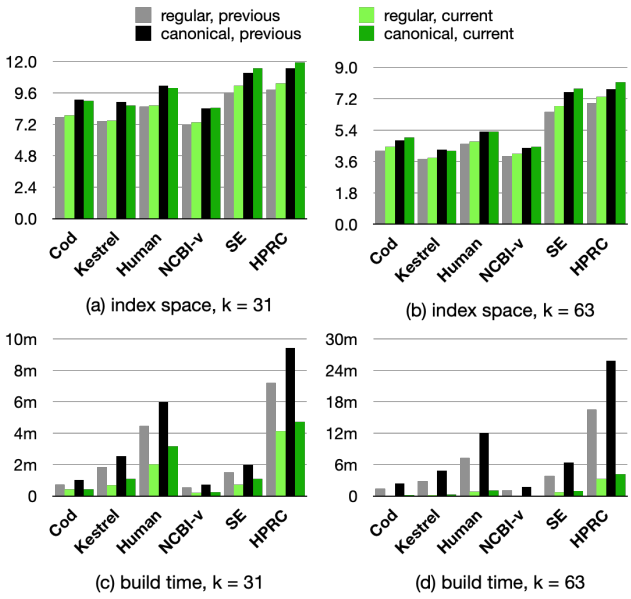

Fig. 4. Comparison between previous and current SSHash: index space (in avg. bits/k-mer) and build time.

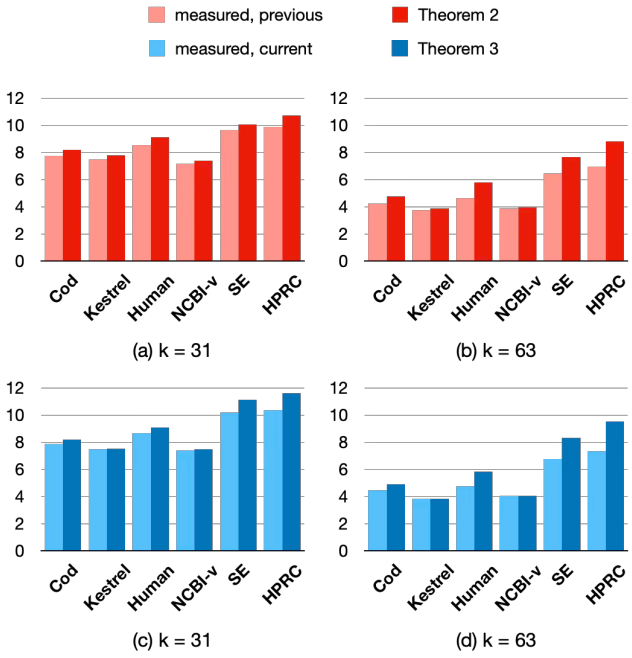

Fig. 5. Comparison between measured index space and that from Theorem 2 and Theorem 3 of the main paper. Space is reported in avg. bits/k-mer. The constants in the asymptotic terms $\Theta(\alpha)$ and $\Theta(M)$ are the same from both theorems and equal to 2.5 and 3.0 respectively, which are faithful to our implementation.

several times faster than its competitors. The only exception is the SE datasets (high and mixed workloads), where the SBWT gives competitive runtimes.

6. Experimental comparison against the previous version

We compare the version of SSHash from this work (referred to as *current* in the plots) and the previous version[1], using the same datasets (see Table 2), machine, and methodology described in Section 8 of the main paper. In general, the current version outperforms the previous one under every aspect and consistently on all tested dataset.

Figure 4 illustrates the space and build time of the two versions. The space is very similar between the two versions. Furthermore, Figure 5 shows the comparison between the actual, measured, space and the bounds from Theorem 2 and Theorem 3 in the main paper. In both cases, the bounds are

[1] GitHub commit: `a2a2d26817fe3f476ceac44809c333ede6622ff3`.

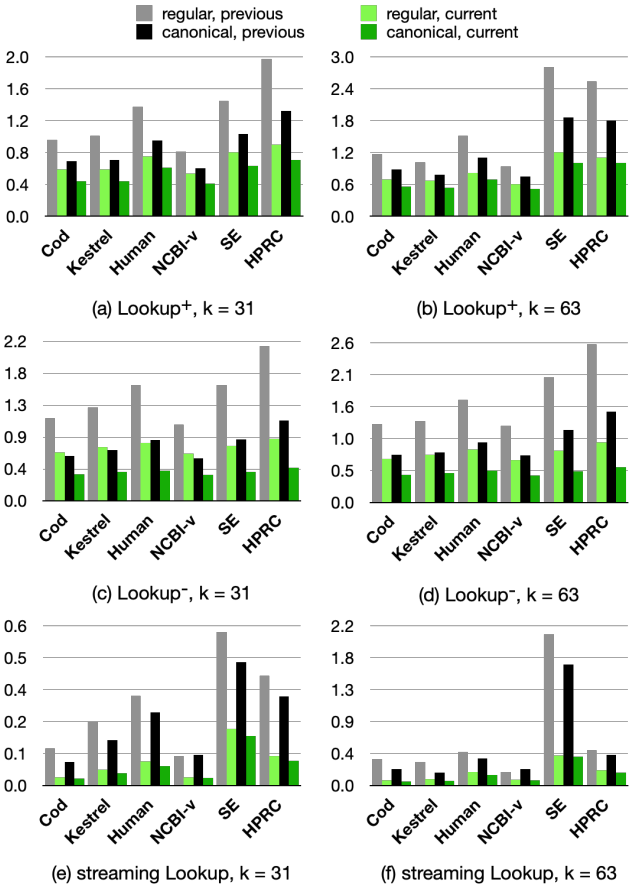

Fig. 6. Comparison between previous and current SSHash: query times are reported in avg. $\mu s/k$-mer.

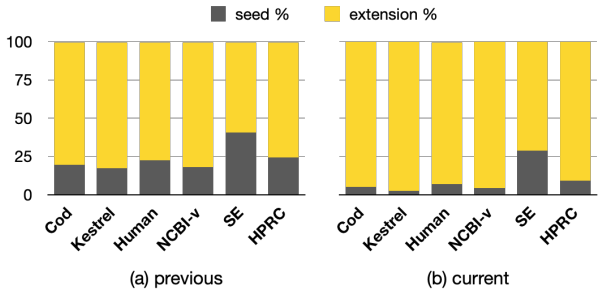

Fig. 7. Seed vs. extension rate for streaming LOOKUP queries, for $k = 31$.

quite close to the measured space and most of the difference comes from overestimating the cost of the skew index with $\alpha(\log_2(N) + \Theta(1))$ bits.

The current version is between $2 - 3\times$ faster to build, on average for $k = 31$. This result improves for larger k; for example, it is up to $6\times$ faster for $k = 63$. The better build time is due to better multi-threading, faster minimizer computations over streams, and faster merging in external memory which the previous version supported only partially.

Figure 6, instead, shows the query times of the two versions. (Query time for ACCESS is almost the same between the two versions, as apparent from the tables in this document, so we do not discuss it in the following.) Avoiding the scan of super-k-mers and the cache-efficient layout both contribute to faster random LOOKUP queries. The simpler logic of the streaming LOOKUP algorithm paired with the more efficient random LOOKUP, results in $2 - 3\times$ faster streaming queries. In particular, the refined logic consistently increases the extension rate compared to the previous version, of about 15% for $k = 31$ (Figure 7).

References

D. Clark. *Compact pat trees*. PhD thesis, University of Waterloo, 1997.

P. Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, 1974.

R. M. Fano. On the number of bits required to implement an associative memory. *Memorandum 61, Computer Structures Group, MIT*, 1971.

D. E. Knuth. *The Art of Computer Programming: Sorting and Searching, Volume 3*. 1998.

D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *ALENEX*, pages 60–70, 2007.

G. E. Pibiri and R. Trani. PTHash: Revisiting FCH minimal perfect hashing. In *SIGIR*, pages 1339–1348, 2021.

G. E. Pibiri and R. Trani. Parallel and external-memory construction of minimal perfect hash functions with PTHash. *Transactions on Knowledge and Data Engineering*, 36(3):1249–1259, 2023.

A. Zheng, I. Lee, V. S. Shivakumar, O. Y. Ahmed, and B. Langmead. Fast and flexible minimizer digestion with digest. *Bioinformatics*, 41(7), 2025.

Table 3. Index space and construction efficiency for current SSHash.

(a) regular

k	Collection	bits/k-mer	GB	build time
31	Cod	7.89	0.50	26s
	Kestrel	7.50	1.08	43s
	Human	8.67	2.72	2m 1s
	NCBI-v	7.37	0.35	13s
	SE	10.17	1.14	45s
	HPRC	10.35	4.81	4m 8s
63	Cod	4.44	0.31	12s
	Kestrel	3.82	0.55	16s
	Human	4.76	1.65	54s
	NCBI-v	4.05	0.21	5s
	SE	6.79	1.29	44s
	HPRC	7.33	5.43	3m 20s

(b) canonical

k	Collection	bits/k-mer	GB	build time
31	Cod	9.01	0.57	26s
	Kestrel	8.67	1.25	1m 6s
	Human	10.01	3.14	3m 10s
	NCBI-v	8.48	0.40	16s
	SE	11.51	1.29	1m 6s
	HPRC	11.93	5.54	4m 45s
63	Cod	4.97	0.35	15s
	Kestrel	4.22	0.61	19s
	Human	5.31	1.84	1m 9s
	NCBI-v	4.46	0.23	7s
	SE	7.77	1.48	58s
	HPRC	8.14	6.03	4m 13s

Table 4. Query efficiency for current SSHash. Timings for Lookup and Access are in avg. microseconds per k-mer. For Streaming (high-hit), we report avg. nanoseconds per k-mer.

(a) regular

k	Collection	Lookup$^+$	Lookup$^-$	Access	Streaming
31	Cod	0.59	0.67	0.28	30
	Kestrel	0.59	0.74	0.28	60
	Human	0.75	0.80	0.36	90
	NCBI-v	0.54	0.65	0.26	30
	SE	0.80	0.76	0.36	213
	HPRC	0.90	0.86	0.54	112
63	Cod	0.69	0.71	0.29	77
	Kestrel	0.67	0.78	0.33	86
	Human	0.82	0.86	0.36	188
	NCBI-v	0.61	0.69	0.28	85
	SE	1.20	0.85	0.41	412
	HPRC	1.10	0.98	0.64	213

(b) canonical

k	Collection	Lookup$^+$	Lookup$^-$	Access	Streaming
31	Cod	0.44	0.37	0.28	26
	Kestrel	0.44	0.40	0.28	46
	Human	0.61	0.42	0.35	74
	NCBI-v	0.41	0.36	0.26	29
	SE	0.63	0.40	0.36	186
	HPRC	0.71	0.46	0.54	93
63	Cod	0.56	0.45	0.29	60
	Kestrel	0.54	0.48	0.33	66
	Human	0.69	0.52	0.36	146
	NCBI-v	0.52	0.44	0.28	72
	SE	1.00	0.51	0.41	400
	HPRC	1.00	0.58	0.64	181

Table 5. Index space and construction efficiency for previous SSHash.

(a) regular

k	Collection	bits/k-mer	GB	build time
31	Cod	7.75	0.49	43s
	Kestrel	7.47	1.07	1m 50s
	Human	8.56	2.68	4m 27s
	NCBI-v	7.17	0.34	33s
	SE	9.65	1.08	1m 30s
	HPRC	9.88	4.59	7m 14s
63	Cod	4.23	0.29	1m 28s
	Kestrel	3.76	0.54	2m 56s
	Human	4.63	1.60	7m 20s
	NCBI-v	3.90	0.20	1m 4s
	SE	6.46	1.23	3m 51s
	HPRC	6.94	5.14	16m 32s

(b) canonical

k	Collection	bits/k-mer	GB	build time
31	Cod	9.11	0.57	1m 1s
	Kestrel	8.93	1.28	2m 34s
	Human	10.15	3.18	6m
	NCBI-v	8.44	0.40	44s
	SE	11.15	1.25	1m 59s
	HPRC	11.50	5.35	9m 26s
63	Cod	4.81	0.33	2m 22s
	Kestrel	4.28	0.62	4m 55s
	Human	5.30	1.83	12m 5s
	NCBI-v	4.37	0.23	1m 45s
	SE	7.59	1.45	6m 25s
	HPRC	7.76	5.75	25m 53s

Table 6. Query efficiency for previous SSHash. Timings for Lookup and Access are in avg. microseconds per k-mer. For Streaming (high-hit), we report avg. nanoseconds per k-mer.

(a) regular

k	Collection	Lookup$^+$	Lookup$^-$	Access	Streaming
31	Cod	0.96	1.14	0.29	140
	Kestrel	1.00	1.29	0.26	239
	Human	1.37	1.60	0.37	337
	NCBI-v	0.81	1.05	0.29	111
	SE	1.45	1.60	0.39	578
	HPRC	1.97	2.14	0.60	412
63	Cod	1.18	1.30	0.31	363
	Kestrel	1.00	1.40	0.27	329
	Human	1.55	1.70	0.36	461
	NCBI-v	0.94	1.27	0.30	184
	SE	2.81	2.04	0.44	2084
	HPRC	2.54	2.58	0.68	488

(b) canonical

k	Collection	Lookup$^+$	Lookup$^-$	Access	Streaming
31	Cod	0.69	0.62	0.29	89
	Kestrel	0.71	0.70	0.27	172
	Human	0.95	0.84	0.37	276
	NCBI-v	0.60	0.59	0.28	117
	SE	1.03	0.85	0.39	465
	HPRC	1.32	1.11	0.60	335
63	Cod	0.88	0.78	0.31	228
	Kestrel	0.78	0.82	0.27	180
	Human	1.11	0.98	0.36	371
	NCBI-v	0.75	0.77	0.30	229
	SE	1.86	1.18	0.44	1671
	HPRC	1.81	1.48	0.68	425