# Spectrum Preserving Tilings Enable Sparse and Modular Reference Indexing

Jason Fan[1], Jamshed Khan[1], Giulio Ermanno Pibiri[2,3], and Rob Patro[1(✉)]

[1] University of Maryland, College Park, MD 20440, USA
jasonfan@umd.edu, {jamshed,rob}@cs.umd.edu
[2] Ca' Foscari University of Venice, Venice, Italy
giulioermanno.pibiri@unive.it
[3] ISTI-CNR, Pisa, Italy

**Abstract.** The reference indexing problem for $k$-mers is to pre-process a collection of reference genomic sequences $\mathcal{R}$ so that the position of all occurrences of any queried $k$-mer can be rapidly identified. An efficient and scalable solution to this problem is fundamental for many tasks in bioinformatics.

In this work, we introduce the *spectrum preserving tiling* (SPT), a general representation of $\mathcal{R}$ that specifies how a set of *tiles* repeatedly occur to spell out the constituent reference sequences in $\mathcal{R}$. By encoding the order and positions where *tiles* occur, SPTs enable the implementation and analysis of a general class of modular indexes. An index over an SPT decomposes the reference indexing problem for $k$-mers into: (1) a $k$-mer-to-tile mapping; and (2) a tile-to-occurrence mapping. Recently introduced work to construct and compactly index $k$-mer sets can be used to efficiently implement the $k$-mer-to-tile mapping. However, implementing the tile-to-occurrence mapping remains prohibitively costly in terms of space. As reference collections become large, the space requirements of the tile-to-occurrence mapping dominates that of the $k$-mer-to-tile mapping since the former depends on the amount of total sequence while the latter depends on the number of unique $k$-mers in $\mathcal{R}$.

To address this, we introduce a class of sampling schemes for SPTs that trade off speed to reduce the size of the tile-to-reference mapping. We implement a practical index with these sampling schemes in the tool `pufferfish2`. When indexing over 30,000 bacterial genomes, `pufferfish2` reduces the size of the tile-to-occurrence mapping from 86.3 GB to 34.6 GB while incurring only a 3.6× slowdown when querying $k$-mers from a sequenced readset.

**Availability:** `pufferfish2` is implemented in Rust and available at https://github.com/COMBINE-lab/pufferfish2.

**Supplementary materials:** S.1 to S.8 available online at https://doi.org/10.5281/zenodo.7504717.

**Supplementary Information** The online version contains supplementary material available at https://doi.org/10.1007/978-3-031-29119-7_2.


H. Tang (Ed.): RECOMB 2023, LNBI 13976, pp. 21–40, 2023.
https://doi.org/10.1007/978-3-031-29119-7_2

**Keywords:** Reference Indexing · Spectrum Preserving Tilings · Minimal Perfect Hashing · Pufferfish2

## 1 Introduction

Indexing of genomic sequences is an important problem in modern computational genomics, as it enables the atomic queries required for analysis of sequencing data—particularly *reference guided* analyses where observed sequencing data is compared to known *reference* sequences. Fundamentally, analyses need to first rapidly locate short exact matches to reference sequences before performing other operations downstream. For example, for guided assembly of genomes, variant calling, and structural variant identification, seed sequences are matched to known references before novel sequences are arranged according to the seeds [1]. For RNA-seq, statistics for groups of related $k$-mers mapping to known transcripts or genes allow algorithms to infer the activity of genes in single-cell and bulk gene-expression analyses [2–4].

Recently, researchers have been interested in indexing collections of genomes for metagenomic and pan-genomic analyses. There have been two main types of approaches: full-text indexes, and hashing based approaches that typically index the *de Bruijn graph* (dBG). With respect to full-text indexes, researchers have developed tools that use the *r-index* [5] to compute matching statistics and locate maximal exact matches for large reference collections [6,7]. For highly repetitive collections, such as many genomes from the same species, r-index based approaches are especially space efficient since they scale linearly to the number of runs in the *Burrows-Wheeler Transform* (BWT) [8] and not the length of the reference text. With respect to hashing based approaches, tools restrict queries to fixed length $k$-mers [1,9] and index the dBG. These tools achieve faster exact queries but typically trade off space. In other related work, graph-based indexes that compactly represent genomic variations as paths on graphs have also been developed [10,11]. However, these indexes require additional work to project queries landing on graph-based coordinates to linear coordinates on reference sequences.

Many tools have been developed to efficiently build and represent the dBG [12,13]. Recently, Khan et al. introduced a pair of methods to construct the compacted dBG from both assembled references [14] and read sets [15]. Ekim et al. [16] introduced the minimizer-space dBG –a highly effective lossy compression scheme that uses minimizers as representative sequences for nodes in the dBG. Karasikov et al. developed the Counting dBG [17] that stores differences between adjacent nodes in the dBG to compress metadata associated with nodes (and sequences) in a dBG. Encouragingly, much recent work on *Spectrum Preserving String Sets* (SPSS) that compactly index the set-membership of $k$-mers in reference texts has been introduced [15,18–23]. Although these approaches do not tackle the *locate* queries directly, they do suggest that even more efficient solutions for reference indexing are possible.

In this work, we extend these recent ideas and introduce the concept of a *Spectrum Preserving Tiling* (SPT) which encodes how and where $k$-mers in

an SPSS occur in a reference text. In introducing the SPT, this work makes two key observations. First, a hashing based solution to the reference indexing problem for $k$-mers does not necessitate a de Bruijn graph but instead requires a *tiling* over the input reference collection—the SPT formalizes this. Second, the reference indexing problem for $k$-mers queries can be cleanly decomposed into a *$k$-mer-to-tile* query and a *tile-to-occurrence* query. Crucially, SPTs enable the implementation and analysis of a general class of modular indexes that can exploit efficient implementations introduced in prior work.

**Contributions.** We focus our work on considering how indexes can, *in practice*, efficiently support the two composable queries—the *$k$-mer-to-tile* query and the *tile-to-occurrence* query. We highlight this work's key contributions below. We introduce:

1. The *spectrum preserving tiling* (SPT). An SPT is a general representation that explicitly encodes how shared sequences—*tiles*—repeatedly occur in a reference collection. The SPT enables an entire *class* of sparse and modular indexes that support exact locate queries for $k$-mers.
2. An algorithm for sampling and compressing an indexed SPT built from unitigs that *samples* unitig-occurrences. For some small constant "sampling rate", $s$, our algorithm stores the positions of only $\approx 1/s$ occurrences and encodes all remaining occurrences using a small *constant* number of bits.
3. `Pufferfish2`: a practical index and implementation of the introduced sampling scheme. We highlight the critical engineering considerations that make `pufferfish2` effective in practice.

## 2 Problem Definition and Preliminaries

**The Mapped Reference Position (MRP) Query.** In this work we consider the *reference indexing problem for $k$-mers*. Given a collection of references $\mathcal{R} = \{R_1, \ldots, R_N\}$, where each reference is a string over the DNA alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{T}, \mathtt{G}\}$, we seek an index that can efficiently compute the *mapped reference position* (MRP) query for a fixed $k$-mer size $k$. Given any $k$-mer $x$, the MRP query enumerates the positions of all occurrences of $x$ in $\mathcal{R}$. Precisely, each returned occurrence is a tuple $(n, p)$ that specifies that $k$-mer, $x$, occurs in reference $n$ at position $p$ where $R_n[p : p + k] = x$. If a $k$-mer does not occur in some $R_n \in \mathcal{R}$, the MRP query returns an empty list.

**Basic Notation.** Strings and lists are zero-indexed. The length of a sequence $S$ is denoted $|S|$. The $i$-th character of a string $S$ is $S[i]$. A $k$-mer is a string of length $k$. A sub-string of length $\ell$ in the string $S$ starting at position $i$ is notated $S[i : i + \ell]$. The prefix and suffix of length $i$ is denoted $S[: i]$ and $S[|S| - i :]$, respectively. The concatenation of strings $A$ and $B$ is denoted $A \circ B$.

We define the *glue* operation, $A \oplus_k B$, to be valid for any pair of strings $A$ and $B$ that overlap by $(k-1)$ characters. If the $(k-1)$-length suffix of $A$ is equal to the $(k-1)$-length prefix of $B$, then $A \oplus_k B \coloneqq A \circ B[(k-1) :]$. When $k$ clear from context, we write $A \oplus B$ in place of $A \oplus_k B$.

**Rank and Select Queries over Sequences.** Given a sequence $S$, the *rank* query given a character $\alpha$ and position $i$, written $\texttt{rank}_\alpha(S, i)$, is the number of occurrences of $\alpha$ in $S[:i]$ The *select* query $\texttt{select}_\alpha(S, r)$ returns the position of the $r$-th occurrence of symbol $\alpha$ in $S$. The *access* query $\texttt{access}(S, i)$ returns $S[i]$. For a sequence of length $n$ over an alphabet of size $\sigma$, these can be computed in $O(\lg \sigma)$ time using a *wavelet matrix* that requires $n \lg \sigma + o(n \lg \sigma)$ bits [24].

## 3 Spectrum Preserving Tilings

In this section, we introduce the *spectrum preserving tiling*, a representation of a given reference collection $\mathcal{R}$ that specifies how a set of *tiles* containing $k$-mers repeatedly occur to spell out the constituent reference sequences in $\mathcal{R}$. This alternative representation enables a modular solution to the reference indexing problem, based on the interplay between two mappings—a $k$-mer-to-tile mapping and a tile-to-occurrence mapping.

### 3.1 Definition

Given a $k$-mer length $k$ and an input reference collection of genomic sequences $\mathcal{R} = \{R_1, \ldots, R_N\}$, a spectrum preserving tiling (SPT) for $\mathcal{R}$ is a five-tuple $\Gamma := (\mathcal{U}, \mathcal{T}, \mathcal{S}, \mathcal{W}, \mathcal{L})$:

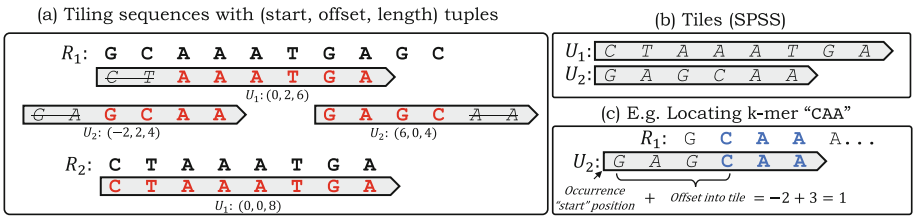


**Fig. 1.** (a) A spectrum preserving tiling (SPT) with $k = 3$, (b) with tiles (an SPSS) that contain all $k$-mers in references. (c) The SPT explicitly encodes where each $k$-mer occurs.

- **Tiles:** $\mathcal{U} = \{U_1, \ldots, U_F\}$. The set of *tiles* is a spectrum preserving string set, i.e., a set of strings such that each $k$-mer in $\mathcal{R}$ occurs in some $U_i \in \mathcal{R}$. Each string $U_i \in \mathcal{U}$ is called a *tile*.
- **Tiling sequences:** $\mathcal{T} = \{T_1, \ldots, T_N\}$ where each $T_n$ corresponds to each reference $R_n \in \mathcal{R}$. Each tiling sequence is an ordered sequence of tiles $T_n = [T_{n,1}, \ldots, T_{n,M_n}]$, of length $M_n$, with each $T_{n,m} = U_i \in \mathcal{U}$. We term each $T_{n,m}$ a *tile-occurrence*.
- **Tile-occurrence lengths:** $\mathcal{L} = \{L_1, \ldots, L_N\}$, where each $L_n = [l_{n,1}, \ldots, l_{n,M_n}]$ is a sequence of lengths.
- **Tile-occurrence offsets:** $\mathcal{W} = \{W_1, \ldots, W_N\}$, where each $W_n = [w_{n,1}, \ldots, w_{n,M_n}]$ is an integer-sequence.

- **Tile-occurrence start positions:** $\mathcal{S} = \{S_1, \ldots, S_N\}$, where each $S_n = [s_{n,1}, \ldots, s_{n,M_n}]$ is an integer-sequence.

A valid SPT must satisfy the *spectrum preserving tiling property*, that every reference sequence $R_n$ can be reconstructed by gluing together *substrings of tiles* at offsets $W_n$ with lengths $L_n$:

$$R_n = T_{n,1}[w_{n,1} : w_{n,1} + l_{n,1}] \oplus \ldots \oplus T_{n,M_n}[w_{n,M_n} : w_{n,M_n} + l_{n,M_n}].$$

Specifically, the SPT encodes how redundant sequences—*tiles*—repeatedly occur in the reference collection $\mathcal{R}$. We illustrate how an ordered sequence of start-positions, offsets, and lengths explicitly specify how redundant sequences tile a pair of references in Fig. 1. More succinctly, each tile-occurrence $T_{n,m}$ with length $l_{n,m}$ tiles the reference sequence $R_n$ as:

$$R_n[s_{n,m} + w_{n,m} : s_{n,m} + w_{n,m} + l_{n,m}] = T_{n,m}[w_{n,m} : w_{n,m} + l_{n,m}].$$

In the same way a small SPSS compactly determines the *presence* of a $k$-mer, a small SPT compactly specifies the *location* of a $k$-mer. For this work, we consider SPTs where any $k$-mer occurs only once in the set of tiles $\mathcal{U}$. The algorithms and ideas introduced in this paper still work with SPTs where a $k$-mer may occur more than once in $\mathcal{U}$ (some extra book-keeping of a one-to-many $k$-mer-to-tile mapping would be needed, however). For ease of exposition, we ignore tile orientations here. We completely specify the SPT with orientations, allowing tiles to simultaneously represent reverse-complement sequences, in Section S.2.

### 3.2 A General and Modular Index over Spectrum Preserving Tilings

Any SPT is immediately amenable to indexing by an entire *class* of algorithms. This is because an SPT yields a natural decomposition of the MRP query (defined in Sect. 2) where $k$-mers first map to the tiles and tile-occurrences then map to positions in references. To index a reference collection, a data structure need only compose a query for the positions where $k$-mers occur on tiles in a SPSS with a query for the positions where tiles cover the input references.

Ideally, an index should find a small SPT where $k$-mers are compactly represented in the set of tiles where tiles are "long" and tiling sequences are "short". Compact tilings exist for almost all practical applications since the amount of *unique* sequence grows much more slowly than the *total* length of reference sequences. Finding a small SPSS where $k$-mers occur only once has been solved efficiently [18–20]. However, it remains unclear if a small SPSS induces a small SPT, since an SPT must additionally encode tile-occurrence positions. Currently, tools like `pufferfish` index reference sequences using an SPT built from the *unitigs* of the compacted de Bruijn graph (cDBG) constructed over the input sequences, which has been found to be sufficiently compact for practical applications. Though the existence of SPSSs smaller than cDBGs suggest that smaller SPTs might be found for indexing, we leave the problem of finding small or even

optimal SPTs to future work. Here, we demonstrate how indexing any given SPT is *modular* and possible in general.

Given an SPT, the MRP query can be decomposed into two queries that can each be supported by sparse and efficient data structures. These queries are:

- **The kmer-to-tile query:** Given a $k$-mer $x$, `k2tile`$(x)$ returns $(i, p)$—the identity of the tile $U_i$ that contains $x$ and the offset (position) into the tile $U_i$ where $x$ occurs. That is, `k2tile`$(x) = (i, p)$ iff $U_i[p : p + k] = x$. If $x$ is not in $\mathcal{R}$, `k2tile`$(x)$ returns $\emptyset$.
- **The tile-to-occurrence query:** Given the $r$-th occurrence of the tile $U_i$, `tile2occ`$(i, r)$ returns the tuple $(n, s, w, l)$ that encodes how $U_i$ tiles the reference $R_n$. When `tile2occ`$(i, r) = (n, s, w, l)$, the $r$-th occurrence of $U_i$ occurs on $R_n$ at position $(s+w)$, with the sequence $U_i[w : w+l]$. Let the $r$-th occurrence of $U_i$ be $T_{n,m}$ on $\mathcal{T}$, then `tile2occ`$(i, r)$ returns $(n, s_{n,m}, w_{n,m}, l_{n,m})$.

When these two queries are supported, the MRP query can be computed by Algorithm 1. By adding the offset of the queried $k$-mer $x$ in a tile $U_i$ to the positions where the tile $U_i$ occurs, Algorithm 1 returns all positions where a $k$-mer occurs. Line 10 checks to ensure that any occurrence of the queried $k$-mer is returned only if the corresponding tile-occurrence of $U_i$ contains that $k$-mer. We note that storing the number of occurrences of a tile and returning `num-occs`$(U_i)$ requires negligible computational overhead. In practice, the length of tiling sequences, $\mathcal{T}$, are orders of magnitude larger than the number of unique tiles. In this work, we shall use $occ_i$, to denote the number of occurrences of $U_i$ in tiling sequences $\mathcal{T}$.

**Algorithm 1:**

```
1  def mrp(x):
2      tup ← k2tile(x)
3      if tup = ∅ then
4          return [ ]
5      (i, p) ← tup
6      occ_i ← num-occs(U_i)
7      ans ← [ ]
8      for r ← 0 to occs_i do
9          (n, s, w, l) ← tile2occ(i, r)
10         if w ≤ p ≤ (w + l − k) then
11             ans.append(n, s + p)
12     return ans
```

### 3.3 "Drop in" Implementations for Efficient $k$-mer-to-tile Queries

Naturally, prior work for indexing and compressing spectrum preserving string sets (SPSS) can be applied to implement the $k$-mer-to-tile query. When `pufferfish` was first developed, the data structures required to support the $k$-mer-to-tile query dominated the size of moderately sized indexes. Thus,

Almodaresi et al. [9] introduced a sampling scheme that samples $k$-mer positions in unitigs. Recently, Pibiri [21,22] introduced `SSHash`, an efficient $k$-mer hashing scheme that exploits minimizer based partitioning and carefully handles highly-skewed distributions of minimizer occurrences. When built over an SPSS, `SSHash` stores the $k$-mers by their order of appearance in the strings (which we term tiles) of an SPSS and thus allows easy computation of a $k$-mer's offset into a tile. Other methods based on the Burrows-Wheeler transform (BWT) [8], such as the Spectral BWT [23] and BOSS [25], could also be used. However, these methods implicitly sort $k$-mers in lexicographical order and would likely need an extra level of indirection to implement `k2tile`. Unless a compact scheme is devised, this can outweigh the space savings offered by the BWT.

### 3.4 Challenges of the Tile-to-Occurrence Query

The straightforward solution to the tile-to-occurrence query is to store the answers in a table, `utab`, where `utab`$[i]$ stores information for all occurrences of the tile $U_i$ and computing `tile2occ`$(i, r)$ amounts to a simple lookup into `utab`$[i][r]$. This is the approach taken in the `pufferfish` index and has proven to be effective for moderately sized indexes. This implementation is output optimal and is fast and cache-friendly since all $occ_i$ occurrences of a tile $U_i$ can be accessed contiguously. However, writing down all start positions of tile-occurrences in `utab` is impractical for large indexes.

For larger indexes (e.g. metagenomic references, many human genomes), explicitly storing `utab` becomes more costly than supporting the $k$-mer-to-tile query. This is because, as the number of indexed references grow, the number of distinct $k$-mers grows sub-linearly whereas the number of occurrences grows with the (cumulative) reference length. Problematically, the number of start positions of tile-occurrences grows *at least* linearly. For a reference collection with total sequence length $L$, a naive encoding for `utab` would take $O(L \lg L)$ bits, as each position require $\lceil \lg L \rceil$ bits and there can be at most $L$ distinct tiles.

Other algorithms that support "locate" queries suffer from a similar problem. To answer queries in time proportional to the number of occurrences of a query, data structures must explicitly store positions of occurrences and access them in constant time. However, storing *all* positions is impractical for large reference texts or large $k$-mer-sets. To address this, some algorithms employ a scheme to *sample* positions at some small sampling rate $s$, and perform $O(s)$ work to retrieve not-sampled positions. Since $s$ is usually chosen to be a small constant, this extra $O(s)$ work only imposes a slight overhead.

One may wonder if `utab`—which is an *inverted index*—can be compressed using the techniques developed in the Information Retrieval field [26]. For biological sequences, a large proportion of `utab` consists of very short inverted lists (e.g. unique variants in indexed genomes) that are not well-compressible. In fact, these short lists occur at a rate that is much higher than for inverted indexes designed for natural languages. So, instead applying existing compression techniques, we develop a novel *sampling* scheme for `utab` and the tile-to-occurrence query that exploits the properties of genomic sequences.

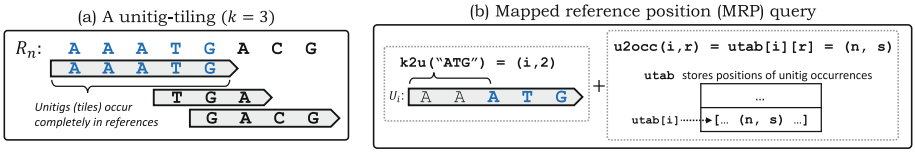


**Fig. 2.** (a) A *unitig-tiling* is an SPT where tiles, *unitigs*, always occur completely in the reference sequences. (b) The MRP query is performed by computing a $k$-mer's offset into a unitig (`k2u`), then adding the offset to the positions where *unitig-occurrences* appear in indexed reference sequences (`u2occ`). To naively support the unitig-to-occurrence query, positions of all unitig-occurrences are stored in a table, `utab`.

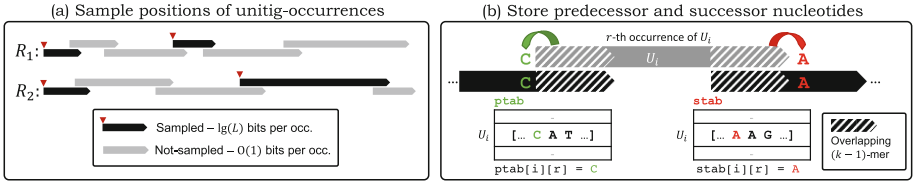


**Fig. 3.** (a) `Pufferfish2` samples unitigs and their occurrences on a unitig-tiling. Only the positions of the occurrences of the *sampled* unitigs (black) are stored in `utab`. Positions of the *not-sampled* unitigs (gray) can be computed relative to the positions of sampled unitigs by traversing backwards on the visualized tiling of references. Sampling the zero-th unitig-occurrence on every reference sequence guarantees that traversals terminate. (b) Predecessor and successor nucleotides are obtained from adjacent unitig occurrences and are stored in the order in which they appear on the references. These nucleotides for the $r$-th occurrence of $U_i$ is stored in `ptab`$[i][r]$ and `stab`$[i][r]$, respectively.

## 4 Pufferfish2

Below, we introduce `pufferfish2`, an index built over an SPT consisting of *unitigs*. `Pufferfish2` applies a sampling scheme to sparsify the tile-to-occurrence query of a given `pufferfish` index [9].

### 4.1 Interpreting `pufferfish` as an Index over a Unitig-Based SPT

Though not introduced this way by Almodaresi et al., `pufferfish` is an index over a *unitig-tiling* of an input reference collection [9]. A *unitig-tiling* is an SPT which satisfies the property that all tiles always occur completely in references where, for every tile-occurrence $T_{n,m} = U_i$, offset $w_{n,m} = 0$ and length $l_{n,m} = |U_i|$. When this property is satisfied, we term tiles *unitigs*.

An index built over unitig-tilings does not need to store tile-occurrence offsets, $\mathcal{W}$, or tile-occurrence lengths $\mathcal{L}$ since all tiles have the same offset (zero) and occur with maximal length. For indexes constructed over unitig-tilings, we shall use `k2u` to mean `k2tile`, and `u2occ` to be `tile2occ` with one change.

That is, `u2occ` omits offsets and lengths of tile occurrences since they are uninformative for unitig-tilings and returns a tuple $(n, s)$ instead of $(n, s, w, l)$, In prose, we shall refer to these queries as the $k$-mer-to-unitig and unitig-to-occurrence queries.

The MRP query over unitig-tilings can be computed with Algorithm 4 (in Section S.1) where Line 10 is removed from Algorithm 1. We illustrate the MRP query and an example of a unitig-tiling in Fig. 2.

### 4.2 Sampling Unitigs and Traversing Tilings to Sparsify the Unitig-to-Occurrence Query

`Pufferfish2` implements a sampling scheme for *unitig-occurrences* on a unitig-tiling. For some small constant $s$, our scheme samples $1/s$ rows in `utab` each corresponding to *all* occurrences of a unique unitig. In doing so, it sparsifies the `u2occ` query and `utab` by only storing positions for a subset of *sampled* unitigs. To compute unitig-to-occurrence queries, it traverses unitig-occurrences on an indexed unitig-tiling.

Notably, `pufferfish2` traverses unitig-tilings that are *implicitly* represented. For unitig-tilings with positions stored in `utab`, there exists no contiguous sequence in memory representing occurrences that is obvious to traverse. However, when viewed as an SPT, *unitig-occurrences* have *ranks* on a tiling and traversals are possible because tiling sequences map uniquely to a sequence of unitig-rank pairs.

Specifically, we define the `pred` query—an atomic traversal step that enables traversals of arbitrary lengths over reference tilings. Given the $r$-th occurrence of the unitig $U_i$, the `pred` query returns the identity and rank of the *preceding* unitig. Let tile $T_{n,m}$ be the $r$-th occurrence of the unitig $U_i$ on all tiling sequences $\mathcal{T}$. Then, $\texttt{pred}(i, r)$ returns $(j, q)$ indicating that $T_{n,m-1}$, the *preceding* unitig-occurrence, is the $q$-th occurrence of the unitig $U_j$. If there is no preceding occurrence and $m = 1$, $\texttt{pred}(i, r)$ returns the sentinel value $\emptyset$.

When an index supports `pred`, it is able to traverse "backwards" on a unitig-tiling. Successively calling `pred` yields the identities of unitigs that form a tiling sequence. Furthermore, since `pred` returns the identity $j$ *and* the rank $q$ of a preceding unitig-occurrence, accessing data associated with each visited occurrence is straightforward in a table like `utab` (i.e., with $\texttt{utab}[j][q]$).

Given the unitig-set $\mathcal{U}$, `pufferfish2` first samples a subset of unitigs $\mathcal{U}_S \subseteq \mathcal{U}$. For each sampled unitig $U_i \in \mathcal{U}_S$, it stores information for unitig-occurrences identically to `pufferfish` and records, for *all* occurrences of a sampled unitig $U_i$, a list of reference identity and position tuples in $\texttt{utab}[i]$.

To recover the position of the $r$-th occurrence a not-sampled unitig $U_i$ and to compute $\texttt{u2occ}(i, r)$, the index traverses the unitig-tiling and iteratively calls `pred` until an occurrence of a sampled unitig is found—let this be the $q$-th occurrence of $U_j$. During the traversal, `pufferfish2` accumulates number of nucleotides covered by the traversed unitig-occurrences. Since $U_j$ is a sampled unitig, the position of the $q$-th occurrence can be found in $\texttt{utab}[j][q]$. To return $\texttt{u2occ}(i, r)$, `pufferfish2` adds the number of nucleotides traversed to the start

position stored at $\texttt{utab}[j][q]$, the position of a preceding occurrence of the sampled unitig $U_j$.

This procedure is implemented in Algorithm 2 and visualized in Fig. 3. Traversals must account for $(k-1)$ overlapping nucleotides of unitig-occurrences that tile a reference (Line 5). Storing the length of the unitigs is negligible since the number of unique unitigs is much smaller than the number of occurrences.

**On the Termination of Traversals.** Any unitig that occurs as the zero-th occurrence (i.e., with rank zero) of a tiling-sequence is always sampled. This way, backwards traversals terminate because every occurrence of a not-sampled unitig occurs after a sampled unitig. This can be seen from Fig. 3. Concretely, if $T_{n,1} = U_i$ for some tiling-sequence $T_n$, then the unitig $U_i$ must always be sampled.

**Algorithm 2:**

```
1 def u2occ(i, r):
2     l ← 0
3     while !isSamp[i] do
4         (i, r) = pred(i, r)
5         l ← l + |U_i| − k + 1
6     (n, s) ← utab[i][r]
7     return (n, s + l)
```

**Algorithm 3:**

```
1 def pred(i, r):
2     p ← ptab[i][r]
3     y ← p ∘ U_i[: k − 1]
4     (j, _) ← k2u(y)
5     s ← U_i[k]
6     t ← rank_p(ptab[i], r)
7     q ← select_s(stab[j], t)
8     return (j, q)
```

### 4.3 Implementing the `pred` Query with `pufferfish2`

`Pufferfish2` computes the `pred` query in constant time while requiring only constant space per unitig-occurrence by carefully storing *predecessor* and *successor* nucleotides of unitig-occurrences.

**Predecessor and Successor Nucleotides.** Given the tiling sequence $T_n = [T_{n,1}, \ldots, T_{n,M_n}]$, we say that a unitig-occurrence $T_{n,m}$ is *preceded* by $T_{n,m-1}$, and that $T_{n,m-1}$ is *succeeded* by $T_{n,m}$. Suppose $T_{n,m} = U_i$, and $T_{n,m-1} = U_j$, and let the unitigs have lengths $\ell_i$ and $\ell_j$, respectively.

We say that, $T_{n,m-1}$ precedes $T_{n,m}$ with predecessor nucleotide $p$. The predecessor nucleotide is the nucleotide that precedes the unitig-occurrence $T_{n,m}$ on the reference sequence $R_n$. Concretely, $p$ is the first nucleotide on the last $k$-mer of the preceding unitig, i.e., $p = T_{n,m-1}[\ell_j - k]$. We say that, $T_{n,m}$ succeeds $T_{n,m-1}$ with successor nucleotide $s$. Accordingly, the successor nucleotide, $s$, is the last nucleotide on the first $k$-mer of the succeeding unitig, i.e., $s = T_{n,m}[k]$.

Abstractly, the preceding occurrence $T_{n,m-1}$ can be "reached" from the succeeding occurrence $T_{n,m}$ by prepending its predecessor nucleotide to the $(k-1)$-length prefix of $T_{n,m}$. Given $T_{n,m}$ and its predecessor nucleotide $p$, the $k$-mer $y$ that is the last $k$-mer on the preceding occurrence $T_{n,m-1}$ can be obtained with $y = p \circ T_{n,m}[: k-1]$. Given an occurrence $T_{n,m}$, let the functions `pred-nuc` $(T_{n,m})$ and `succ-nuc` $(T_{n,m})$ yield the predecessor nucleotide and

the successor nucleotide of $T_{n,m}$, respectively. If $T_{n,m}$ is the first or last unitig-occurrence pair on $T_n$, then $\texttt{succ-nuc}(T_{n,m})$ and $\texttt{pred-nuc}(T_{n,m})$ return the "null" character, '$'.

These notationally dense definitions can be more easily understood with a figure. Figure 3 shows how predecessor and successor nucleotides of a given unitig-occurrence on a tiling are obtained.

**Concrete Representation.** `Pufferfish2` first samples a set of unitigs $\mathcal{U}_S \subseteq \mathcal{U}$ from $\mathcal{U}$ and stores a bit vector, `isSamp`, to record if a unitig $U_i$ is sampled where $\texttt{isSamp}[i] = 1$ iff $U_i \in \mathcal{U}_S$. `Pufferfish2` stores in `utab` the reference identity and position pairs for occurrences of *sampled* unitigs only.

After sampling unique unitigs, `pufferfish2` stores a *predecessor nucleotide table*, `ptab`, and a *successor nucleotide table*, `stab`. For each not-sampled unitig $U_i$ *only*, $\texttt{ptab}[i]$ stores a list of predecessor nucleotides for each occurrence of $U_i$ in the unitig-tiling. For *all* unitigs $U_i$, $\texttt{stab}[i]$ stores a list of successor nucleotides for each occurrence of $U_i$. Concretely, when the unitig-occurrence $T_{n,m}$ is the $r$-th occurrence of $U_i$,

$$\texttt{ptab}[i][r] = \texttt{pred-nuc}(T_{n,m}) \quad \text{and} \quad \texttt{stab}[i][r] = \texttt{succ-nuc}(T_{n,m}).$$

As discussed in Sect. 4.2, unitigs that occur as the zero-th element on a tiling is always sampled so that every occurrence of a not-sampled unitig has a predecessor. If $T_{n,m}$ has no successor and is the last unitig-occurrence on a tiling sequence, $\texttt{stab}[i][j]$ contains the sentinel symbol '$'. Figure 3 illustrates how predecessor and successor nucleotides are stored.

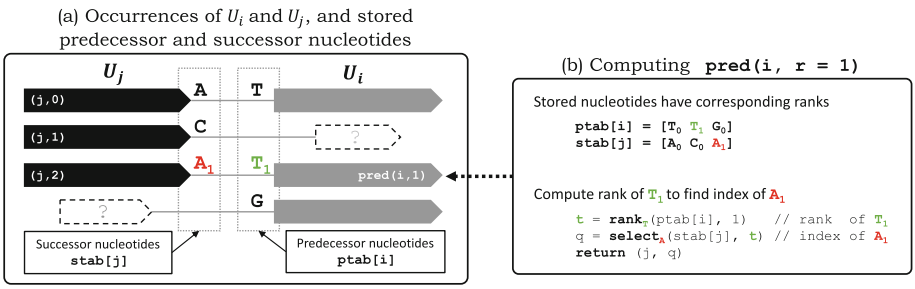


**Fig. 4.** Visualizing the `pred` query that finds the occurrence of $U_j$ that precedes the queried occurrence of $U_i$ with rank 1. (a) All occurrences of $U_i$ and $U_j$ are visualized (in sorted order) with their preceding and succeeding unitig occurrences, respectively. The figure shows stored successor nucleotides for $U_j$, and predecessor nucleotides for $U_i$. Whenever an occurrence of $U_j$ precedes an occurrence of $U_i$, a corresponding pair of nucleotides "`A`" and "`T`" occur and are stored in $\texttt{stab}[j]$ and $\texttt{ptab}[i]$ respectively. (b) Their *ranks* (annotated with subscripts) of the corresponding predecessor-successor nucleotide pair *match* in $\texttt{ptab}[i]$ and $\texttt{stab}[j]$, but the *indices* do not. A rank query for predecessor nucleotide "`T`" at index $r = 1$ yields the matching rank of the successor nucleotide "`A`". A select query for the nucleotide "`A`" with rank 1 yields the *index* and occurrence of the predecessor $U_j$.

**Computing the `pred` Query.** Given the $k$-mer-to-unitig query, `pufferfish2` supports the `pred` query for any unitig $U_i$ that is not-sampled. When the $r$-th occurrence of $U_i$ succeeds the $q$-th occurrence of $U_j$, it computes $\texttt{pred}(i, r) = (j, q)$ with Algorithm 3. To compute `pred`, it constructs a $k$-mer to find $U_j$, and then computes one rank and one select query over the stored lists of nucleotides to find the correct occurrence.

`Pufferfish2` first computes $j$, the identity of the preceding unitig. The last $k$-mer on the preceding unitig must be the first $(k-1)$-mer of $U_i$ *prepended* with predecessor nucleotide of the $r$-th occurrence of $U_i$. Given $\texttt{ptab}[i][r] = p$, it constructs the $k$-mer, $y = p \circ U_i[:k-1]$, that must be the last $k$-mer on $U_j$. So on Line 4, it computes $\texttt{k2u}(y)$ to obtain the identity of the preceding unitig $U_j$.

It then computes the unitig-rank, $q$, of the preceding unitig-occurrence of $U_j$. Each time $U_i$ is preceded by the nucleotide $p$, it must be preceded by the *same* unitig $U_j$ since any $k$-mer occurs in only one unitig. Accordingly, each occurrence $U_j$ that is succeeded by $U_i$ must always be succeeded by the *same* nucleotide $s$ equal to the $k$-th nucleotide of $U_i$, $U_i[k]$. For the preceding occurrence of $U_j$ that the algorithm seeks to find, the nucleotide $s$ is stored at some unknown index $q$ in $\texttt{stab}[j]$—the list of successor nucleotides of $U_j$.

Whenever an occurrence of $U_i$ succeeds an occurrence of $U_j$, so do the corresponding pair predecessor and successor nucleotides stored in $\texttt{ptab}[i]$ and $\texttt{stab}[j]$. Since $\texttt{ptab}[i]$ and $\texttt{stab}[j]$ store predecessor and successor nucleotides in the order in which unitig-occurrences appear in the tiling sequences, the following *ranks* of stored *nucleotides* must be equal: (1) the rank of the nucleotide $p = \texttt{ptab}[i][r]$ at index $r$ in the list of predecessor nucleotides, $\texttt{ptab}[i]$, of the succeeding unitig $U_i$, and (2) the rank of the nucleotide $s = U_i[k]$ at index $q$ in the list of successor nucleotides, $\texttt{stab}[j]$, of the preceding unitig $U_j$. We illustrate this correspondence between ranks in Fig. 4. So to find $q$, the rank of the preceding unitig-occurrence, `pufferfish2` computes the rank of the predecessor nucleotide, $t = \texttt{rank}_p(\texttt{ptab}[i], r)$. Then, computing $\texttt{select}_s(\texttt{stab}[i], t)$, the index where the $t$-th rank successor nucleotide of $U_j$ occurs must yield $q$.

**Time and Space Analysis.** `Pufferfish2` computes the `pred` query in constant time. The $k$-mer for the query `k2u` is assembled in constant time, and the `k2u` query itself is answered in constant time, as already done in the `pufferfish` index [9].

For not-sampled unitigs, `pufferfish2` does not store positions of unitig-occurrences in `utab`. Instead, it stores nucleotides in tables `stab` and `ptab`. These tables are implemented by *wavelet matrices* that support rank, select, and access operations in $O(\lg \sigma)$ time on sequences with alphabet size $\sigma$ while requiring only $\lg \sigma + o(\lg \sigma)$ bits per element [24].

As explained in Sect. 3.1, we have avoided the treatment of *orientations* of nucleotide sequences for brevity. In actuality, unitigs may occur in a *forward* or a *backwards* orientation (i.e., with a reverse complement sequence). When considering orientations, `pufferfish2` implements the `pred` query by storing and

querying over lists of *nucleotide-orientation* pairs. In this case, `ptab` and `stab` instead store predecessor-orientation and successor-orientation pairs. Accordingly, wavelet matrices are then built over alphabets of size 8 and 9 respectively—deriving from eight nucleotide-orientation pairs and one sentinel value for unitig-occurrences that have no predecessor. Thus, `ptab` and `stab` in total require $\approx 7$ bits per unitig-occurrence (since $7 = \lceil \lg 8 \rceil + \lceil \lg 9 \rceil$). We describe how the `pred` query is implemented with orientations in Section S.3.

**Construction.** The current implementation of `pufferfish2` sparsifies the unitig-to-occurrence query and compresses the table of unitig occurrences, `utab`, of an existing `pufferfish` index, and inherits its $k$-mer-to-unitig mapping. In practice, sampling and building a `pufferfish2` index always takes less time than the initial `pufferfish` index construction. In brief, building `pufferfish2` amounts to a linear scan over an SPT. We describe how `pufferfish2` in constructed in more detail in Section S.4.

### 4.4 A Random Sampling Scheme to Guarantee Short Backwards Traversals

Even with a constant-time `pred` query, computing the unitig-to-occurrence query is fast only if the length of backwards traversals—the number of times `pred` is called — is small. So for some small constant $s$, a sampling scheme should sample $1/s$ of *unique* unitigs, store positions of only $1/s$ of unitig-*occurrences* in `utab`, and result in traversal lengths usually of length $s$.

At first, one may think that a greedy sampling scheme that traverses tiling sequences to sample unitigs could be used to bound traversal lengths to some given maximum length, $s$. However, when tiling sequences become much longer than the number of unique unitigs, such a greedy scheme samples almost *all* unitigs and only somewhat effective in limited scenarios (see Section S.5). Thus, we introduce the *random* sampling scheme that samples $1/s$ of unitigs uniformly at random from $\mathcal{U}$. This scheme guarantees that traversals using the `pred` query terminate in $s$ steps *in expectation* if each unitig-occurrence $T_{n,m}$ is independent and identically distributed and drawn from an arbitrary distribution. Then, backwards traversals until the occurrence of a sampled unitig is a series of Bernoulli trials with probability $1/s$, and traversal lengths follow a geometric distribution with mean $s$. Although this property relies on a simplifying assumption, the random sampling scheme works well in practice.

**Table 1.** Size and speed of `pufferfish2` indexes querying 10 million random $k$-mers and 100,000 reads. Uncompressed, baseline implementations of the unitig-to-occurrence query (`pufferfish` indexes with the *sparse* `k2u` implementation [9]) are labeled with "None" sampling strategy. Relative sizes of compressed representations and relative slowdowns to the baseline are indicated in parentheses.

| Dataset | Sampling strategy | `u2occ` size (GB) | 10M $k$-mers (secs) | 100K reads (secs) |
|---|---|---|---|---|
| 7 Humans | None | 16.8 | 86.1 | 139.4 |
| | Random ($s = 3, t = .05$) | 7.8 (0.46) | 4159.1 (43.8×) | 8092.8 (58.04×) |
| | Random ($s = 3, t = .25$) | 9.9 (0.59) | 681.1 (7.9×) | 1466.2 (10.52×) |
| 4000 Bacteria | None | 7.7 | 35.5 | 12.6 |
| | Random ($s = 3, t = .05$) | 3.7 (0.48) | 420.4 (11.9×) | 15.6 (1.24×) |
| | Random ($s = 3, t = .25$) | 4.7 (0.61) | 323.8 (9.1×) | 15.5 (1.23×) |
| 30K Human gut | None | 86.3 | 80.6 | 178.7 |
| | Random ($s = 3, t = .05$) | 45.6 (0.53) | 439.4 (5.5×) | 570.2 (3.19×) |
| | Random ($s = 3, t = .25$) | 54.4 (0.63) | 365.2 (4.5×) | 576.9 (3.23×) |
| | Random ($s = 6, t = .05$) | 34.6 (0.40) | 1037.5 (12.9×) | 644.8 (3.61×) |
| | Random ($s = 6, t = .25$) | 45.6 (0.53) | 614.0 (7.6×) | 646.1 (3.56×) |

### 4.5 Closing the Gap Between a Constant Time `pred` Query and Contiguous Array Access

Even though the `pred` query is constant time and traversals are short, it is difficult to implement `pred` queries in with speed comparable to *contiguous array accesses* that are used to compute the `u2occ` for when `utab` is "dense"—i.e., uncompressed and not sampled. In fact, any compression scheme for `utab` would have difficulty contending with constant time contiguous array access regardless of their asymptotics since dense implementations are output optimal, very cache friendly, and simply store the answers to queries in an array. To close the gap between theory and practice, `pufferfish2` exploits several optimizations.

In practice, a small proportion of unique unitigs are "popular" and occur extremely frequently. Fortunately, the total number of occurrences of popular unitigs is small relative to other unitigs. To avoid an excessively large number of traversals from a not-sampled unitig, `pufferfish2` modifies the sampling scheme to always sample popular unitigs that occur more than a preset number, $\alpha$, times. Better yet, we re-parameterize this optimization and set $\alpha$ so that the total number of occurrences of popular unitigs sum to a given proportion $0 < t \leq 1$ of the total occurrences of all the unitigs. For example, setting $t = 0.25$ restricts `pufferfish2` to sample from 75% of the total size of `utab` consisting of unitigs that occur most infrequently.

Also, the MRP and `pred` query are especially amenable to caching. Notably, `pufferfish2` caches and memoizes redundant `k2u` queries in successive `pred` queries. Also, it caches "streaming" queries to exploit the fact that successive queried $k$-mers (e.g., from the same sequenced read) likely land on the same unitig. We describe in more detail these and other important optimizations in Section S.6.

## 5 Experiments

We assessed the space-usage of the indexes constructed by `pufferfish2` from several different whole-genome sequence collections, as well as its query performance with different sampling schemes. Reported experiments were performed on a server with an Intel Xeon CPU (E5-2699 v4) with 44 cores and clocked at 2.20 GHz, 512 GB of memory, and a 3.6 TB Toshiba MG03ACA4 HDD.

**Datasets.** We evaluated the performances on a number of datasets with varying attributes: (1) Bacterial collection: a random set of 4000 bacterial genomes from the NCBI microbial database; (2) Human collection: 7 assembled human genome sequences from [27]; and (3) Metagenomic collection: 30,691 representative sequences from the most prevalent human gut prokaryotic genomes from [28].

**Results.** To emulate a difficult query workload, we queried the indexes with 10 million random *true positive* $k$-mers sampled uniformly from the indexed references. Our results from Table 1 show that sampling *popular* unitigs is critical to achieve reasonable trade-offs between space and speed. When indexing seven human genomes, the difference in space between always sampling using $t = 0.05$ and $t = 0.25$, is only 2.1 GB (12.5% of the uncompressed `utab`). However, explicitly recording 2.1 GB of positions of occurrences of popular unitigs, *substantially* reduces the comparative slowdown from 43.8× to 7.9×. This is because setting $t = 0.25$ instead of $t = 0.05$ greatly reduces the maximum number of occurrences of a *not-sampled* unitig—from ≈87,000 to ≈9,000 times, respectively. Here, setting $t = 0.25$ means that random $k$-mer queries that land in not-sampled unitigs perform many fewer traversals over reference tilings.

On metagenomic datasets, indexes are compressed to a similar degree but differences in query speed at different parameter settings are small. `Pufferfish2` is especially effective for a *large* collection of bacterial genomes. With the fastest parameter setting, it incurs only a 4.5× slowdown for random queries while reducing the size of `utab` for the collection of 30,000 bacterial genomes by 37% (from 86.3 GB to 54.4 GB).

Apart from random lookup queries, we also queried the indexes with $k$-mers deriving from sequenced readsets [29,30]. We measured the time to query and recover the positions of all $k$-mers on 100,000 reads. This experiment demonstrates how the slowdown incurred from sampling can (in most cases) be further reduced when queries are positionally coherent or miss. Successive $k$-mer queries from the same read often land on the same unitig and can thus be cached (see Sect. 4.5). *True negative* $k$-mers that do not occur in the indexed reference collection neither require traversals nor incur any slowdowns.

To simulate a metagenomic analysis, we queried reads from a human stool sample against 4,000 bacterial genomes. This is an example of a low hit-rate analysis where 18% of queried $k$-mers map to indexed references. In this scenario, `pufferfish2` reduces the size of `utab` by *half* but incurs only a 1.2× slowdown. We also queried reads from the same human stool sample against the collection of 30,000 bacterial genomes representative of the human gut. Here, 88% of $k$-mers are found in the indexed references. At the sparsest setting, `pufferfish2` indexes incur only a 3.6× slowdown while reducing the size of `utab` by 60%.

We observe that `pufferfish2`'s sampling scheme is less effective when indexing a collection of seven human genomes. When sampled with $s = 3$ and $t = 0.25$, `pufferfish2` incurs a 10.5× slowdown when querying reads from a DNA-seq experiment in which 92% of queried $k$-mers occur in reference sequences. Interestingly, the slowdown when querying reads is larger than the slowdown when querying random $k$-mers. This is likely due to biases from sequencing that cause $k$-mers and reads to map to non-uniformly indexed references. Nonetheless, this result motivates future work that could design sampling schemes optimized for specific distributions of query patterns.

We expect to see less-pronounced slowdowns in practice than those reported in Table 1. This is because tools downstream of an index like `pufferfish2` almost always perform operations *much* slower after straightforward exact lookups for $k$-mers. For example, aligners have to perform alignment accounting for mismatches and edits. Also, our experiments pre-process random $k$-mer sets and read-sets so that no benchmark is I/O bound. Critically, the compromises in speed that `pufferfish2` makes are especially palatable because it trades-off speed in the *fastest* operations in analyses—*exact* $k$-mer queries—while substantially reducing the space required for the *most space intensive* operation.

**Table 2.** Sizes in GB of possible, new indexes—with `k2u` implemented by `SSHash` and `u2occ` by `pufferfish2`—compared to the size of original `pufferfish` indexes. Selected sampling parameters for datasets (top-to-bottom) are $(s = 3, t = 0.25)$, $(s = 3, t = 0.05)$, and $(s = 6, t = 0.05)$, respectively.

| Dataset | `u2occ` w/`pufferfish2` | `k2u` w/`SSHash` | New index | Original `pufferfish` index |
|---|---|---|---|---|
| 7 Human | 9.9 | 3.2 | **13.1** | 28.0 |
| 4000 Bacteria | 3.7 | 7.3 | **11.0** | 26.1 |
| 30K Human gut | 34.6 | 22.0 | **55.6** | 131.7 |

**Using `SSHash` for Even Smaller Indexes.** For convenience, we have implemented our SPT compression scheme within an index that uses the *specific* sparse `pufferfish` implementation for the $k$-mer-to-tile ($k$-mer-to-unitig) mapping [9]. However, the SPT enables the construction of modular indexes that use *various* data structures for the $k$-mer-to-tile mapping and the tile-to-reference mapping, provided only a minimalistic API between them. A recent representation of the $k$-mer-to-tile mapping that supports all the necessary functionality is `SSHash` [22]. Compared to the `k2u` component of `pufferfish`, `SSHash` is almost always substantially smaller. Further, it usually provides faster query speed compared to the *sparse* `pufferfish` implementation of the $k$-mer-to-tile query, especially when streaming queries are being performed.

In Table 2, we calculate the size of indexes if `SSHash` is used for the $k$-mer-to-tile mapping—rather than the *sparse* `pufferfish` implementation. These sizes then represent overall index sizes that would be obtained by pairing a state-of-the-art representation of the $k$-mer-to-tile mapping with a state-of-the-art

representation of the tile-to-reference mapping (that we have presented in this work). Practically, the only impediment to constructing a fully-functional index from these components is that they are implemented in different languages (`C++` for `SSHash` and `Rust` for `pufferfish2`)—we are currently addressing this issue.

Importantly, these results demonstrate that, when `SSHash` is used, the representation of the tile-to-occurrence query becomes a bottleneck in terms of space, occupying an increasingly larger fraction of the overall index. Table 2 shows that, in theory, if one fully exploits the modularity of SPTs, new indexes that combine `SSHash` with `pufferfish2` would be *half* the space of the original `pufferfish` index. As of writing, with respect to an index over 30,000 bacterial genomes, the estimated difference in *monetary* cost of an AWS EC2 instance that can fit a new 55.6 GB index versus a 131 GB `pufferfish` index in memory is 300USD per month (see Section S.7).

**Comparing to MONI and the r-Index.** We compared `pufferfish2` to MONI, a tool that builds an r-index to locate maximal exact matches in highly repetitive reference collections [6]. In brief, `pufferfish2` is faster and requires less space than MONI for our benchmarked bacterial dataset. Our tool does so with some trade-offs. `Pufferfish2` supports rapid locate queries for $k$-mers of a *fixed* length, while r-index based approaches supports locate queries for patterns of any *arbitrary* length and can be used to find MEMs. Notably, it has been shown that both $k$-mer and MEM queries can be used for highly effective read-mapping and alignment [1,6].

For reference, we built MONI on our collection of 4,000 bacterial genomes. Here, MONI required 51.0G of disk space to store which is 29% larger than the `pufferfish` index (39.5 GB) with its *dense* `k2u` implementation—its *least* space-efficient configuration. The most space efficient configuration of the `pufferfish2` index (with $s = 3$, $t = .25$) is 42% the size of MONI when built on from the same data and requires 21.7 GB of space. Compared to a theoretically possible index specified in Table 2 that would only require 11.0 GB, MONI would need 4.6× more space.

We also performed a best-effort comparison of query speed between `pufferfish2` and MONI. Unfortunately, it is not possible to directly measure the speed of exact locate queries for MONI because it does not expose an interface for such queries. Instead, we queried MONI to find MEMs on true-positive $k$-mers treating each $k$-mer as unique read (encoded in FASTQ format as MONI requires). We argue that this is a reasonable proxy to exact locate queries because, for each true-positive $k$-mer deriving from an indexed reference sequence, the entire $k$-mer itself is the maximal exact match. For MONI, just like in benchmarks for in Table 1, we report the time taken for computing queries only and ignore time required for I/O operations (i.e. loading the index and quries, and writing results to disk).

We found that `pufferfish2` is faster than MONI when querying $k$-mers against our collection of 4,000 bacterial genomes. MONI required 1,481.7 s to query the same set of 10 million random true-positive k-mers queried in Table 1. When compared to the slowest built most space efficient configuration of `pufferfish2` benchmarked in Table 1, `pufferfish2` is 3.5× faster.

## 6 Discussion and Future Work

In this work, we introduce the *spectrum preserving tiling* (SPT), which describes how a spectrum preserving string set (SPSS) tiles and "spells" an input collection of reference sequences. While considerable research effort has been dedicated to constructing space and time-efficient indexes for SPSS, little work has been done to develop efficient representations of the tilings themselves, despite the fact that these tilings tend to grow more quickly than the SPSS and quickly become the size bottleneck when these components are combined into reference indexes. We describe and implement a sparsification scheme in which the space required for representing an SPT can be greatly reduced in exchange for an expected constant-factor increase in the query time. We also describe several important heuristics that are used to substantially lessen this constant-factor in practice. Having demonstrated that modular reference indexes can be constructed by composing a $k$-mer-to-tile mapping with a tile-to-occurrence mapping, we have thus opened the door to exploring an increasingly diverse collection of related reference indexing data structures.

Despite the encouraging progress that has been made here, we believe that there is much left to be explored regarding the representation of SPTs, and that many interesting questions remain open. Some of these questions are: (1) How would an algorithm sample individual unitig-occurrences instead of all occurrences of a unitig to *explicitly* bound the lengths of backwards traversals? (2) Does a smaller SPSS imply a small SPT and could one compute an optimally small SPT? (3) Given some distributional assumptions for queries, can an algorithm sample SPTs to minimize the expected query time? (4) In practice, how can an implemented tool combine our sampling scheme with existing compression algorithms for the highly skewed tile-to-occurrence query? (5) Can a *lossy* index over an SPT be constructed and applied effectively in practical use cases?

With excitement, we discuss in more detail these possibilities for future work in more detail in Section S.8.

**Funding.** This work is supported by the NIH under grant award numbers R01HG009937 to R.P.; the NSF awards CCF-1750472 to R.P. and CNS-1763680 to R.P; and NSF award No. to DGE-1840340 J.F. This work was also partially supported by the project MobiDataLab (EU H2020 RIA, grant agreement Nº101006879).

**Conflicts of Interest.** R.P. is a co-founder of Ocean Genomics Inc.

## References

1. Almodaresi, F., Zakeri, M., Patro, R.: PuffAligner: a fast, efficient and accurate aligner based on the pufferfish index. Bioinformatics **37**(22), 404–4055 (2021)
2. Patro, R., Duggal, G., Love, M.I., Irizarry, R.A., Kingsford, C.: Salmon provides fast and bias-aware quantification of transcript expression. Nat. Methods **14**(4), 417–419 (2017)
3. Bray, N.L., Pimentel, H., Melsted, P., Pachter, L.: Near-optimal probabilistic RNA-seq quantification. Nat. Biotechnol. **34**(5), 525–527 (2016)

4. Patro, R., Mount, S.M., Kingsford, C.: Sailfish enables alignment-free isoform quantification from RNA-seq reads using lightweight algorithms. Nat. Biotechnol. **32**(5), 462–464 (2014)
5. Gagie, T., Navarro, G., Prezza, N.: Optimal-time text indexing in BWT-runs bounded space. In: Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, USA, pp. 1459–1477. Society for Industrial and Applied Mathematics (2018)
6. Rossi, M., Oliva, M., Langmead, B., Gagie, T., Boucher, C.: MONI: a pangenomic index for finding maximal exact matches. J. Comput. Biol. **29**(2), 169–187 (2022). PMID: 35041495
7. Ahmed, O., Rossi, M., Gagie, T., Boucher, C., Langmead, B.: SPUMONI 2: improved pangenome classification using a compressed index of minimizer digests. BioRxiv (2022)
8. Burrows, M., Wheeler, D.: A block-sorting lossless data compression algorithm. Digital SRC Research Report, Citeseer (1994)
9. Almodaresi, F., Sarkar, H., Srivastava, A., Patro, R.: A space and time-efficient index for the compacted colored de Bruijn graph. Bioinformatics **34**(13), i169–i177 (2018)
10. Kim, D., Paggi, J.M., Park, C., Bennett, C., Salzberg, S.L.: Graph-based genome alignment and genotyping with HISAT2 and HISAT-genotype. Nat. Biotechnol. **37**(8), 907–915 (2019)
11. Garrison, E., et al.: Variation graph toolkit improves read mapping by representing genetic variation in the reference. Nat. Biotechnol. **36**(9), 875–879 (2018)
12. Minkin, I., Pham, S., Medvedev, P.: TwoPaCo: an efficient algorithm to build the compacted de Bruijn graph from many complete genomes. Bioinformatics **33**(24), 4024–4032 (2016)
13. Chikhi, R., Limasset, A., Medvedev, P.: Compacting de Bruijn graphs from sequencing data quickly and in low memory. Bioinformatics **32**(12), i201–i208 (2016)
14. Khan, J., Patro, R.: Cuttlefish: fast, parallel and low-memory compaction of de Bruijn graphs from large-scale genome collections. Bioinformatics **37**(Supplement_1), i177–i186 (2021)
15. Khan, J., Kokot, M., Deorowicz, S., Patro, R.: Scalable, ultra-fast, and low-memory construction of compacted de Bruijn graphs with Cuttlefish 2. Genome Biol. **23**(1), 190 (2022). https://doi.org/10.1186/s13059-022-02743-6
16. Ekim, B., Berger, B., Chikhi, R.: Minimizer-space de Bruijn graphs: whole-genome assembly of long reads in minutes on a personal computer. Cell Syst. **12**(10), 958-968.e6 (2021)
17. Karasikov, M., Mustafa, H., Rätsch, G., Kahles, A.: Lossless indexing with counting de Bruijn graphs. Genome Res. **32**(9), 1754–1764 (2022)
18. Rahman, A., Medvedev, P.: Representation of $k$-mer sets using spectrum-preserving string sets. In: Schwartz, R. (ed.) RECOMB 2020. LNCS, vol. 12074, pp. 152–168. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45257-5_10
19. Schmidt, S., Alanko, J.N.: Eulertigs: minimum plain text representation of k-mer sets without repetitions in linear time. BioRxiv (2022)
20. Břinda, K., Baym, M., Kucherov, G.: Simplitigs as an efficient and scalable representation of de Bruijn graphs. Genome Biol. **22**(1), 96 (2021). https://doi.org/10.1186/s13059-021-02297-z
21. Pibiri, G.E.: On weighted k-mer dictionaries. In: International Workshop on Algorithms in Bioinformatics (WABI), pp. 9:1–9:20 (2022)

22. Pibiri, G.E.: Sparse and skew hashing of k-mers. Bioinformatics **38**(Supplement_1), i185–i194 (2022)
23. Alanko, J.N., Puglisi, S.J., Vuohtoniemi, J.: Succinct k-mer sets using subset rank queries on the spectral burrows-wheeler transform. BioRxiv (2022)
24. Claude, F., Navarro, G.: The wavelet matrix. In: Calderón-Benavides, L., González-Caro, C., Chávez, E., Ziviani, N. (eds.) SPIRE 2012. LNCS, vol. 7608, pp. 167–179. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34109-0_18
25. Bowe, A., Onodera, T., Sadakane, K., Shibuya, T.: Succinct de Bruijn graphs. In: Raphael, B., Tang, J. (eds.) WABI 2012. LNCS, vol. 7534, pp. 225–235. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33122-0_18
26. Pibiri, G.E., Venturini, R.: Techniques for inverted index compression. ACM Comput. Surv. **53**(6), 125:1–125:36 (2021)
27. Baier, U., Beller, T., Ohlebusch, E.: Graphical pan-genome analysis with compressed suffix trees and the Burrows-Wheeler transform. Bioinformatics **32**(4), 497–504 (2015)
28. Hiseni, P., Rudi, K., Wilson, R.C., Hegge, F.T., Snipen, L.: HumGut: a comprehensive human gut prokaryotic genomes collection filtered by metagenome data. Microbiome **9**(1), 165 (2021)
29. Zook, J.M., et al.: Extensive sequencing of seven human genomes to characterize benchmark reference materials. Sci. Data **3**(1), 160025 (2016)
30. Mas-Lloret, J., et al.: Gut microbiome diversity detected by high-coverage 16S and shotgun sequencing of paired stool and colon sample. Sci. Data **7**(1), 92 (2020)
31. Moffat, A., Stuiver, L.: Binary interpolative coding for effective index compression. Inf. Retrieval **3**(1), 25–47 (2000). https://doi.org/10.1023/A:1013002601898
32. Li, H.: Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics **34**(18), 3094–3100 (2018)