Batched k-mer lookup on the Spectral Burrows-Wheeler Transform

Jarno N. Alanko* Elena Biagi* Joel Mackenzie† Simon J. Puglisi*

Abstract

Since their emergence some two decades ago, indexes based on the Burrows-Wheeler transform (BWT) have been intensely studied and today find wide use in genomics, where they form the basis of software tools for read alignment and k-mer lookup—routine tasks in modern data-intensive bioinformatics pipelines.

BWT-based indexes reduce an existential query for a pattern P of length m to a sequence of up to m pairs of rank queries on a sequence derived from the underlying indexed data. In general these rank queries exhibit poor locality of memory reference, with each pair causing one or two cache misses, something that has become generally accepted as a limitation of these indexes.

However, in the above mentioned applications a typical experimental run will search for 100s of millions—even billions—of patterns using the index. In this paper we show that, taken across such a large set of patterns, rank queries do exhibit locality of memory reference and that this can be exploited by reorganising the order in which rank queries are issued. We show this leads to significant performance gains—in particular, k-mer lookup queries can be answered several times faster when a batch of patterns is treated holistically.

1 Introduction

The Burrows-Wheeler transform (BWT) [9] of a string T of $n = |T|$ symbols drawn from an alphabet σ is a reversible permutation of the symbols of T, defined by the lexicographical order of T's cyclic rotations. The BWT was introduced as a tool for data compression, but its deep relationship to pattern matching was observed by Ferragina and Manzini [13] who described an index taking space proportional to the empirical entropy of T while being able to count the number of occurrences of a pattern P in T in $O(|P| \log \sigma)$ time. In the 25 years since, BWT-based text indexes have gained widespread use in the field of genomics, for tasks such as read alignment [24, 23, 25, 16] and genome assembly [33, 12].

The spectral Burrows-Wheeler transform (SBWT) [4] is a recent variant of the BWT specialized for k-mer lookup queries on the k-spectrum of T—the set of all distinct k-length substrings occurring in T. A k-mer lookup query asks for the colexicographic (colex) rank of a query pattern $P[1..k]$ amongst all distinct k-length strings occurring in T (or $\perp$ if P does not exist in T's spectrum). These queries have a multitude of applications in high-throughput genomics, for example in algorithms for pseudoalignment [5].

A k-mer lookup query can be reduced to a sequence of up to k pairs of rank queries on the SBWT sequence, denoted X, which is a sequence of symbols derived from and encoding T's k-spectrum (we give a formal definition below). A rank query $\text{rank}_c(i)$ returns the number of occurrences of symbol c in $X[1..i]$. For a query k-mer P, each pair of rank queries issued delineate the colex interval of the k-spectrum that contains all k-mers having $P[1..i]$ as a suffix.[1] Thus, each step in the lookup process may take the current interval to a very distant region of colex space from the previous interval. The result is cache misses—approximately one per rank query, something that has become a generally accepted price with the use of (S)BWT-based indexes.

In the pseudoalignment application mentioned above, a single experimental run may issue billions of k-mer lookups. The prevailing view in both theoretical work and in software tools implementing BWT-based indexes is that queries are independent and are processed one at a time. For reasons that will become clear, we call this *horizontal batch processing*. Given a batch of patterns $P_1 \dots P_b$, we search for P_1 first (issuing up to $2|P_1|$ rank queries), before moving to P_2 (possibly another $2|P_2|$ rank queries), and so on. With horizontal batch processing, no relatedness between the rank queries *across patterns in the batch* is exploited. There are, however, many ways in which performance can be improved by treating a batch of patterns holistically. To take a simple example, patterns sharing long common prefixes (and, at an extreme, identical patterns) will issue many identical rank queries.

Contributions With the preceding discussion in mind, this article explores a range of techniques for exploiting batched query processing to improve the per-query

*University of Helsinki, Helsinki, Finland.
†The University of Queensland, Brisbane, Australia.

[1] Pattern matching search in traditional BWT-based indexes works essentially the same way.

performance of BWT-based indexes. For the sake of keeping scope reasonable, we focus throughout on the SBWT and k-mer lookup queries. Our main contributions are summarized as follows.

1. We describe an orthogonal approach to horizontal processing—*vertical batch processing*—that, via a careful reorganization of computation, drastically reduces the number of cache-misses compared to processing queries independently. At a high level, the idea is to view a batch of b patterns each of length k as a $b \times k$ matrix. In contrast to horizontal processing, in which the matrix is processed one row (i.e. pattern) at a time, scanning the characters in each row left-to-right, vertical search processes the matrix columnwise. The upshot is that for a given position within the k-mers (i.e. column) all rank queries for a given letter are now issued in a non-decreasing sequence of positions. This translates, essentially, to making k cache-friendly passes over the SBWT and is 7× faster than basic horizontal processing in our experiments.

2. Streaming search is a form of horizontal batch processing that aims to exploit the structure of a certain type of query batch, in particular when the query k-mers are part of a longer string. This scenario is exploited by hashing-based k-mer lookup methods [28, 31] via use of rolling hash functions, similar to Karp-Rabin [22]. It can also be exploited by SBWT-based methods by storing a bit vector indicating the boundaries of k-mers sharing the same $(k-1)$-length prefix [4]. In this paper we explore a generalization of this approach that instead stores the longest common suffix of each k-mer with its colexicographic predecessor. This data structure was used recently in the context of computing signatures of k-mers called *finimizers* [2]. We apply it here directly to the k-mer lookup problem.

3. We then show how to combine vertical search with streaming horizontal search. This hybrid approach achieves even greater throughput and is 3-4× faster than hashing-based k-mer lookup methods.

We emphasise that while our focus is on the SBWT in this paper, we expect many of our techniques to translate to other forms of BWT index. We return to this subject briefly toward the end of the paper.

Roadmap This article is organized as follows. In the remainder of this section we review the sparse related work on batch query processing for BWT-based indexes, before laying down notation and important preliminary concepts in Section 2. Section 3 describes horizontal and vertical k-mer lookup for a batch of query k-mers. Section 4 then extends these approaches to an important special case where sub-batches of the query k-mers come overlapping one another, i.e., embedded in longer strings. This section shows how vertical search and streaming search can be effectively combined. Section 5 reports on extensive experiments measuring the performance of our new techniques. Conclusions, reflections, and avenues for future work are then offered.

Related Work For many important problems, such as, e.g., nearest neighbor queries [19], selection [8], and predecessor queries [6], superior performance can be achieved, both in theory and in practice, by considering a batch of queries holistically.

As noted above, prior work on batched processing for indexed pattern matching is sparse. Popular BWT-based read-alignment tools such as BowTie [23], BWA [25], Soap [26], and pseudoalignment tools like Themisto [5] use horizontal processing—that is, they process one pattern at a time.

In theoretical work, Gagie et al. [15] show that given the LZ parsing of a concatenation of t patterns of total length ℓ and maximum individual length m, the number of occurrences of each pattern can be computed in a total time of $O((z+t)\log \ell \log m \log^{1+\epsilon} n)$, where z is the number of phrases in the parse. We remark that computing the LZ parsing takes $O(\ell)$ time (see, e.g., [21]). In a similar vein, Gog et al. [17] explore the scenario where the pattern batch itself is allowed to be indexed. Such heavy preprocessing may have applications where the same batch is to be searched for in many different texts, but is not of interest to us here.

2 Preliminaries

Strings Because we are motivated by applications in genomic sequence analysis, throughout this paper, we assume a *string* $X[1..n]$ is a sequence of $|X|$ symbols over the DNA alphabet $\Sigma = \mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}$ and $\sigma = |\Sigma| = 4$. The empty string is denoted ϵ and $|\epsilon| = 0$. The *substring* of X starting at symbol i and ending at symbol j is denoted $X[i..j]$. A *prefix* is a substring starting at position 1 and a *suffix* is a substring ending at position n. The *colexicographic order* of two strings corresponds to the lexicographic order of their reverse strings. We define a k-mer as a (sub)string of length k, and the set of distinct k-mers occurring in a string X is referred to as the *k-spectrum* of X.

Definition 2.1. *(k-Spectrum)* The k-spectrum of a string X, denoted with $S_k(X)$, is the set of all distinct k-mers $\{X[i..i+k-1] \mid i = 1, \ldots, |X|-k+1\}$. The k-spectrum $S_k(X_1, \ldots, X_m)$ of a set of m strings

$X_1, \ldots, X_m$ is the union $\bigcup_{i=1}^{m} S_k(X_i)$.

The following definition enables us to search a k-spectrum with the SBWT.

Definition 2.2. *(Padded k-Spectrum)* Let $R = S_k(X_1, \ldots, X_m)$ be the k-spectrum of the set of strings $X_1, \ldots, X_m$, with alphabet Σ, and let $R^0 \subseteq R$ be the set of k-mers Y such that $Y[1..k-1]$ is not a suffix of any k-mer in R. The padded k-spectrum is the set $S_k^+(X_1, \ldots, X_m) = R \cup \{\$^k\} \cup \bigcup_{Y\,2\,R'} \{\$^k \;^iY[1..i] \mid i = 1, \ldots, k-1\}$, where \$ is a special character that is not found in the alphabet and is smaller than all characters of the alphabet.

For example, if $X_1 = \mathsf{AGTC}$, $X_2 = \mathsf{GAGT}$ and $k = 3$, then $S_3(X_1, X_2) = \{\mathsf{AGT}, \mathsf{GTC}, \mathsf{GAG}\}$ and $S_3^+(X_1, X_2) = \{\mathsf{AGT}, \mathsf{GTC}, \mathsf{GAG}, \$\$\$, \$\$\mathsf{G}, \$\mathsf{GA}\}$.

Spectral Burrows-Wheeler transform (SBWT) We can now define the Spectral Burrows-Wheeler transform. This definition corresponds to the multi-SBWT definition of Alanko et al. [4].

Definition 2.3. *(Spectral Burrows-Wheeler Transform (SBWT))* Let R^+ be a padded k-spectrum and let $X_1, \ldots, X_{jRj}$ be the colexicographically sorted elements of R^+. The SBWT is the sequence of sets of characters $A_1, \ldots, A_{jRj}$ with $A_i \subseteq \Sigma$ such that $A_i = \emptyset$ if $i > 1$ and $X_i[2..k] = X_{i\ 1}[2..k]$, otherwise $A_i = \{c \in \Sigma \mid X_i[2..k]c \in R^+\}$.

For example, the SBWT of $\{\mathsf{AGTC}, \mathsf{GAGT}, \mathsf{AAGT}\}$, with $k = 3$, is: $\{\mathsf{A}, \mathsf{G}\}$,$\{\mathsf{A}\}$,$\{\mathsf{G}\}$,$\{\mathsf{G}\}$,$\{\}$,$\{\mathsf{A}\}$,$\{\mathsf{T}\}$,$\{\}$,$\{\mathsf{C}\}$ (see also Figure 2).

In this paper the primary operation we aim to support is the *k-mer lookup* query, defined as follows.

Definition 2.4. *(k-mer lookup query)* Given an input a string S of length k, a k-mer lookup query returns the colexicographic rank of S in the underlying spectrum of the SBWT, or $\perp$ if the string is not in the spectrum.

We can think of the SBWT as encoding the set R^+ of k-mers as a colexicographically sorted list. In the context of k-mer lookup, the SBWT allows us to navigate that ordered list via the operation *ExtendRight*, defined as follows.

Definition 2.5. *(ExtendRight)* Let $[s, e]_\alpha$ be the *colexicographic interval* of string α, where s and e are respectively the colexicographic ranks of the smallest and largest k-mer in the SBWT suffixed by the substring α. *ExtendRight*$([s, e]_\alpha, c)$ denotes the *right extension* of the interval $[s, e]_\alpha$ with a character $c \in \Sigma$. This outputs the interval $[s^0, e^0]_{\alpha c}$, or $\perp$ if no such interval exists.

ExtendRight can be answered in $O(1)$ time using two *subset rank* operations on the SBWT. We first define the *rank* operation.

Let X be a bit string (or bit vector) of length n, for every index $i \leq n$ and $x \in \{0, 1\}$, $\text{rank}_x(i)$ is equal to the number of x's among the first i bits of X.

Definition 2.6. *(Subset rank query)* Let $X_1, \ldots, X_n$ be a sequence of subsets of characters from an alphabet Σ. A subset rank query takes as input an index i and a character $c \in \Sigma$, and returns the number of subsets X_j with $j \leq i$ such that $c \in X_j$.

With this, ExtendRight$([s, e]_\alpha, c)$ can be computed using the formulas $s^0 = 1 + C[c] + \text{subsetrank}_c(s-1) + 1$ and $e^0 = 1 + C[c] + \text{subsetrank}_c(e)$ [4], where $C[c]$ is the number of occurrences of characters smaller than c in the SBWT sequence. In our running example $C[\mathsf{A}] = 0$, $C[\mathsf{C}] = 3$, $C[\mathsf{G}] = 4$ and $C[\mathsf{T}] = 7$.

The subset sequence of the SBWT can be represented in many ways to support subset rank queries [3, 4]. In this paper, we focus on the *matrix representation* (Figure 2) and the *split representation* (Figure 1), both of which sit on the time–space Pareto curve [4]: matrix uses ∼5 bits per k-mer and is the fastest SBWT index we know of, while split uses around ∼2.5 bits per k-mer and is 3-4 times slower than matrix.

Definition 2.7. *(Plain Matrix representation)* The plain matrix representation of the SBWT sequence is a binary matrix M with σ rows and n columns, such that the value of $M[i][j]$ is set to 1 iff subset X_j includes the i^{th} character in the alphabet.

The rows of M are indexed for constant time rank queries [29] (we use the implementation from [18]). The subset rank query for the i-th character of the alphabet up to index j is answered in constant time with a single rank query on row $M[i]$ up to index j, $\text{rank}_{M[i]}(j)$.

The so-called *split representation* is a modification of the plain matrix representation aimed at exploiting the property that many of the SBWT subsets are singleton because the total number of elements in the sets is one less than the number of subsets.

Definition 2.8. *(Split representation)* Let M be the submatrix of matrix M, in the plain matrix representation, that contains only the columns of M with only a single 1-bit set, and let M^+ be the submatrix of M containing the remaining columns. Let B be a bit vector of length n, marking with 1-bits the columns of M that are in M^+. M is replaced by a string W which is the concatenation of the characters corresponding to the marked bits. The split representation of the SBWT consists of W, M^+ and B, and rank-support structures.

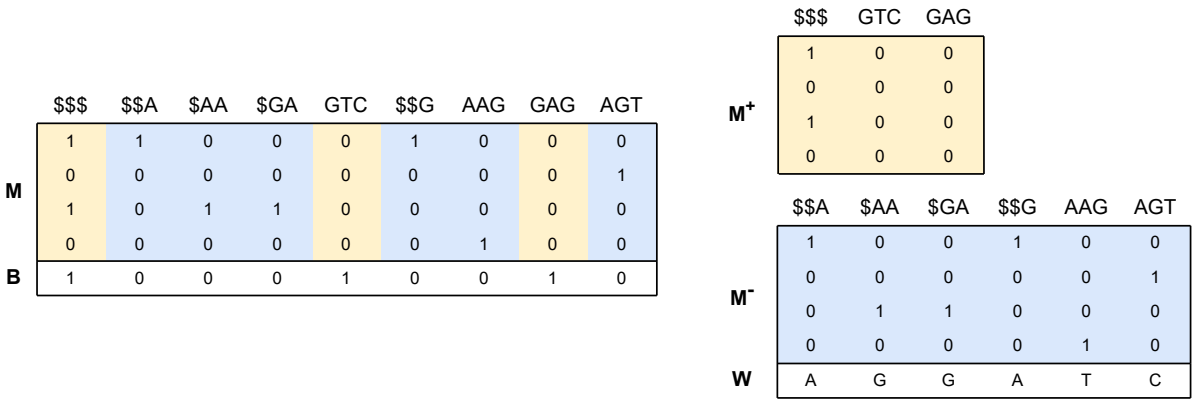

Figure 1: The Split representation of the SBWT of {AGTC, GAGT, AAGT} with $k = 3$. The index consists of only B, M^+ and W.

$$$	$$A	$AA	$GA	GTC	$$G	AAG	GAG	AGT
1	1	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	1
1	0	1	1	0	0	0	0	0
0	0	0	0	0	0	1	0	0

Figure 2: The Plain Matrix representation of the SBWT of {AGTC, GAGT, AAGT} with $k = 3$.

Every row in M^+ is indexed like in the plain matrix representation, just like B, and W is indexed as a wavelet tree. In this paper in particular we will use a split representation in which the bit vector B is stored compressed in the Elias-Fano data structure [30, 27].

3 Batched k-mer lookup queries

A batched k-mer lookup algorithm takes as input a set of b patterns, each of length k. The standard way to process a batch, which we call *horizontal search* is to iterate over the patterns and process each pattern fully before moving to the next. See Algorithm 1.

The algorithm involves two nested loops. The outer loop simply iterates through the patterns. The inner loop performs a k-mer lookup on the current pattern. Processing of a pattern involves iterating over its symbols from left to right and issuing up to $2k$ subset rank queries (i.e., k *ExtendRight* operations) on the SBWT sequence. The invariant maintained by the inner loop—which we do not prove here—is that at the end of the ith iteration, after processing prefix $P[1..i]$ of the pattern, the interval $[s, e]$ corresponds to the colexicographical interval of the k-mer spectrum

Algorithm 1 The horizontal (i.e. naive) SBWT batch search algorithm for a set of b patterns.
Input: b patterns, each of length k.
Output: The colexicographic rank of each pattern found in the underlying spectrum of the SBWT, or 0 if the pattern is not in the spectrum.

```
function HorizontalSearch(B[1, b][1, k])
    results → array of size b initialized to 0
    for i = 1, ..., b do
        P ← B[i]
        [s, e] ← [1, n]
        for j = 1, ..., k do
            c ← P[j]
            s ← 1 + C[c] + subsetrank_c(s − 1) + 1
            e ← 1 + C[c] + subsetrank_c(e)
            if e < s then
                s → 0
                break
        results[i] ← s
    output results
```

containing k-mers having $P[1..i]$ as a suffix.[2] In general, the colexicographical intervals of $P[1..i]$ and $P[1..i+1]$ can be in very different parts of colexicographic space, and so accesses to the SBWT sequence (and data structures built on it) when answering subset rank queries have poor locality of memory reference, as Figure 3 left illustrates. Each interval in the figure corresponds to an (s, e) pair in one iteration of the inner loop in the pseudocode. As the prefix processed increases, the interval tends to become narrower and

[2]The inner loop can exit early (i.e. before processing all k symbols of the k-mer) if $e > s$, in which case the k-mer does not exist in the spectrum.

only one cache miss occurs (for the subset rank query at s), but early rounds of the algorithm cause two cache misses, one each for the subset rank queries at s and e.

3.1 Presorting batches An intuitive and straightforward band-aid for the unattractive memory access behaviour described above is to sort the patterns in the batch into lexicographical order before starting the outer loop. The philosophy is that if patterns are sorted in lexicographical order, then, in a large batch, adjacent patterns will share long common prefixes. Thus, some of the colexicographical intervals accessed by the query algorithm will be shared and so may already reside in cache (from the processing of the previous pattern)—the effect, in other words, is to increase the temporal locality of memory accesses. The price, of course, is the cost of sorting, which may be non-trivial for a large batch of k-mers.

3.2 Vertical batch search Algorithm 2 lists an alternative SBWT search algorithm—vertical search. Unlike in horizontal search, each round of the outer loop in vertical search progresses the search for *all* the patterns in the batch by one character.

The operation of the algorithm proceeds in a manner analogous to a radix sort of the patterns in the batch. We maintain σ queues, Q_c, one for each character c in the alphabet. Q_c is initialised to contain all patterns starting with c along with, as satellite data, an interval containing all k-mers in the padded k-spectrum that end with c. As the algorithm proceeds, these intervals are maintained via right extensions (implemented as two subset rank queries) using the next character of the corresponding pattern, so that at the end of round i, queue Q_c contains all the patterns P that have $P[i] = c$ along with the colex interval in the SBWT containing all k-mers having $P[1..i]$ as a suffix. Critically, as we prove below, intervals in each queue are in non-decreasing order. Thus, the rank queries issued to update intervals from one round to the next are issued in a cache-friendly pattern across the SBWT and its rank structures.

In a final step we output the position in the SBWT of found patterns (their colex rank) in their original order, or 0 if the pattern is not found.

3.2.1 Memory access pattern We now prove that in each round of vertical batch search, the intervals come in non-decreasing order. This is the basis for the efficient cache behaviour of the algorithm. An interval $[s_1, e_1]$ is considered smaller than $[s_2, e_2]$ iff $e_1 < s_2$.

LEMMA 1. *In a given round i of the for-loop on line 5 of Algorithm 2, the intervals $[s, e]$ popped from the queues are in non-decreasing order.*

Algorithm 2 The vertical SBWT batch search algorithm for a batch of b patterns, each of length k.

function VERTICALSEARCH($B[1, b][1, k]$)
1: $results \rightarrow$ array of size b initialized to 0
2: **for** $j = 1, \ldots, b$ **do**
3: $\quad P \leftarrow B[j]$ ▷ $j = k$-mer id
4: $\quad Q_{P[1]}$.append($C[P[1]], C[P[1]+1], P, j$) [a]
5: **for** $i = 2, \ldots, k$ **do**
6: $\quad$ **for each** $c \in \Sigma$ in order **do** ▷ Σ = A, C, G, T
7: $\quad\quad$ **while** $Q_c \neq \emptyset$ **do**
8: $\quad\quad\quad [s, e, P, j] \leftarrow Q_c$.pop()
9: $\quad\quad\quad s \leftarrow 1 + C[P[i]] + \text{subsetrank}_{P[i]}(s-1) + 1$
10: $\quad\quad\quad e \leftarrow 1 + C[P[i]] + \text{subsetrank}_{P[i]}(e)$
11: $\quad\quad\quad$ **if** $e \geq s$ **then** ▷ symbols left to match
12: $\quad\quad\quad\quad Q^{\emptyset}_{P[i]}$.append($s, e, P, j$)
13: $\quad$ **for each** $c \in \Sigma$ **do**
14: $\quad\quad$ swap($Q_c, Q^{\emptyset}_c$)
15: **for each** $c \in \Sigma$ **do**
16: $\quad$ **while** $Q_c \neq \emptyset$ **do**
17: $\quad\quad [s, e, P, j] \leftarrow Q_c$.pop()
18: $\quad\quad results[j] = s$
19: **output** *results*

[a] $C[\sigma + 1] = n$, number of k-mers in the extended k-spectrum.

Proof. Every colexicographic interval in Q_c corresponds to a pattern ending in character c, so therefore for two symbols $c_1 < c_2$, all intervals in Q_{c_1} are smaller than all intervals in Q_{c_2}. The queues are processed in increasing order of c, so it remains to show that the intervals inside each individual queue are in sorted order.

We proceed by induction. On the first round, the property holds since all intervals in each individual queue are equal. Assume now that this property holds at round $i-1$. Consider the subsequence of intervals popped in round $i-1$ that extend successfully with a fixed character c. The queue Q_c in round i contains the extensions from these intervals with c, in the order of the subsequence. Take any two intervals $[s_1, e_1]$ and $[s_2, e_2]$ that are consecutive in this subsequence in round $i-1$. By the induction assumption, we have either $[s_1, e_1] = [s_2, e_2]$, or $e_1 < s_2$. In the former case, the extended intervals will be the same, and the non-decreasing order is maintained for the next round. In the latter case, since $e_1 < s_2$, we have $subsetrank_c(e_1) < subsetrank_c(s_2 - 1) + 1$, and the order is maintained. ∎

3.3 Resolving rank queries via scanning As Lemma 1 establishes, in a given round of vertical

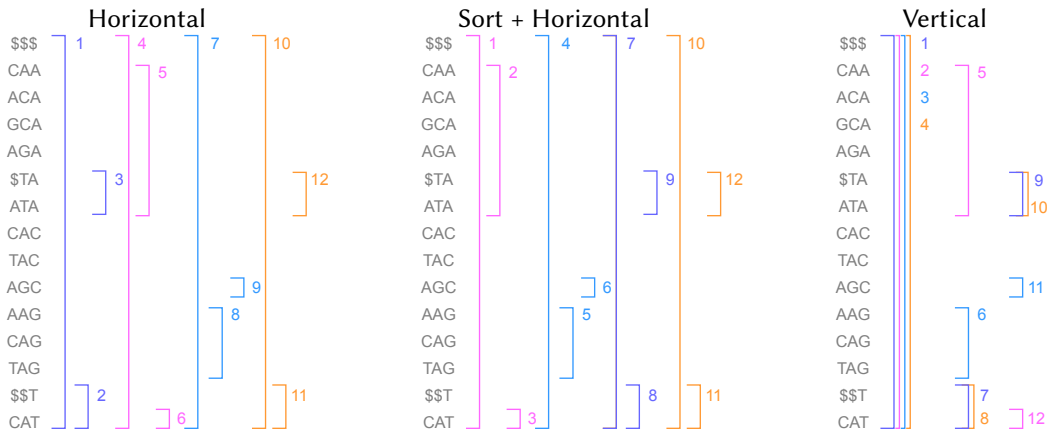

Figure 3: The intervals corresponding to pairs of SBWT subset rank queries issued in horizontal (left) and vertical (right) SBWT batch search for the patterns TAC, ATA, GCA, TAG. The middle figure shows the order of the query intervals if the patterns in the batch are first sorted. Numbers next to each interval indicate the order in which the algorithms resolve them. A similar illustration derived from searching for a larger batch of patterns over larger data set in each mode can be seen in Figure 4.

search, intervals are processed in non-decreasing order. In particular, for two subsequent intervals $[s_j, e_j]$ and $[s_{j+1}, e_{j+1}]$, either the intervals are equal, or $s_{j+1} > e_j$. In the first case, if the rank queries issued on the two intervals are for the same symbol c, once the answer is known for $[s_j, e_j]$, no further computation is needed to resolve $[s_{j+1}, e_{j+1}]$. In the second case, again assuming the same symbol c, the answer for $\text{rank}(s_{j+1}, c)$ is equal to $\text{rank}(e_j, c)$ plus the number of occurrences of c in the SBWT sequence between e_j and s_{j+1}.

This suggests that, for a big enough batch, in which intervals are well spread over the colexicographical space of the k-spectrum encoded by the SBWT, it may be preferable to simply scan the SBWT sequence, maintaining a cumulative sum of each symbol encountered up to the current scan position—rather than answering queries via rank structures built in preprocessing. The aim is to both save space (fast rank support structures incur 10-25% overhead) and reduce computation (nearby rank queries perform the same arithmetic within the rank structure). The precise details of the scan depend, of course, on the SBWT encoding used. In the case of Plain Matrix, a pointer into each bitvector is maintained. For Split pointers into B, M^+, and W are required.

4 Batches of overlapping k-mers

In many genomics applications, the k-mers to be searched for lie within longer strings, and so overlap each other. For example, we may be presented with the string $P = \mathsf{CAGCATAC}$ and asked to query its constituent 3-mers, CAG, AGC, GCA, and so on, each of which overlap by two symbols. In a typical scenario we are given a large set length-100 strings, called *reads*, and are asked to query for all the 30-mers that they contain. One would like to exploit the clear relatedness (i.e. overlap) of the k-mers within each read to reduce computation. We refer to this as a *streaming query*.

4.1 Streaming horizontal search

The approach to streaming search taken by the SBWT-based k-mer lookup tool Themisto [5] is to store a bit vector of length m that allows us to go from the index of a k-mer x in the SBWT to the indices of the range of k-mers that have $x[2..k]$ as a suffix, and then execute one more iteration of the inner loop in Algorithm 1 from this range with the next character in the query. The bit vector marks the first set of the SBWT and all sets where the suffix of length $k-1$ of the corresponding k-mer is different from the suffix of the colex previous k-mer. Now, if we are at index i corresponding to k-mer x, the range of $(k-1)$-mer $x[2..k]$ is $[pred(B, i), succ(B, i)-1]$, where $pred(B, i)$ and succ(B,i) are the indices of the previous and next 1-bits respectively in B from index i (if $B[i] = 1$, then $pred(B, i) = succ(B, i) = i$). Operations *pred* and *succ* are efficiently implemented as a scan of B in either direction from i because the gap between consecutive 1-bits in B can not be larger than $|\Sigma| + 1$ (the maximum number of distinct k-mers that have the same suffix of length $k-1$).

We now describe a generalization of the bit vector approach that instead makes use of the *Longest Common Suffix* (LCS) array of the SBWT, defined formally as follows.

DEFINITION 4.1. *(Longest common suffix array,* LCS *array)* Let $R^+ = \{X_1, \ldots, X_m\}$ be a set of k-mers encoded in the padded k-spectrum of an SBWT, such that X_i is the i-th element in colexicographic order. The LCS array is an array of length m such that $\text{LCS}[1] = 0$ and for any index $i > 1$, the value of $\text{LCS}[i]$ is the length of the longest common suffix of X_i and X_{i-1}.

The LCS array can be constructed from the SBWT in $O(n)$ time [1]. In the definition above, we consider the empty string as a valid common suffix of any two k-mers. Therefore the longest common suffix is well-defined for any pair of k-mers.

In addition to *ExtendRight*, the LCS allows us to navigate colexicographically-ordered padded k-mer spectrum also with what we call *ContractLeft*.

DEFINITION 4.2. *(ContractLeft)* Let $[s, e]_\alpha$ be the *colexicographic interval* of string α, where s and e are respectively the colexicographic ranks of the smallest and largest k-mer in the SBWT suffixed by the substring α. *ContractLeft*$([s, e]_\alpha, \ell)$ for $|\alpha| > \ell$ returns the interval $[s^0, e^0]_{\alpha[\ell+1..j\alpha j]}$.

ContractLeft can be answered with a *previous-smaller-value* (PSV) query, for s, by setting s^0 to be the largest position smaller or equal to s in LCS, such that $\text{LCS}[s^0] < |\alpha| - 1$, or to 1 if no such position exists. Symmetrically, e^0 can be found with a next-smaller-value (NSV) query for e. There are data structures capable of efficiently answering both PSV and NSV queries in constant time with minimal space overhead [14, 10]. In practice however, we implement left contractions by simply scanning the LCS array left from s and right from e. Since with biological data, in our algorithms, the vast majority of scans tend to be very short [2], this solution has proven to work well in practice. The scanning approach is also memory-local and is thus likely faster than more sophisticated data structures that have worst-case guarantees.

This combination of data structures was recently used for computing k-mer signatures called *finimizers* [2]. We apply it here directly to k-mer lookup.

ExtendRight queries are here solved with 2 subset rank queries on the SBWT as in section 3. Algorithm 3 lists pseudocode. Whenever a right extension fails, we perform at least a left contraction before successfully right extending again.

THEOREM 4.1. *Given constant-time ContractLeft and ExtendRight queries on the padded k-spectrum R^+, Algorithm 3 solves k-mer lookup queries for query T in time $O(|T|)$.*

Proof. The invariant maintained is that at the end of iteration i, $[s, e]$ is the interval of the longest suffix of $T[1..i]$ that is suffix of at least one k-mer in the SBWT. If at iteration $i-1$ we have the interval of the longest match ending at $i-1$, then at iteration i, after lines 5–7, we have the interval of the longest match ending at $i-1$ that can be extended with $T[i]$. If the longest match ending at i is nonempty, the right extension succeeds and the invariant is maintained. The invariant holds with an empty match as the interval is set to $[1, n]$ after the last *ContractLeft*. Time is linear in $|T|$ as each query position is subject to at most one successful *ExtendRight* and *ContractLeft*. □

Algorithm 3 The SBWT streaming search algorithm for a query T.
Input: A string T.
Output: The colexicographic rank of each k-mer of T found in the underlying spectrum of the SBWT, or 0 if the pattern is not in the spectrum.

```
1:  results → array of size |T| − k + 1 initialized to 0
2:  [s, e] ← [1, n]
3:  d ← 0                                   ▷ Length of the current match
4:  for i = 1..|T| do
5:      while d > 0 and ExtendRight([s, e], T[i]) = ∅ do
6:          [s, e] ← ContractLeft([s, e], 1)
7:          d ← d − 1
8:      if ExtendRight([s, e], T[i]) ≠ ∅ then
9:          [s, e] ← ExtendRight([s, e], T[i])
10:         d ← min(k, d + 1)
11:     if i ≥ k and d = k then
12:         results[i] ← s
13: output results
```

An important implementation detail for Algorithm 3 is that when a right extension fails in the while loop, we can store the ranks at the ends of the interval and maintain the ranks while scanning backward and forward during the left contraction. This way, we avoid issuing rank queries again for the next attempted right extension, and instead use the stored values. Likewise, we can reuse stored rank values when we apply the extension after the while loop.

4.2 Streaming vertical search A streaming vertical k-mer lookup algorithm takes as input a set of n query sequences of equal length $m \geq k$, and looks up the k-mers in a vertical left-to-right order. That is, the algorithm first processes the first nucleotide of every query, then the second nucleotide of every query, and so on. In short, the algorithm runs the horizontal streaming search algorithm described in Section 4.1 for each query, but with reordered computation to improve memory locality.

In more detail, the algorithm is a modification of the regular vertical search (Algorithm 2), with the change that the algorithm runs for $m \geq k$ rounds, and when a right extension fails, instead of giving up on the query, we issue left contraction queries until the right extension succeeds. With this, we maintain the invariant that at the end of iteration i, we have the colexicographic interval of the longest match ending at position i in every query. Moreover, by keeping track of the current match length in each query, we know whenever we have a full k-mer match, and can report query answers accordingly.

The items pushed to the work queues now contain the full state of the horizontal streaming algorithm, including the current colex interval, the match length, the position in the query, the query string itself for fast local access to the next character, and a pointer to where to write the next query answer. As a small but important implementation detail, we also include a small local write buffer with each query and write answers in batches of 8 to reduce cache misses.

Memory access pattern The memory access pattern of streaming vertical search is slightly less predictable than that of regular vertical search. Unlike in regular vertical search, there is no longer a guarantee that the intervals processed on each round come in non-decreasing order. A weaker property holds instead, that while two intervals may now nest, they cannot swap places completely:

LEMMA 2. *On a given round i, if interval $[s_1, e_1]$ is popped before $[s_2, e_2]$, then $e_2 \geq s_1$.*

Proof. Like in the proof of Lemma 1, it is enough to show that the individual queues are sorted. We proceed by induction on the round number i. The base case holds since initially, the intervals in each individual queue are equal. Assume now that the Lemma holds at round $i-1$. Consider the subsequence S_c of intervals popped in round $i-1$ that extend with a fixed character c, possibly after some number of left contractions. The queue Q_c in round i contains the intervals after left contracting and right extending each interval from S_c. Take any two intervals $[s_1, e_1]$ and $[s_2, e_2]$ that are consecutive in S_c and let $[s_1', e_1']$ and $[s_2', e_2']$ be the intervals after the possible left contractions. A left contraction moves start points to the left and end points to the right, so we have $e_2' \geq e_2$ and $s_1' \leq s_1$, and therefore by the induction assumption $e_2' \geq s_1'$. Let e_2'' and s_1'' be the end point and start point after the right extension. We have $e_2'' = 1 + C[c] + \text{subsetrank}_c(e_2') \geq 1 + C[c] + \text{subsetrank}_c(s_1' - 1) + 1 = s_1''$, the order property in the Lemma continues to hold. ∎

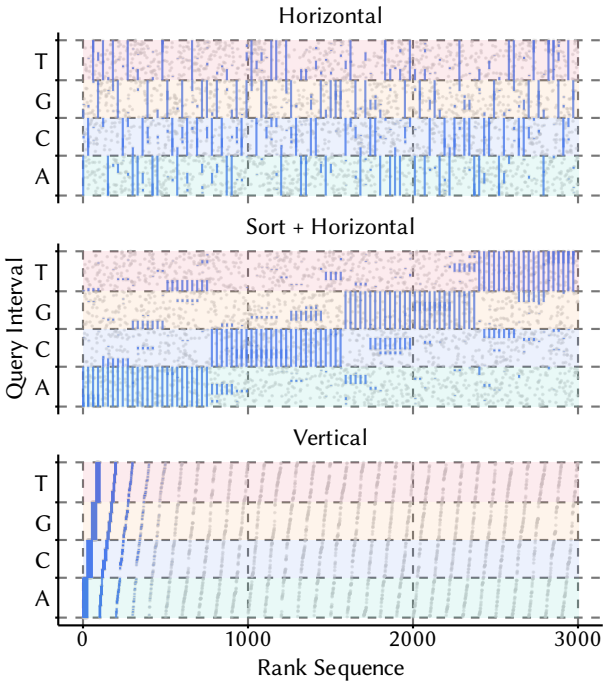

Figure 4: Comparing subset rank query intervals across horizontal, horizontal after sorting, and vertical batch querying over 100 positive queries on the E. coli data. Lines indicate query intervals (and points illustrate small intervals that would otherwise be imperceivable).

5 Experiments

5.1 Experimental Machine We used a Linux server with two Intel Xeon Gold 6144 CPUs (3.5GHz) and 512 GiB of memory. Only a single thread of execution was used. The compiler was `g++` version 8.4.0 (flags `-O3 -march=native`). Runtimes were measured with calls to the high-resolution clock (`std::chrono`). Reported times do not include time for I/O.

5.2 Datasets We ran experiments on three data sets representing different types of sequencing data typical of genomics applications. The unique 31-mer counts below include both DNA strands.

1. **E. coli** A pangenome of 3682 E. coli genomes. [3] The collection includes 745,409 sequences with a total length of 18,957,578,183 characters, resulting in 341,297,220 unique 31-mers.
2. **Metagenome** A set of 17,336,887 Illumina HiSeq reads of length 502 sampled from the human gut (SRA id ERR5035349) [20] of length 8,703,117,274 characters (5,523,047,870 distinct 31-mers).

[3] The collection is available at zenodo.org/record/6577997.

3. **Blackwell** A collection of 639,981 high-quality genomes from 661k bacterial genomes dataset of Blackwell et al. [7]. It has 2,477,615,026,052 characters and 71 billion unique 31-mers.

5.3 Queries We ran our experiments with $k = 31$, which represents a typical scenario in genomics applications. We generated three types of query sets of single k-mers: *Negative queries*, which are randomly generated sequences of length k, highly unlikely to occur in the data; *Positive queries*, generated by randomly sampling k-mers from each dataset; *Mixed queries*, half randomly generated and half sampled k-mers, randomly ordered. Our query sets contain 1, 10, and 100 million queries, as well as 1 billion queries.

To test streaming search we used reads of length $L = 200$. We generated two types of query sets: *Negative reads*, which are randomly generated sequences of length L, in which each k-mer is highly unlikely to occur in the data; and *Positive reads*, generated by randomly sampling strings of length L from each dataset. Our sets contain 1, 10, and 100 million reads.

5.4 Indexes For each data set, we built the two SBWT representations from the study on SBWT indexing by Alanko et al. [4]: Plain-Matrix (PM) and Elias-Fano Split (EFS). We constructed these indexes setting $k = 31$ and enabling reverse complements.

We also use SSHash [31], a k-mer lookup method based on minimal perfect hashing, as a performance baseline. It is the fastest compact k-mer lookup method we know of. SSHash was run on Eulertigs [32]—minimal a set of strings with the same k-mers as the input strings—built with GGCAT [11]. We set $m = 16, 17, 19$ on the three datasets, respectively, following the author's advice to use $m = \lceil \log_4(N) \rceil + 1$, where N is the number of symbols in the input to SSHash.

6 Results

6.1 Querying Individual k-mers Our first series of experiments focus on the task of processing a large batch of *individual* query k-mers. In particular, we measure the per-query latency (i.e. runtime) of processing the entire batch of b patterns (a $b \times k$ matrix of characters) over a series of different batches and indexes.

Before we discuss the results, we briefly re-iterate the methods that are under consideration. Horizontal processing uses the SBWT in a "typical" manner, processing each pattern of length k in the original batch order; Sort + Horizontal is the same, but the batch is first sorted lexicographically (via radix sort) before search

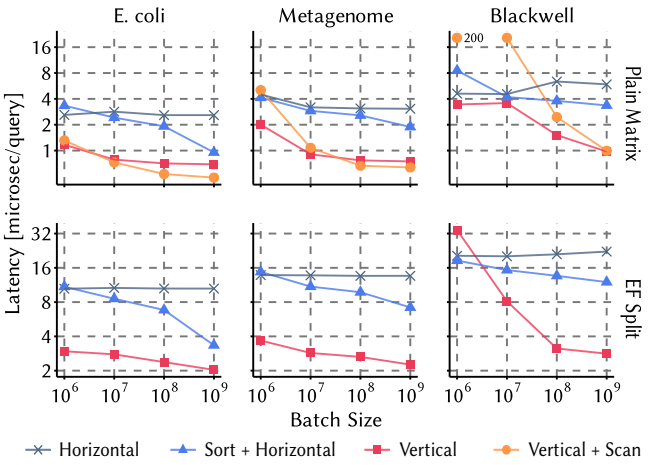

Figure 5: Query time (μsec/query) of batch processing strategies as a function of the increasing batch size, on both the Plain-Matrix (top) and EF-Split (bottom) indexes, two datasets, and four sets of positive 31-mers.

commences;[4] Vertical represents the proposed method in Algorithm 2; finally, SSHash, is our hashing baseline. For Plain Matrix, we also measure the cost of vertical processing with the additional scanning optimization described in Section 3.3, denoted Vertical + Scan.

Table 1 compares the average latency (per k-mer) across three batches of 10^9 individual k-mers. Interestingly, sorting the batch does yield benefits to all query types and indexes; we observed sorting to cost around 200 nanoseconds per k-mer, a modest pre-processing cost which is, in turn, recouped via faster rank queries. However, vertical batch processing clearly outperforms horizontal processing, even if the batch is sorted (with the exception of negative queries on the E. coli collection). We also see that the scanning optimization to vertical search is helpful on the smaller indexes, but is less effective on the big Blackwell dataset, where the distance between rank queries—and so the length of scans—can become very large.

Figure 4 visually supports the observed speedups—reordering the computation with sorting clearly improves the access coherence of the rank queries; but further coherence is achieved through the vertical batch scheme. Figure 5 shows how the latency changes with increasing batch size b. As expected, larger batches result in faster querying, except for with standard horizontal processing. Nonetheless, Vertical processing is always significantly faster than Sort + Horizontal in all

[4] In both Horizontal and Sort + Horizontal, we bootstrap the search by precalculating the colex intervals of all 8-mers and start the searches from there.

	Approach	E. coli			Metagenome			Blackwell		
		POS	MIX	NEG	POS	MIX	NEG	POS	MIX	NEG
	SSHash	1.70	1.70	1.72	1.96	2.04	2.10	3.15	3.44	3.28
PM	Horizontal	2.59	1.70	0.79	3.07	2.07	1.09	5.91	4.07	2.78
PM	Sort + Horizontal	0.95	0.82	0.42	1.89	1.34	0.57	3.36	2.69	1.41
PM	Vertical	0.69	0.75	0.49	0.75	0.69	0.48	0.97	0.89	0.72
PM	Vertical + Scan	0.48	0.50	0.33	0.64	0.51	0.33	0.99	1.00	0.79
EFS	Horizontal	10.52	6.92	3.22	13.61	9.42	5.21	22.22	15.74	11.87
EFS	Sort + Horizontal	3.34	2.60	0.85	7.18	4.73	1.34	11.99	8.66	3.94
EFS	Vertical	2.03	1.60	1.02	2.26	1.78	1.14	2.82	2.09	1.73

Table 1: Query time (μsec/query) of batch processing strategies on both the Plain-Matrix (top) and EF-Split (bottom) indexes, three datasets, and three types of query batches. All batches contain 10^9 individual 31-mers.

	Approach	E. coli		Metagenome		Blackwell	
		POS	NEG	POS	NEG	POS	NEG
$L = 200$	SSHash	0.32	0.52	0.25	0.55	0.38	0.80
$L = 200$	Horizontal-S	0.15	0.41	0.17	0.59	0.26	1.12
$L = 200$	Horizontal-S + LCS	0.14	0.26	0.17	0.30	0.37	0.98
$L = 200$	Vertical-S + LCS	0.12	0.20	0.13	0.22	0.17	0.82

Table 2: Query time (μsec/query) of various streaming batch processing strategies on the Plain-Matrix index, three datasets, and two types of query batches. Positive batches contain 10^8 reads, and negative batches contain 10^6 reads. Reads are of length $L = 200$. Results are consistent for smaller read sets.

but one run (the smallest batch on the largest SBWT with the EF Split index). Here again we can see that the efficacy of scanning (the orange line in the figure) depends critically on the size of the query batch relative to the index size. When batch size is small relative to index size, the resulting large scans are slower than using an index to support rank.

Table 1 shows significant improvements as measured on large batches containing 10^9 k-mers. EF-Split shows the most dramatic improvement in the shift from horizontal to vertical processing, with an 8× speed up for positive queries on the Blackwell dataset. Plain-Matrix is always clearly faster, but uses around 40% more memory than EF-Split. For brevity, detailed space usage numbers are not shown, but in summary: Plain-Matrix indexes are $1.22-1.45\times$ the size of SSHash; and EF-Split indexes are smaller, $0.80-1.00\times$ the size of SSHash.

6.2 Streaming search In the second set of experiments, queries are no longer a batch of single k-mers, but a batch of *reads* formed by overlapping k-mers. As before we measure per-k-mer latency of processing the set of reads over different batches and search methods.

We compare four methods: Horizontal-S denotes the standard processing reads using the SBWT with streaming support, described in [4] and at the start of Section 4.1; Horizontal-S + LCS represents the method proposed in Algorithm 3; Vertical-S + LCS combines the second method listed here with vertical batch search, as described in Section 4.2; and finally SSHash.

Results are presented for the Plain-Matrix representation. Table 2 offers a comparison of the average latency per k-mer across batches of 100 million positive and 1 million negative reads. Vertical-S + LCS appears to be consistently faster than the other methods and is roughly 2.5× faster than SSHash in these experiments.

Figure 6 shows the sortedness of intervals during a run of Vertical-S + LCS. We see that even though the memory access pattern guarantee is weaker than in Vertical, the sortedness of intervals does not deteriorate anymore after approximately 20 rounds, and in any case approximately 95% or more of the start points and end points still come in sorted order.

7 Conclusions and Future Work

Remarkably, despite 25 years of research on BWT-based text indexing, this is the first focused study examining batched pattern matching with those indexes.

We have described vertical batch processing, a careful reorganization of computation that drastically improves locality of memory reference compared to processing queries independently.

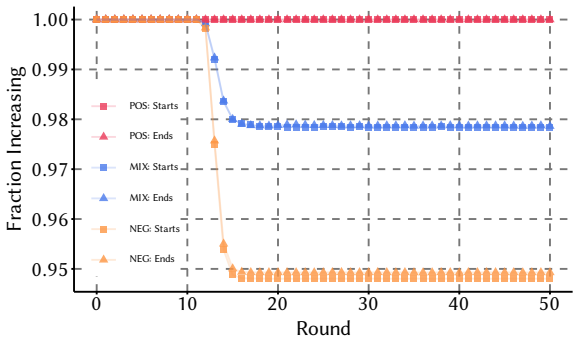

Figure 6: The sortedness of start and end points of intervals during streaming vertical search against the E. coli dataset. The sortedness of start points (or end points) during a round is defined as the fraction of consecutive start (or end) points popped from the queue that are in sorted order. Three read sets were queried: positive (red), negative (orange), and mixed (blue). The mixed dataset is the metagenomic read dataset, with a positivity rate of 0.54%.

We have also shown that in the scenario that the patterns in a batch come overlapping one another inside longer strings—an important special case in genomics applications—streaming search, a heuristic used in some practical tools, can be generalized and combined with vertical search to achieve even faster search times.

An important caveat of the batch processing framework is the increased memory usage required. Assuming k-mers are stored as bit-packed sequences, each k-mer may occupy 8 bytes (with $k = 31$). Then, a batch of 10^9 31-mers would require about 7.5GiB, which must be resident in main memory during querying, not including other support structures (such as a vector of answers). Our experiments show that using memory this way is worthwhile if increased throughput is the goal.

Our focus throughout this article has been on the SBWT and k-mer lookup. A direction for further research is to translate our techniques onto indexes based on the regular BWT that support search for patterns of variable length. Our successful combination of vertical and streaming search suggests this is possible. In the context of the SBWT, experiments on different values of k (rather than the single common value we have used in our experiments) would also be valuable.

With memory locality now vastly improved, it seems natural to explore parallelism in the context of batched search to further improve throughput. In the same vein, GPUs represent another interesting opportunity, where batch processing seems necessary to exploit massive parallelism present on those devices.

Acknowledgments We thank Brendan Gregg, Luke Gallagher, David Gwynne, and Alistair Moffat for helpful conversations, and Giulio Pibiri, whose careful reading materially improved our first manuscript. The third author was supported by the Google Research Scholar program. The code to reproduce the experiments is available at: https://github.com/JMMackenzie/BatchSBWT/

References

[1] Jarno N. Alanko, Elena Biagi, and Simon J. Puglisi. Longest common prefix arrays for succinct k-spectra. In *Proc. SPIRE*, LNCS 14240, pages 1–13. Springer, 2023.

[2] Jarno N. Alanko, Elena Biagi, and Simon J. Puglisi. Finimizers: Variable-length bounded-frequency minimizers for k-mer sets. *BioRχiv*, 02 2024.

[3] Jarno N. Alanko, Elena Biagi, Simon J. Puglisi, and Jaakko Vuohtoniemi. Subset wavelet trees. In *Proc. of the 21st International Symposium on Experimental Algorithms (SEA)*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[4] Jarno N. Alanko, Simon J. Puglisi, and Jaakko Vuohtoniemi. Small searchable k-spectra via subset rank queries on the spectral Burrows-Wheeler transform. In *Proc. of SIAM Conference on Applied and Computational Discrete Algorithms (ACDA)*, pages 225–236. Society for Industrial and Applied Mathematics, 2023.

[5] Jarno N. Alanko, Jaakko Vuohtoniemi, Tommi Mäklin, and Simon J. Puglisi. Themisto: a scalable colored k-mer index for sensitive pseudoalignment against hundreds of thousands of bacterial genomes. *Bioinformatics*, 39(S1):i260–i269, 2023.

[6] Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Dzejla Medjedovic, Pablo Montes, and Meng-Tsung Tsai. The batched predecessor problem in external memory. In *Proc. 22th Annual European Symposium (ESA)*, LNCS 8737, pages 112–124. Springer, 2014.

[7] Grace A. Blackwell, Martin Hunt, Kerri M. Malone, Leandro Lima, Gal Horesh, Blaise T. F. Alako, Nicholas R. Thomson, and Zamin Iqbal. Exploring bacterial diversity via a curated and searchable snapshot of archived DNA sequences. *PLOS Biology*, 19:1–16, 11 2021.

[8] Gerth Stølting Brodal and Sebastian Wild. Funnelselect: Cache-oblivious multiple selection. In *Proc. 31st Annual European Symposium on Algorithms (ESA)*, volume 274 of *LIPIcs*, pages 25:1–25:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[9] Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[10] Rodrigo Cánovas and Gonzalo Navarro. Practical compressed suffix trees. In Paola Festa, editor, *Proc.*

9th International Symposium Experimental Algorithms (SEA), volume 6049 of *Lecture Notes in Computer Science*, pages 94–105. Springer, 2010.

[11] Andrea Cracco and Alexandru Tomescu. Extremely fast construction and querying of compacted and colored de Bruijn graphs with GGCAT. *Genome res.*, 05 2023.

[12] Diego Díaz-Domínguez, Taku Onodera, Simon J. Puglisi, and Leena Salmela. Genome assembly with variable order de bruijn graphs. *BioRχiv*, 2022.

[13] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–398. IEEE Computer Society, 2000.

[14] Johannes Fischer. Combined data structure for previous-and next-smaller-values. *Theoretical Computer Science*, 412(22):2451–2456, 2011.

[15] Travis Gagie, Kalle Karhu, Juha Kärkkäinen, Veli Mäkinen, Leena Salmela, and Jorma Tarhio. Indexed multi-pattern matching. In *Proc. 10th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 7256, pages 399–407. Springer, 2012.

[16] E. Garrison, J. Sirén, A. Novak, et al. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature Biotechnology*, 36:875–879, 2018.

[17] Simon Gog, Kalle Karhu, Juha Kärkkäinen, Veli Mäkinen, and Niko Välimäki. Multi-pattern matching with bidirectional indexes. *J. Discrete Algorithms*, 24:26–39, 2014.

[18] Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Softw. Pract. Exp.*, 44(11):1287–1314, 2014.

[19] Ben Gum and Richard J. Lipton. Cheaper by the dozen: Batched algorithms. In *Proc. First SIAM International Conference on Data Mining (SDM)*, pages 1–11. SIAM, 2001.

[20] Ian B. Jeffery et al. Differences in fecal microbiomes and metabolomes of people with vs without irritable bowel syndrome and bile acid malabsorption. *Gastroenterology*, 158(4):1016–1028, 2020.

[21] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lazy lempel-ziv factorization algorithms. *ACM J. Exp. Algorithmics*, 21(1):2.4:1–2.4:19, 2016.

[22] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[23] Ben Langmead and Steven L. Salzberg. Fast gapped-read alignment with bowtie 2. *Nature Methods*, 9(4):357—359, 2012.

[24] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10:R25, 2009.

[25] Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinform.*, 26(5):589–595, 2010.

[26] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[27] Danyang Ma, Simon J. Puglisi, Rajeev Raman, and Bella Zhukova. On elias-fano for rank queries in fm-indexes. In *31st Data Compression Conference, DCC 2021, Snowbird, UT, USA, March 23-26, 2021*, pages 223–232. IEEE, 2021.

[28] Camille Marchet, Mael Kerbiriou, and Antoine Limasset. BLight: efficient exact associative structure for k-mers. *Bioinformatics*, 37(18):2858–2865, 2021.

[29] J. Ian Munro. Tables. In *Proc. 16th Foundations of Software Technology and Theoretical Computer Science*, LNCS 1180, pages 37–42. Springer, 1996.

[30] Gonzalo Navarro. *Compact Data Structures { A practical approach*. Cambridge University Press, 2016.

[31] G. E. Pibiri. Sparse and skew hashing of K-mers. *Bioinformatics*, 38(Suppl 1):i185–i194, 06 2022.

[32] Sebastian S. Schmidt and Jarno N. Alanko. Eulertigs: minimum plain text representation of k-mer sets without repetitions in linear time. *Algorithms Mol. Biol.*, 18(1):5, 2023.

[33] Jared T. Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22:549–556, 2012.