Algorithmica (2015) 71:258–278
DOI 10.1007/s00453-013-9792-1

Geometric BWT: Compressed Text Indexing via Sparse Suffixes and Range Searching

Yu-Feng Chien · Wing-Kai Hon · Rahul Shah · Sharma V. Thankachan · Jeffrey Scott Vitter

Received: 10 June 2012 / Accepted: 27 April 2013 / Published online: 10 May 2013

Abstract We introduce a new variant of the popular Burrows-Wheeler transform (BWT), called Geometric Burrows-Wheeler Transform (GBWT), which converts a text into a set of points in 2-dimensional geometry. We also introduce a reverse transform, called `Points2Text`, which converts a set of points into text. Using these two transforms, we show strong equivalence between data structural problems in geometric range searching and text pattern matching. This allows us to apply the lower bounds known in the field of orthogonal range searching to the problems in compressed text indexing. In addition, we give the first succinct (compact) index for I/O-efficient pattern matching in external memory, and show how this index can be further improved to achieve higher-order entropy compressed space.

Keywords Text indexing · Entropy compression · Geometric range searching

This work is supported in part by Taiwan NSC Grant 99-2221-E-007-123 (W.-K. Hon) and US NSF Grant CCF-1017623 (R. Shah).
Early parts of this work appeared in DCC 2008 [11] and SPIRE 2009 [26].

Y.-F. Chien · W.-K. Hon
Department of CS, National Tsing Hua University, Hsinchu, Taiwan

W.-K. Hon
e-mail: wkhon@cs.nthu.edu.tw

R. Shah (✉) · S.V. Thankachan
Department of CS, Louisiana State University, Baton Rouge, LA, USA
e-mail: rahul@csc.lsu.edu

S.V. Thankachan
e-mail: thanks@csc.lsu.edu

J.S. Vitter
Department of EECS, The University of Kansas, Lawrence, KS, USA
e-mail: jsv@ku.edu

1 Introduction

Given a text T of length n over an alphabet set Σ, we can construct the suffix trees [39, 46] and the suffix arrays [38] with $O(n \log n)$ bits of space for answering pattern matching queries efficiently. With the recent interest in *succinct* data structures (such as [30, 40]) where the main goal is to design data structures in the (information-theoretic) minimal space while supporting query in near-optimal time. For the text data T, this information theoretically minimum space would be $n \log |\Sigma|$ bits or even better in terms of (kth order) entropy compression of the text and it would be nH_k bits of space. The central question first addressed in compressed text indexing is whether an index size can be reduced to $O(n \log |\Sigma|)$ bits while supporting pattern matching queries efficiently over T. This was answered positively by Grossi and Vitter [20] and Ferragina and Manzini [14]; subsequently, an exciting field of compressed text indexing was established (see [34] for an excellent survey). One of the main techniques in this field has been to use the *Burrows-Wheeler Transform* (BWT) [8] to achieve space reduction.

All the above described indexes focussed on the internal memory model. However, there are increasing needs to deal with massive data sets that do not easily fit into the internal memory and thus the data and the index must be stored on secondary storage, such as disk drives, or in a distributed fashion in a network [29]. This leads to the requirement of an I/O-efficient external memory index. Ferragina and Grossi introduced an $O(n \log n)$-bit space index called string B-tree (SBT) [13] which can support pattern matching of a query pattern P in $O(|P|/B + \log_B n + occ/B)$,[1] where B is the block size measured in terms of words.

It seems difficult to get insight into this problem using the current BWT-based approach. In particular, since the BWT permutes the text, the contiguous characters in the text can end up being far apart in the permutation. If the index is placed in the external memory, every character match of a pattern could possibly come from a different disk page. This forms the key bottleneck and thus until now there were no successes in achieving an additive term like $|P|/B$ or $|P|/(B \log_{|\Sigma|} n)$ in query I/Os using this approach.

In this paper, we take a fresh look at this problem and develop an alternative approach for compressed text indexing. Similar to the Burrows-Wheeler transform (BWT), which transforms a text into another text, we define the *Geometric Burrows-Wheeler Transform* (GBWT), which transforms a text into a set of points. Conceptually, it is equivalent to taking Burrows-Wheeler transform on *blocked* text. But unlike BWT, which needs the text characters to be stored in a particular order, GBWT can maintain position information (for each character) explicitly within $O(n \log |\Sigma|)$ bits and hence it is more amenable for the external memory model; in particular, the $|P|/B$ (or even $|P|/(B \log_{|\Sigma|} n)$) additive term can now be achieved. Unfortunately our index cannot achieve optimality in the other two terms ($\log_B n$ and occ/B) simultaneously. The details of the practical implementation of our index is available in [12].

[1] Although the more optimal first term would be $|P|/(B \log_{|\Sigma|} n)$ because the block size is measured in terms of words and we assume the word-size to be $\log n$ bits and each character of the pattern takes $\log |\Sigma|$ bits.

We also define a reverse transform called `Points2Text`, which transforms a set of points into a text. Both GBWT and `Points2Text` preserve space up to a constant factor. These transformations allow results in orthogonal range searching and compressed text indexing to be interchangeably used for each other. Since orthogonal range searching is a very extensively studied field [1], many results (lower bounds and upper bounds) can now be translated to the field of compressed text indexing. Using `Points2Text` transform, we show that it is impossible to perform pattern matching queries in $O(|P|\log^{O(1)} n + occ/B)$ I/Os with an index using only succinct space. However, this is possible in a special case where the pattern is sufficiently long, or if we are willing to incur an extra $\tilde{O}(\sqrt{n/B})$ term in the query I/Os.[2]

The key components of our succinct index are very simple, which include a sparse suffix tree (or a sparse string B-tree), an orthogonal range searching data structure, and a four-russians lookup table. Furthermore, we introduce a variable-length blocking scheme for the text, so that by encoding each block using arithmetic coding, we can achieve higher order entropy compression.

Problem Definitions Given a text T with n characters over an alphabet Σ, and an input query pattern P, the pattern matching query of P on T returns the set $Q_{match}(T, P) = \{i \mid T[i..(i+|P|-1)] = P\}$. The problem of designing the data structure for this taking only $O(n\log|\Sigma|)$ bits is called the (compressed text indexing) problem.

In two-dimensional orthogonal range searching, we are given a set S of n points by their x and y coordinates: $S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$. The query Q_{range} specifies a rectangle $R = (x_\ell, x_r, y_\ell, y_r)$. The answer to the query is given by $Q_{range}(S, R) = \{(x_i, y_i) \in S \mid x_\ell \le x_i \le x_r,\ y_\ell \le y_i \le y_r\}$. Two specific versions of this query have been considered: counting and reporting. We call the problems of designing data structures on S for efficient orthogonal range queries the $\mathcal{RS2D}$ problem.

Our Results Based on our transforms, we show strong connections between the two problems. The following is a summary of our results:

1. We propose two (asymptotically) space-preserving transforms GBWT and `Points2Text`; these transforms are simple, quickly computable, and invertible. With these, we show that text pattern matching and orthogonal range searching can be reduced from one into the other.
2. Internal Memory Results
   (a) Using GBWT, we derive a succinct full text index of $O(n\log|\Sigma|)$ bits space and $O(|P| + (\log_{|\Sigma|} n + occ)\log n/\log\log n)$ query time.
   (b) Using `Points2Text` transform, we prove a space lower bound of $\Omega(n\log n/\log\log n)$ bits for any full text index with $O(|P|\log^{O(1)} n + occ)$ query time.
3. External Memory Results
   (a) Using GBWT, we derive a succinct full text index of $O(n\log|\Sigma|)$ bits space and $O(|P|/B + (\log_{|\Sigma|} n + occ)\log_B n)$ query I/O's. The I/O bound is further improved to $O(|P|/(B\log_{|\Sigma|} n) + \log_{|\Sigma|} n\log^{1+\epsilon} n + occ\log_B n)$.

[2]The notation $\tilde{O}$ ignores poly-logarithmic factors. Precisely, $\tilde{O}(f(n)) \equiv O(f(n)\log^{O(1)} n)$.

(b) Using `Points2Text` transform, we prove a space lower bound of $\Omega(n \log n/\log\log n)$ bits for any full text index with $O(|P|\log^{O(1)} n + occ/B)$ query I/O's.
(c) We also show how to achieve entropy compressed space of $O(nH_k + n) + o(n\log|\Sigma|)$ bits.

Here H_k denotes the kth-order entropy of the text T, occ denotes the number of occurrences of P in T, and B is the size of a disk page (measured in words).

Related Work A compact representation of suffix tree in secondary memory, called Compact Pat Tree [10] is an efficient practical index even though it does not provide any theoretical guarantees. Disk based suffix array data structures are also available in literature [7]. Mäkinen et al. proposed an external memory version compressed suffix array (CSA) taking $nH_0(T) + O(n\log\log\Sigma)$ bits space, however its I/O bound $O(|P|\log_B n + occ \log n)$ is too expensive [37]. There were several attempts to design I/O-efficient compressed indexes based on popular Lempel-Ziv compression. In [6], Arroyuelo and Navarro proposed an index whose space is $8nH_k + o(n\log|\Sigma|)$ bits, but the I/O bounds for pattern searching were not given. Their work is practical in nature and claims to support pattern matching queries in about 20–60 disk accesses. In [18], González and Navarro provided an index which takes $O(|P| + occ/B)$ I/Os for answering pattern matching query. However, their space usage is $O((n\log n) \times H_k \log(1/H_k))$ bits, which is an $\Omega(\log n)$ factor away from the optimal space complexity. The most famous external memory text index "String B-tree" [13], which combines B-tree with Patricia tries, takes $O(n)$ words or $O(n/B)$ disk blocks space and performs pattern matching queries in optimal $O(|P|/B + \log_B n + occ/B)$ I/O's.

One of the main technique we use in this paper is suffix sampling and thus maintain a trie of only those selected suffixes called sparse suffix tree. This technique have been used as a simple tool for designing compressed indexes for many applications such as, fully compressed suffix tree [41], aligned pattern matching [45], text indexing with wild card [27], dictionary matching [23, 24, 28] etc. Recently Kolpakov et al. [33] proposed an internal memory succinct index using sparse suffix trees. Kärkkäinen and Ukkonen [32] attempted to obtain a compressed text index by using sparse suffix array, which indexes only a subset of the suffixes of the text rather than all suffixes. They also used range query data structures, but could only achieve exponential query complexity in the worst case.

Our `Points2Text` transform has been used to prove some lower bounds results such as, Chan [11] et al. showed that we cannot have a succinct index for position restricted sub-string searching with $O(|P|\log^{O(1)} n + |output|)$ query time. Similar results were proved for aligned pattern matching, sub-string range reporting, two pattern matching etc in pointer machine [16, 17, 45].

Organization of the Paper The organization of the paper is as follows. Section 2 summarizes the existing related results that will be applied later. Section 3 defines the two transformations GBWT and `Points2Text`. In Sect. 4, we introduce our GBWT-based index for the internal memory model, which forms the framework for our external memory index in Sect. 5. In Sect. 6, we present our lower bound results. We conclude the paper in Sect. 7 with some open problems.

2 Preliminaries

2.1 Suffix Trees, Suffix Arrays, and Burrows-Wheeler Transform

Suffix trees [39, 46] and suffix arrays [38] are two well-known and popular text indexes that support online pattern matching queries in optimal (or nearly optimal) time. For text $T[1..n]$ to be indexed, each substring $T[i..n]$, with $i \in [1, n]$, is called a *suffix* of T. The suffix tree for T is a lexicographic arrangement of all these n suffixes in a compact trie structure, where the ith leftmost leaf represents the ith lexicographically smallest suffix. Each edge e in the suffix tree is labelled by a series of characters, such that if we examine each root-to-leaf path, the concatenation of the edge labels along the path is exactly equal to the corresponding suffix represented by the leaf.

Suffix array $SA[1..n]$ is an array of length n, where $SA[i]$ is the starting position (in T) of the ith lexicographically smallest suffix of T. An important property of SA is that the starting positions of all suffixes with the same prefix are always stored in a contiguous region in SA. Based on this property, we define the *suffix range* of a pattern P in SA to be the maximal range $[\ell, r]$ such that for all $j \in [\ell, r]$, $SA[j]$ is the starting point of a suffix of T with P as a prefix. Note that SA can be obtained by traversing the leaves of suffix tree in a left-to-right order, and outputting the starting position of each leaf (i.e., a suffix of T) along this traversal. In particular, we have the following technical lemma about suffix trees, suffix arrays, and suffix ranges.

Lemma 1 *Given a text T of length n, we can index T using suffix tree and suffix array in $\Theta(n \log n)$ bits such that the suffix range of any input pattern P can be obtained in $O(\min\{|P|, |P|/\log_{|\Sigma|} n + \log\log n\})$ time.*

Proof The normal suffix tree supports pattern matching in $O(|P|)$ time. Alternatively, we may slightly modify the definition of the suffix tree and perform searching in $O(|P|/\log_{|\Sigma|} n + \log\log n)$ time as follows: Each suffix $T[i..n]$ is converted into $T'[i..n]$ by blocking every $0.5 \log_{|\Sigma|} n$ consecutive characters as a single meta-character; the modified suffix tree is defined as a compact trie of all $T'[i..n]$'s. Here each meta-character is of length $0.5 \log n$ bits, hence the number of such distinct meta-characters is $|\Sigma|^{0.5 \log_{|\Sigma|} n} = \sqrt{n}$. Note that each edge in this modified suffix tree contains an integral number of meta-characters, and each meta-character can be processed in $O(1)$ time in word RAM model. Therefore P can be matched in $O(|P|/\log_{|\Sigma|} n)$ time until the last node in the tree is matched. Unlike the normal suffix tree, we may need to perform a partial match (less than $0.5 \log_{|\Sigma|} n$ original characters) starting from the last node, so as to find out all branches with prefix equal to the partial match. This step can be done by binary search, and can be sped up to $O(\log\log n)$ time using a y-fast trie [47]. This proves the above lemma. □

Suffix trees or suffix arrays maintain relevant information of all n suffixes of T such that on given any input pattern P, we can easily search for the occurrences of P *simultaneously* in each position of T. However, a major drawback is the blow-up in space requirement, from the original $\Theta(n \log |\Sigma|)$ bits of storing the text in plain form to the $\Theta(n \log n)$ bits of maintaining the indexes.

The Burrows-Wheeler transform of a text T is an array BWT of characters such that $BWT[i]$ is the character preceding the ith lexicographically smallest suffix of T. That is, $BWT[i] = T[SA[i] - 1]$.

2.2 External-Memory Model

The external-memory model [2] or I/O model was introduced by Aggarwal and Vitter in 1988. In this model, the CPU is connected directly to an internal memory of size M, which is then connected to a much larger and slower disk. The disk is partitioned into disk pages of B words (i.e., $B \log n$ bits). The CPU can only operate on data inside the internal memory. So, we need to transfer data between internal memory and disk through I/O operations, where each I/O may transfer a disk page from the disk to the memory (or vice versa). Since internal memory (RAM) is much faster, operations on data inside this memory are considered free. Performance of an algorithm in the external-memory model is measured by the number of I/O operations used.

2.3 String B-Tree

String B-tree (SBT) [13] is an index for a text T that supports efficient online pattern matching queries in the external-memory setting. Basically, it is a B-tree over the suffix array SA of T but with extra information stored in each B-tree node to facilitate the matching. The performance of SBT is summarized as follows.

Lemma 2 *Given a text T of length n characters, we can index T using a string B-tree in $\Theta(n/B)$ pages or $\Theta(n \log n)$ bits such that the suffix range of any input pattern P can be obtained in $O(|P|/(B \log_{|\Sigma|} n) + \log_B n)$ I/Os.*

In our compressed text index for the external-memory setting, we again achieve space reduction by maintaining fewer suffixes. Thus, our index includes a sparsified version of the SBT as the main component.

2.4 Orthogonal Range Searching in 2D Grid

In our compressed text index, in addition to the suffix trees or SBT, another key component is a data structure to represent some integer array $A[1..n]$, with each integer represented in $O(\log n)$ bits, and can efficiently support online 4-sided queries of the following form:

Input:	A position range $[\ell, r]$ and a value bound $[y, y']$
Output:	All those z's in $[\ell, r]$ such that $y \le A[z] \le y'$

The above problem can easily be modeled as a geometric problem as follows. First, for each $i \in [1, n]$, generate a point $(i, A[i])$ in the 2-dimensional grid $[1, n] \times [1, n]$. This gives an alternative representation of the array A. Then, for any input query with position range $[\ell, r]$ and value bound $[y, y']$, the desired output corresponds to all points in the grid that are lying inside the rectangle $[\ell, r] \times [y, y']$.

Such a query is called an *orthogonal range query* in the literature [5], and many indexing schemes are devised that have different trade-offs between the index space and the query time and we use the results summarized in the following lemma.

Lemma 3 *Given an integer array A of length n with values drawn from $[1, n]$, we can index A in $O(n \log n)$ bits such that the four-sided query of any position range $[\ell, r]$ and any value bound $[y, y']$ can be answered in $O((|output| + 1) \log n / \log \log n)$ time in the RAM model and $O((|output| + 1) \log_B n)$ I/Os in the external-memory model. If only counting query is required (that is, only the value $|output|$ is needed), it can be answered in $O(\log n)$ time in RAM or $O(\log_B n)$ I/Os in the external-memory model.*

Proof For the RAM model, the desired counting query bound is achieved by [36], while the desired four-sided query bound is achieved by [48]. For the external-memory model, we generalize the binary wavelet tree proposed in [19] into a degree-B wavelet tree. Precisely, we maintain a complete degree-B wavelet tree WT with $h = \log_B n$ levels,[3] so that for each internal node, the pointers to its B children are stored compactly in one disk page (called the *directory* page). Each node of the wavelet tree WT is augmented with an array of $\log B$-bit items. The root r of WT corresponds to the whole array $A[1..n]$, which stores an array $A_r[1..n]$ where $A_r[i]$ is the first $\log B$ bits of $A[i]$. Next, suppose $A[i_1], A[i_2], \dots, A[i_j]$ is the subsequence of $A[1..n]$ where the first $\log B$ bits of each entry is equal to the binary representation of the integer x. Then, we define a new array $A'[1..j]$ such that $A'[k]$ is equal to $A[k]$ with its first $\log B$ bits removed (i.e., each entry of A' will be represented by only $(h - 1) \log B$ bits). The xth child of the root r is defined recursively as a wavelet tree with $h - 1$ levels that corresponds to the array $A'[1..j]$.

Each integer in the original array A is thus partitioned into h parts, one appearing in each level of the wavelet tree. As in the binary wavelet tree proposed in [19], we shall provide a way to link a part of an integer to its other part in the next level. First, the array in each node, say $Z[1..k]$, is stored in contiguous disk pages. For a disk page storing $Z[k'..k'']$, we associate it with a disk page (called the *counter* page) storing B values $C[0..B - 1]$, such that $C[x]$ counts the number of entries with value x in $Z[1..k' - 1]$. Now, for an entry $Z[\ell]$ (which is a part of a certain integer), $\ell \in [k', k'']$, its other part in the next level will be stored in the $Z[\ell]$th child of the current node, as the $C[Z[\ell]] + \gamma$th entry in its stored array, where γ is the number of entries in $Z[k'..\ell]$ whose value is equal to $Z[\ell]$. The value $C[Z[\ell]]$, and the location of the desired child, can be returned in $O(1)$ I/Os by checking the directory and the counter pages. The value γ can be computed based on the content of $Z[k'..k'']$, thus requiring $O(1)$ I/Os as well. In general, linking of a part of an integer to its other part in the wavelet tree requires $O(1)$ I/Os per level. This consequently allows us to answer the four-sided range query and the counting query, analogously to that by using a binary wavelet tree, within the stated bounds.

[3]For simplicity, we assume that n is a power of B, so that $\log_B n$ is an integer. Otherwise, we simply consider the range of values in A as $[1, n']$, where $n' = B^{\lceil \log_B n \rceil}$, so that both the space and query bounds in our proposed scheme follow.

For the space, all directories in the wavelet tree, and all arrays within the wavelet tree nodes, can be stored in a total of $O(n \log n)$ bits. Nevertheless, there is a problem with the current wavelet tree design, concerning the space of the counter pages. Precisely, we are using B counters, even though its associated disk page may contain very few entries. To avoid the problem, we assume that the recursion of a wavelet tree stops as soon as the number of entries is too few (fewer than $B \log n / \log B$), so that the corresponding array can be naively stored in $O(\log_B n)$ disk pages, and the query bounds still hold. Also, there is no need to associate a counter page for the *first* of the contiguous disk pages that store the array. Thus, the number of counter pages is no more than the number of disk pages for storing all arrays, so that the space of the counter pages is also bounded by $O(n \log n)$ bits. □

3 The Two Transformations

This section describes two transformations GBWT and `Points2Text` which are fundamental in deriving all results in this paper.

Definition of GBWT Given a text T with characters drawn from an alphabet $\{1, 2, \ldots, |\Sigma|\}$ and a blocking factor d, GBWT(T, d) is a set S of n/d points (x_i, y_i).[4] Let $T'[1..n/d]$ be the text formed by blocking every d characters of T to form a single meta-character. Thus, the suffix of T' at starting position i corresponds to the suffix of T starting at position $(i-1)d + 1$. Let $SA'[1..n/d]$ be the suffix array of T'. For each character c appearing in T', its binary representation, denoted by bin(c), has $d \log |\Sigma|$ bits. Let $\overleftarrow{c}$ be called as the *reverse character* of c, such that bin$(\overleftarrow{c})$ is the reverse bit-string of bin(c), and is of length $d \log |\Sigma|$ bits.

The GBWT(T, d) is simply the set of n/d points $S = \{(\overleftarrow{T'[SA'[i]-1]}, i) \mid 1 \le i \le n/d\}$. Note that when the points in S are sorted (in increasing order) in the y-coordinates, the corresponding x-coordinates will be similar to the BWT of T', except that each character is replaced by its reverse character.

The GBWT of T can be constructed in the same time as the BWT of T'. Given GBWT, T can be recovered easily in $O(n)$ time. Also, GBWT is space-preserving within a constant factor. The GBWT can also be high-order entropy compressed to achieve nH_k bit representation using the results of [15, 35].

Definition of `Points2Text` Let $S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ be a set of n points in an $n \times n$ grid, such that the x- or y- coordinates of each point is represented naturally in binary using $h = O(\log n)$ bits. Fix an alphabet $\{0, 1, \#, \star\}$ (i.e., each character is encoded in two bits). Let $\langle x \rangle$ denote the string of h characters formed by translating each bit (0 or 1) in the representation of x into the corresponding character encoded in two bits.

[4]For simplicity, we assume n is a multiple of d. Otherwise, T is first padded with enough special character \$ at the end to make the length a multiple of d.

To represent the n points of S, we construct a text T with alphabet $\{\texttt{0}, \texttt{1}, \#, \star\}$ as follows:

$$T = \langle \overleftarrow{x_1} \rangle \# \langle y_1 \rangle \star \langle \overleftarrow{x_2} \rangle \# \langle y_2 \rangle \star \cdots \star \langle \overleftarrow{x_n} \rangle \# \langle y_n \rangle.$$

This text T forms the `Points2Text` transform of S.

The `Points2Text` of S can be constructed and inverted in $O(n)$ time in RAM. In addition, `Points2Text` is space-preserving within a constant factor.

4 Internal Memory Text Index Using GBWT

First, we shall show an alternative succinct text index in internal memory using GBWT. Then, we shall show a higher order entropy compressed version of this index. The space-time bounds of our succinct index is summarized in the following theorem.

Theorem 1 *We can index a text T in $O(n \log |\Sigma|)$ bits such that finding all occurrences of a pattern P in T can be done in $O(|P| + (\log_{|\Sigma|} n + occ) \log n / \log \log n)$ time.*

Let T be a text and T' be the meta-text formed by blocking every $d = \delta \log_{|\Sigma|} n$ characters of T into a single meta-character, with $\delta = 0.5$.[5] To obtain an $O(n \log |\Sigma|)$-bit text index, we first construct a data structure Δ consisting of the suffix tree and the suffix array of T', so that it occupies only $O((n/d) \log(n/d)) = O(n \log |\Sigma|)$ bits. With Δ, we can already support pattern searching, though in a very restricted form. Precisely, it can only report those occurrences of P in T which occur at positions of the form $id + 1$ (Note that $|P|$ does not need to be a multiple of d here.)

To extend the power of Δ, we obtain the points GBWT(T, d), sort them in the y-coordinates, and get the modified Burrows-Wheeler transform BWT_{mod} of T' by listing the corresponding x-coordinates. That is, $BWT_{\mathrm{mod}}[i] = \overleftarrow{BWT[i]}$. After that, we construct the wavelet tree [19, 25, 36] of BWT_{mod}. As each value of BWT_{mod} is character in T', it is represented in $d \log |\Sigma| = \delta \log n$ bits, so the wavelet tree takes $O((n/d)\delta \log n) = O(n \log |\Sigma|)$ bits.

We now show how to use the wavelet tree of BWT_{mod} to extend the searching power of Δ. Note that we can alternatively use any $\mathcal{RS2D}$ data structure on GBWT(T, d). We describe this in terms of wavelet tree because it is easier to derive higher-order entropy compressed index. In particular, we find all those occurrences of P in T with starting position *inside* a character in T'. That is, those occurring at positions i in T with $i \pmod d = k$, where k may not be 1. We call any such occurrence an *offset-k* occurrence. Here, we require that P is longer than $\pi = d - k + 1$ so

[5] For simplicity, we assume that d is an integer. If not, we can slightly modify the data structures without affecting the overall complexity.

that the offset-k occurrence of P starting inside a character in T' does not end inside the same character.

Let $\hat{P}$ denote the prefix of P of length π (i.e., with $\pi \log |\Sigma|$ bits) and $\tilde{P}$ denote the suffix of P formed by taking $\hat{P}$ away from P. We define two characters $c_{\min}$ and $c_{\max}$ as follows: Reverse the bit-string of $\hat{P}$ and then append $(d - \pi) \log |\Sigma|$ bits of 0's to it, we obtain the $d \log |\Sigma|$ bit-string of the character $c_{\min}$; if we append $(d - \pi) \log |\Sigma|$ bits of 1's instead, we obtain the $d \log |\Sigma|$ bit-string of the character $c_{\max}$.

To find all offset-k occurrences of P in T, it is sufficient to find all positions i' in T' such that $\tilde{P}$ occurs at i' in T' (precisely, $\tilde{P}$ occurs at $(i' - 1)d + 1$ in T), with the binary encoding of $\hat{P}$ matching the suffix of the binary encoding of $T'[i' - 1]$. The latter happens if and only if $c_{\min} \le \overleftarrow{T'[i' - 1]} \le c_{\max}$. Based on this, the set of i's can be found as follows:

1. Search $\tilde{P}$ in Δ to obtain the SA range $[\ell, r]$ of $\tilde{P}$ in T'. That is, $SA'[\ell..r]$ contain all occurrences of $\tilde{P}$ in T'.
2. Construct $c_{\min}$ and $c_{\max}$ based on $\hat{P}$.
3. Search wavelet tree of BWT_{mod} to find all y's in $[\ell, r]$ such that $c_{\min} \le BWT_{\text{mod}}[y] \le c_{\max}$.
4. Find $SA'[y]$ for all the y's in Step 3 (using the suffix array Δ built on T'), which are the offset-k occurrences of P.

We apply the above step to find offset-k occurrences of P for $k = 2, 3, \ldots, d$. This gives the following:

Lemma 4 *Based on Δ and the wavelet tree of BWT_{mod}, all occurrences of P with starting and ending positions inside different characters in T' can be found in $O(|P| + (\log_{|\Sigma|} n + occ) \log n / \log \log n)$ time.*

Proof The total time to perform searching in Δ is $O(|P|/d + \log \log n) \times d = O(|P| + d \log \log n)$. The total time to report the occurrences is $O(\sum_{k=1}^{d} (occ_k + 1) \log n / \log \log n)$, where occ_k denotes the number of offset-k occurrences. The previous sum is equal to $O((d + occ) \log n / \log \log n)$, and lemma follows by combining this two bounds. □

In next lemma we show how to find those occurrences of P that start and end in the *same* character of T'.

Lemma 5 *Using four-russians technique, all occurrences of P with starting and ending positions within the same characters in T' can be found in $O(|P| + occ)$ time using an $O(n \log |\Sigma|)$ bits data structure.*

Proof Since $d = 0.5 \log_{|\Sigma|} n$, the number of distinct meta-characters is bounded by $|\Sigma|^d = \sqrt{n}$. We maintain a generalized suffix tree of all meta-characters, using index space of $O(\sqrt{n} \times d \log n) = o(n)$ bits. Let M_i be a meta-character and L_i be the list of all positions in T' where M_i appears. We define a meta-character list L which is formed by the concatenation of L_i's corresponding to all distinct M_i's. Hence the space needed for L is $O(|T'|)$ words or $(|T'|\log n)$ bits. For each leaf v representing a string s in the generalized suffix tree, v contains pointers to the distinct lists L_i's (may be more than 1) whose meta-characters have s as a suffix. As each meta-characters have d suffixes, the total number of pointers is bounded by $O(\sqrt{n} \times d)$ so that their total space is again bounded by $o(n)$ bits. When searching a pattern P, we can use $O(|P|)$ time to locate all the leaves where P occurs as a prefix, and then by chasing the pointers from the leaves to their corresponding L_i's, we obtain all the occurrences. In conclusion, the desired occurrences of P can be found in $O(|P| + occ)$ time, and the space of the overall data structure is $O(n \log|\Sigma|)$ bits. □

By combining Lemmas 4 and 5, we obtain Theorem 1.

4.1 Higher-Order Entropy Compressed Index

We prove the following in this section.

Theorem 2 *A text T can be indexed in $O(nH_k) + o(n\log|\Sigma|)$ bits space, such that all the occurrences of a pattern P in T can be reported in $O(|P|\log^{\epsilon} n + \log^{2+\epsilon} n/\log|\Sigma| + occ \log n/\log\log n)$ time, where H_k represents the kth order empirical entropy of T.*

Proof In order to achieve entropy compressed space, we took a novel approach of variable length blocking of T combined with arithmetic coding scheme. We first transform T into an equivalent text T' such that T' consists of at most $O((nH_k + o(n\log|\Sigma|))/\log n)$ meta-characters, where each meta-character represents at most d consecutive characters in the original text for some threshold d. In addition, we also require that each meta-character can be described in $O(\log n)$ bits, so that T' can be described in $O(nH_k) + o(n\log|\Sigma|)$ bits. Therefore, instead of having each meta-character contain a fixed number of characters, we allow a variable number of characters. Each meta-character is encoded in such a way that, its first k characters are written explicitly (using fixed length encoding) and the rest using kth-order arithmetic coding. The number of characters within a meta-character is restricted by the following two conditions.

- The number of characters should not exceed a threshold $d = \log^{1+\epsilon} n/\log|\Sigma|$.
- After encoding, the total length should not exceed $0.5\log n$ bits.[6]

In our new index, the transformation of T into T' can be performed as follows. Start encoding T from $T[1]$ and get its longest prefix $T[1..j]$, which satisfies the conditions of a meta-character. Hence, $T[1..j]$ in its encoded form is our first meta-character. After that the remainder of T is encoded recursively. (Note that the strings corresponding to distinct meta-characters are not required to be prefix-free.) The

[6]Without loss of generality, we assume here that $|\Sigma| < \sqrt{n}$. The parameters can be appropriately adjusted for the more general case when $|\Sigma| = O(n^{1-\epsilon})$ for any fixed $\epsilon > 0$.

starting position of each meta-character is stored in an array M such that $M[i]$ corresponds to the starting position of ith meta-character in T. In other words, the substring $T[M[i],\dots,(M[i+1]-1)]$ corresponds to the ith meta-character. For instance, $M[1]=1$ and $M[2]=j+1$. By concatenating all these meta-characters (in the order in which the corresponding block appears in T), we obtain the desired string T'.

Since each meta-character corresponds to a maximal substring of T without violating the two conditions, a meta-character corresponds either to (i) exactly d characters of T, or (ii) its encoding is just below $0.5\log n$ in which case the encoding is of $\Theta(\log n)$ bits and corresponds to $\Theta(\log_{|\Sigma|} n)$ characters of T.[7] Note that in both cases each meta-character corresponds to $\Omega(\log_{|\Sigma|} n)$ characters.

Direct entropy compression of T would have resulted in $nH_k + o(n\log|\Sigma|)$-bit space for T'. But in our scheme, the first k characters are written explicitly in each block. This results in an overhead of $O((n/\log_{|\Sigma|} n) \times k\log|\Sigma|) = o(n\log|\Sigma|)$ bits to encode T', assuming $k = o(\log_{|\Sigma|} n)$.[8] Thus, the number of meta-characters from (i) cannot exceed $n/d = o(n\log|\Sigma|/\log n)$, while the number of meta-characters from (ii) is bounded by $O((nH_k + o(n\log|\Sigma|))/\log n)$. In summary, the length of $T' = nH_k + o(n\log|\Sigma|)$ bits, and there is a total of $O((nH_k + o(n\log|\Sigma|))/\log n)$ meta-characters in T'.

By considering each meta-character as a single character from the new alphabet set, we construct the suffix tree Δ of T'. As the length of T' is given by $O((nH_k + o(n\log|\Sigma|))/\log n)$, so is the number of nodes in Δ. Thus, Δ takes $O((nH_k + o(n\log|\Sigma|))/\log n \times \log n) = O(nH_k) + o(n\log|\Sigma|)$ bits of space.

As each meta-character has an encoding between 1 and $0.5\log n$ bits, the number of distinct meta-character is at most $\sum_{r=1}^{0.5\log n} 2^r = O(\sqrt{n})$. Hence the reverse of each meta-character can also be encoded in $O(\log n)$ bits. Therefore the wavelet tree of BWT_{mod} of T' takes $O(|T'|\log n) = O(nH_k) + o(n\log|\Sigma|)$ bits of space. We maintain an auxiliary trie structure of all meta-characters in reverse order with its leaf number (number of reverse meta-characters which are lexicographically smaller) represents the corresponding encoded value. The size of this trie is $O(\sqrt{n}(\log^{1+\epsilon}/\log|\Sigma| + \log n)) = o(n)$ bits. Therefore (encoded) $C_{\min}$ and $C_{\max}$ corresponding to a $\hat{P}$ (prefix of P) can be calculated from this trie structure in $O(d\log|\Sigma|/\log n) = O(\log^{\epsilon} n)$ time and the pattern matching can be performed in the similar way as before. The size of the auxiliary structure for small patterns is $o(n)$ bits and the associated list takes $O(|T'|\log n)$ bits, hence the total size of our compressed index is $O(nH_k) + o(n\log|\Sigma|)$. The pattern matching time is $O((|P|/\log_{|\Sigma|} n + \log n)d + occ\log n/\log\log n)$. □

[7] Here, we make a slight modification that one extra bit is spent for each meta-character, such that if our kth-order encoding of the next $o(\log_{|\Sigma|} n)$ characters already exceeds $0.5\log n$, we shall instead encode the next $0.5\log_{|\Sigma|} n$ characters (i.e., more characters) in its plain form. The extra bit is used to indicate whether we use the plain encoding or the kth-order encoding.

[8] As mentioned, there is also an extra bit overhead per meta-character; however, we will soon see that the number of meta-characters = $O((nH_k + o(n\log|\Sigma|))/\log n)$ so that this overhead is negligible.

5 External Memory Text Index Using GBWT

We first prove the following.

Theorem 3 *We can index a text T in $O(n \log |\Sigma|)$ bits such that finding all occurrences of a pattern P in T can be done in $O(|P|/B + \log_{|\Sigma|} n \log_B n + occ \log_B n)$ I/Os. The space can be further improved to $O(nH_k) + o(n \log |\Sigma|)$ bits, with a query answering I/Os of $O(|P| \log^\epsilon n/B + \log_{|\Sigma|} n \log_B n \log^\epsilon n + occ \log_B n)$.*

Proof In order to obtain the bounds, we simply replaces each data structure used in Sect. 4 by its external memory counter-part. That is, we replace Δ of T' by the string B-tree [13] of T' and the wavelet tree of BWT_{mod} by the external memory wavelet tree of BWT_{mod} (Lemma 3); for the suffix trees inside the data structures that search short patterns, they are replaced by string B-trees as well. Recall that the B is the size of disk page, which is measured in terms of memory words. Here, we further assume that the decoding table for arithmetic coding fits in the internal memory. By choosing appropriate parameters and with the condition that $k = o(\log_{|\Sigma|} n)$, we can ensure that the decoding table size is $O(n^\epsilon)$ bits. The case of short patterns can be handled using a different structure and the result is summarized in Lemma 6. □

Lemma 6 *All the occurrences of P with starting and ending positions within the same meta-character can be found in $O(d + occ/B)$ I/Os.*

Proof Recall the generalized suffix tree (string B-tree) and the list L of all distinct meta-characters. Here we maintain two lists L^+ and L^-, where L^+ is same as L in which we maintain L_i only if M_i occurs at least B times in T'. Let L^i be the list corresponding to the meta-character in which the ith suffix (in generalised suffix tree of meta-characters) belongs to. The list L^- is the concatenation of all those L^i's in which $|L^i| < B$. Note that L^- contain repeated lists (if ith and jth suffix may belongs to the same meta-character, then $L^i = L^j$), but the total space taken by L^- can be bounded as $O(|L^-| \log n) = O(\sqrt{n} \times B \log n) = o(n)$ bits (assume $B = O(n^{1/2-\varepsilon})$). Now for those lists which are guaranteed to give at least B occurrences, we search in L^+ list by spending $O(1)$ I/O per B occurrences. For short lists, we retrieve the lists from L^ℓ to L^r in L^-, where $[\ell, r]$ be the suffix range of P. Here $O(d)$ is the time for searching P ($|P| < d$) in the generalized suffix tree of all distinct meta-characters. □

The additive terms $|P|/B$ (or $|P| \log^\epsilon n/B$) and $occ \log_B n$ in the query I/O are not optimal. In general, one desire factors like $|P|/(B \log_{|\Sigma|} n)$ and occ/B. In this section, first we show how to achieve $|P|/(B \log_{|\Sigma|} n)$ term and later (in section 6) we prove that achieving occ/B term in polynomial I/O's and succinct space is not possible in general. However, this factor can be achieved when patterns are sufficiently large.

Inorder to achieve $|P|/(B \log_{|\Sigma|} n)$ term, we allow slightly more index space. This is done by combining our index with Sadakane's Compressed Suffix Tree (CST) [42]. Our goal is to avoid repeated pattern matching for various offsets, which is done by using the "suffix link" functionality provided by CST. For any internal node u inside the suffix tree, let $path(u)$ denote the string obtained by concatenation of edge labels from root to u. The *suffix link* of u is defined to be the (unique) internal node v such that the removal of the first character of $path(u)$ is exactly the same as $path(v)$. The main idea is that if some part of the pattern is matched during the offset-k search then we avoid re-matching it for offset-$(k+1)$ search and onwards; instead we rely on the suffix link to provide information for the subsequent search. However, suffix link with respect to the original suffix tree may not exist in the sparse suffix tree or the sparse string B-tree (simply because some suffixes are missing). In our algorithm, the full (non-sparse) suffix tree on T must be used and to stay within our space we choose the CST of [42], which provides *all* suffix tree functionalities in compressed space.

5.1 Compressed Suffix Tree

Let us assume we have stored Compressed Suffix Tree CST of the text T. In addition, all the nodes in CST which are also in the sparse suffix tree ST' are marked. For this marking, a bit-vector is maintained in addition to CST. The nodes in CST are considered in pre-order fashion and whenever a marked node is visited we write "1" or else we write "0". Thus, this bit-vector $\mathcal{B}$ stores marking information on the top of CST.

We shall need the following functionalities provided by the recent CSTs of [42] together with our bit-vector $\mathcal{B}$. We choose the $O((1/\varepsilon)n\log|\Sigma|)$ bits and $nH_k + 6n + O(n\log\log n/\log_{|\Sigma|} n)$ bits CSTs for our succinct index and the entropy compressed index respectively.

Suffix link: Given a node u (by its pre-order rank) in CST, return the suffix link node v (by its pre-order rank). This function can be done in $O(1)$ in succinct space and in $O(\log|\Sigma|)$ in entropy compressed space.

Highest marked descendant: Given a node u in CST, its highest marked descendant is defined to be the node v such that v is in the subtree of u, v is marked, and no nodes between u and v is marked. Such a node v (if exists) is unique. This is due to the fact that the least common ancestor of two marked nodes (i.e., the least common ancestor of two sparse suffix tree nodes) is also marked. Note that this functionality is not directly provided by CST of [42] but can easily be implemented in $O(1)$ by storing a rank/select data structure over the bit-vector $\mathcal{B}$ along with the parentheses encoding of CST.

Lowest marked ancestor: Given a node u in CST, report its lowest marked ancestor (if exists). This can be done in $O(1)$ based on $\mathcal{B}$ and its the rank/select data structure.

Leftmost leaf: Given a node u in CST, locate its leftmost (rightmost) leaf node in its subtree. This can be done in $O(1)$.

String-depth: Given a node u, report the length of $path(u)$. This can be done in $O(\log^{\epsilon} n)$ in succinct space and in $O(\log^2 n/\log\log n)$ in entropy compressed space.

Weighted level ancestor: Given a leaf ℓ and string-depth w, report the (unique) node u such that u is the first node on the path from root to ℓ with string-depth $\geq w$. This node u must be a lowest common ancestor between ℓ and some other leaf ℓ', so that we can find u if ℓ' is determined. Such ℓ' can be found by binary searching all leaves to the right of ℓ, and examine the string-depth of lowest common ancestor of ℓ and the leaf. The process can be done in $O(\log^{1+\epsilon} n)$ in succinct space and $O(\log^3 n/\log\log n)$ in entropy compressed space.

5.2 Sparse String B-Tree

Our explanation below shall refer to both the sparse suffix tree Δ_{st} and the sparse string B-tree Δ_{sbt}. However, the sparse suffix tree is never stored and is just for the sake of notation and the identification of nodes. Firstly, the following two functionalities of the sparse string-B tree Δ_{sbt} will be used. The I/O complexity for both functions follows directly from the searching strategy of SBT in the original paper [13].

1. Given a pattern P, let $lcp(P, \Delta_{st})$ be the length of the longest common prefix of P with any suffix stored in Δ_{sbt}: we can use $O(lcp(P, \Delta_{st})/B + \log_B n)$ I/Os to find the node u (by its pre-order ranking in the suffix tree Δ_{st}) such that u is the node with smallest string-depth in Δ_{st} and $lcp(P, \Delta_{st}) = lcp(P, path(u))$.
2. If we are given a node u in Δ_{st} such that the pattern P is guaranteed to match up to some length x on $path(u)$, then the above lcp search can be done in $O((lcp(P, \Delta_{st}) - x)/B + \log_B n)$ I/Os.

5.3 Pattern Matching Algorithm

Now, we are ready to show how we match a pattern P in this combination of sparse string B-tree and CST. First we start with finding offset-0 occurrences, then we find offset-1 occurrences, then offset-2 occurrences and so on. Let P_i denote the pattern P with the first i characters deleted. Thus we have to match $P_0, P_1, P_2, \dots, P_{d-1}$ in the sparse string B-tree. Corresponding to each offset i we find the range $[\ell_i, r_i]$ in the sparse string B-tree.

We start matching the pattern $P = P_0$ in Δ_{sbt}; this allows us to find the node u in Δ_{st}, such that u is the closest node from root such that $lcp(path(u), P) = lcp(P, \Delta_{st})$. If the pattern is matched entirely, then we call this offset a success and output its range. In this case we set $lcp = p$, and also obtain the range $[\ell_0, r_0]$. If not, we set $lcp = lcp(P, \Delta_{st})$ and follow the "suffix link". Let's first define the notion of suffix link in the sparse suffix tree Δ_{st} (or Δ_{sbt}).

Definition 1 Given the pair (u, lcp), let pair (v, lcp') be such that (1) $lcp' = lcp - t$, (2) $path(u)[t + 1..lcp] = path(v)[1..lcp']$ and (3) t is the smallest integer ≥ 1 for which such a node v exists in Δ_{st}. If more than one v exists in Δ_{st}, we set v to be the highest node among them. Then (v, lcp') as is called t-suffix link of (u, lcp).

Now, we show how to compute t-suffix link for pair (u, lcp) in $O(t \log^{1+\epsilon} n)$ I/Os in succinct space and in $O(t \log^3 n/\log\log n)$ I/Os in entropy compressed space. This is done by using the suffix link functionality provided by CST. Note that the CST is used only for implementing t-suffix links, and is residing ON DISK (not in internal memory). Therefore, an operation in CST which takes $O(x)$ time in internal memory is counted as $O(x)$ I/Os.

First, we use the pre-order rank of u to find the corresponding node in CST. Then, inside CST, we can find u's ancestor y such that string-depth of y is just more than lcp. This can be done by the weighted level ancestor query in $O(\log^{1+\epsilon} n)$ I/Os in succinct space and in $O(\log^3 n/\log\log n)$ I/Os in entropy compressed space. The node y represents the location where P stops in the CST if P were matched with the CST instead. To proceed for the next offset, we follow the suffix link from y and reach node w (and increment t by 1). Now, we first find the lowest marked ancestor m of w in $O(1)$ I/Os and check if its string-depth is at least $lcp - t$. If so, we come back to its corresponding node v in Δ_{st} and set $lcp' = lcp - t$. Note that (v, lcp') is the desired t-suffix link of (u, lcp), so that we can proceed with the pattern matching in Δ_{sbt}.[9] Otherwise, if m does not exist or its string-depth is too small, we find in the subtree of w and try the highest marked descendant m' of w in $O(1)$ I/Os. If m' exists, we come back to its corresponding node v' in Δ_{st} and set $lcp' = lcp - t$, while it follows that (v', lcp') is the desired t-suffix link of (u, lcp) so that we can again proceed with the pattern matching in Δ_{sbt}. If there is no such marked descendant m', we follow further the suffix link from w (and increment t), and keep following suffix links until we reach either a node m or m' using the above procedure. In this case, we can be sure that none of the offsets between 1 and $t - 1$ would produce any results. Consequently the corresponding (v, lcp') or (v', lcp') will be the desired t-suffix link and we can directly jump to offset-t match.

Theorem 4 *The t-suffix link for pair (u, lcp) in a sparse suffix tree (or sparse string B-tree) can be calculated in $O(t \log^{1+\epsilon} n)$ time (or I/Os) in succinct space and in $O(t \log^3 n/\log\log n)$ time (or I/Os) in entropy compressed space.*

Thus by chasing t-suffix links we obtain all the ranges $[\ell_i, r_i]$ for all the possible offsets (up to at most d of them).

5.4 Analysis

For matching the pattern P, there are d phases. In each phase, we match some *distinct* part of P and then spend $O(\log^{1+\epsilon} n)$ I/Os (in our succinct index) in CST plus an extra $O(\log_B n)$ I/Os (apart from matching characters of P) in Δ_{sbt}. Thus, in total, we spend $O(d \log^{1+\epsilon} n)$ in addition to the I/O in which the pattern is matched with the actual text inside the Δ_{sbt}. On the other hand, since the characters of P are accessed once and are accessed sequentially, the total I/Os for matching characters of P can be bounded by $O(|P|/(B \log_{|\Sigma|} n) + d \log_B n)$. Overall, this gives us $O(|P|/(B \log_{|\Sigma|} n) + d \log^{1+\epsilon} n + d \log_B n)$ I/Os for finding out all the ranges $[\ell_0, r_0], [\ell_1, r_1], \ldots, [\ell_{d-1}, r_{d-1}]$. Once these ranges are ready, we can use the external memory wavelet tree to find out the actual occurrences (which cross a meta-character boundary) and the short patterns are also handled in the same way as before. In our succinct index $d = 0.5 \log_{|\Sigma|} n$ and the extra space taken by CST and associated bit vector is also $O(n \log |\Sigma|)$ bits. We summarize the result as follows.

[9] Note that when we switch back to a node in Δ_{sbt}, we choose the top-most node in Δ_{sbt} corresponding to the node v.

Theorem 5 *A text T can be indexed in $O(n \log |\Sigma|)$ bits in external memory, such that all occurrences of a pattern P in T can be reported in $O(|P|/(B \log_{|\Sigma|} n) + \log_{|\Sigma|} n \log^{1+\epsilon} n + occ \log_B n)$ I/Os.*

Similar analysis can be performed for the entropy compressed index and the resulting I/O bound is $O(|P|/(B \log_{|\Sigma|} n) + d \log^3 n / \log\log n + d \log_B n + occ \log_B n)$. Since the space of CST is $O(nH_k + n)$ bits which is the bottleneck, we may reduce the blocking factor to be $d = 0.5 \log n$ (thus having the effect of more meta-characters in T' but faster query) without affecting the space. The following theorem captures our new result.

Theorem 6 *A text T can be indexed in $O(nH_k + n) + o(n \log |\Sigma|)$ bits in external memory, such that all occurrences of a pattern P in T can be reported in $O(|P|/(B \log_{|\Sigma|} n) + \log^4 n / \log\log n + occ \log_B n)$ I/Os.*

The term occ/B can be achieved for long patterns

$$\bigl(|P| \geq d = \Theta\bigl(\log^2 n/(\log |\Sigma| \log\log_B n)\bigr)\bigr)$$

as follows. First we block the text T using a new blocking factor d.[10] We maintain the string B-tree of the blocked text T'. Instead of using the external memory wavelet tree to index BWT_{mod}, we use the four-sided query index I of Arge et al. [4] (which takes $O((n/B)\log(n/B)\log\log_B n)$ pages space and $O(\log_B n + |output|/B)$ I/O's for two dimensional orthogonal range reporting) to store the points $(i, BWT_{\mathrm{mod}}[i])$. It is easy to check that the index I performs the desired query supported by the wavelet tree of BWT_{mod}. We obtain the following theorem:

Theorem 7 *A given text T can be indexed in $O(n \log |\Sigma|)$ bits in the external memory, such that for a pattern P with $|P| \geq d = \Theta(\log^2 n/(\log |\Sigma| \log\log_B n))$, all its occurrences in T can be reported in $O(|P|/(B \log_{|\Sigma|} n) + d \log^{1+\epsilon} n + occ/B)$ I/Os, where occ is the number of occurrences. In a compressed space of $O(nH_k + n) + o(n \log |\Sigma|)$ bits, this can be performed in $O(|P|/(B \log_{|\Sigma|} n) + d \log^3 n / \log\log n + occ/B)$ I/Os.*

Similarly, if the blocking factor is $d = 0.5 \log_{|\Sigma|} n$, we can use the four-sided query index by Kanth and Singh [31] (which takes $O(n/B)$ pages space and $O(\sqrt{n/B} + |output|/B)$ I/O's for two dimensional orthogonal range reporting) instead of wavelet tree. Combining the results for short patterns in Lemma 6, we have the following theorem.

Theorem 8 *A given text T can be indexed in $O(n \log |\Sigma|)$ bits (or $O(nH_k + n) + o(n \log |\Sigma|)$ bits) in the external memory model, such that, finding all occurrences of a pattern P in T can be done in $O(|P|/(B \log_{|\Sigma|} n) + \sqrt{n/B} \log_{|\Sigma|} n + occ/B)$ I/Os, where occ is the number of occurrences.*

[10] Note that choosing larger d allows more sparsification, but it is not possible to design the four-russians data structure for small patterns in such cases.

6 Lower Bounds Using `Points2Text`

While the internal memory result described above was only an alternative text index described mainly to facilitate the external memory result, the lower bound result is actually a new result even for the internal memory model. We first demonstrate the reduction from 2-dimensional range query with n points in $[1,n] \times [1,n]$ (so that each point is represented in $h = \Theta(\log n)$ bits) to text searching. Based on this, we can obtain the lower bound result for pattern matching with succinct text index.

Theorem 9 *In the pointer machine model, a text index on $T[1..n]$ supporting pattern matching query in $O(|P|\log^{O(1)} n + occ)$ time requires $\Omega(n\log n/\log\log n)$ bits in the worst case. An external memory index on $T[1..n]$ supporting pattern matching query in $O(|P|\log^{O(1)} n + occ/B)$ I/Os, requires $\Omega(n\log n/\log\log_B n)$ bits in the worst case.*

Proof Let $S = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ be a set of n points in $[1,n] \times [1,n]$, and T be the text T in the `Points2Text` of S. On an input ranges $[x_{left}, x_{right}]$ and $[y_{bottom}, y_{top}]$, it is easy to see that finding all points in S that fall inside the ranges can be done by issuing the corresponding $(x_{right} - x_{left}) \times (y_{top} - y_{bottom})$ pattern searching queries in T. In fact, we can limit the number of pattern queries to $O(\log^2 n)$ by the following observation.

Observation 1 For any k and any i, we call the range $[k2^i, (k+1)2^i - 1]$ a *complete range*, which is denoted by $R_{k,i}$. That is, the range contains all 2^i numbers whose quotient, when divided by 2^i, is k. For any range $[\ell, r]$ with $1 \le \ell \le r \le 2^h$, it can be partitioned into at most $2h$ complete ranges, and these ranges can be found in $O(h)$ time.

Now, suppose that $R_{k,i}$ is a complete range in $[x_{left}, x_{right}]$ and $R_{k',i'}$ is a complete range in $[y_{bottom}, y_{top}]$. Then, each point in S with x-coordinate falling in $R_{k,i}$ and y-coordinate falling in $R_{k',i'}$ corresponds to exactly a substring of T in the form of:

$$\text{The last } h-i \text{ characters of } \overleftarrow{\langle k2^i \rangle},\ \text{then } \#,\ \text{then the first } h-i' \text{ characters of } \langle k'2^{i'} \rangle.$$

Thus, we need to issue only $4h^2 = \Theta(\log^2 n)$ pattern queries in T. This gives the following.

Lemma 7 *Given a set S of n points in an $n \times n$ grid, each represented in $h' = O(\log u)$ bits, we can construct two sorted arrays for storing the x-coordinates and y-coordinates, and a text T of length $O(n\log n)$ with characters chosen from an alphabet of size 4; then, a four-sided range query on S can be answered by $O(\log^2 n)$ pattern match queries on T, each query searching a pattern of $O(\log n)$ characters.*

The pointer machine result is obtained by combing the above lemma with the following lower bound by Chazelle [9]: any two dimensional range reporting data structure with an I/O bound of $O(\log^{O(1)} n + |output|)$ must use $\Omega(n \log n / \log\log n)$ words. The external memory result is obtained by combining the above with the lower bound result by Subramanian and Ramaswamy [44] stating, any two dimensional range reporting data structure with an I/O bound of $O(\log^{O(1)} n + |output|/B)$ must use $\Omega(n \log n / \log\log_B n)$ words. □

7 Conclusions and Open Problems

In this paper, we introduced the first I/O-efficient external memory text index, which takes $O(nH_k)$ bits of space, which is asymptotically equal to the space taken by the text in entropy-compressed form. This index is based on our newly introduced transform called GBWT. Furthermore, its reverse transform, called `Points2Text`, enables us to derive our lower bound results. However, many problems still remain open. For example, can we improve the space complexity from $O(nH_k)$ bits to strictly $nH_k + o(n \log |\Sigma|)$ bits without compromising its I/O efficiency? It has been shown that the LZ-based external memory indexes of [6, 18] are space-efficient in practice, though it may be hard to achieve the theoretical I/O bounds as ours. It is interesting to know if we can build a new index so as to take the advantage of both approaches. Also, we remark that although linear-space data structures for $\mathcal{RS2D}$ are not very encouraging *in theory*, many alternatives like R-trees, kd-trees, and Quadtrees [3, 21, 22, 43] are efficient *in practice*, and they are popular among the database community. As $\mathcal{RS2D}$ data structure is a key component of our index, it may be worthwhile to engineer the best range searching structures and incorporate them with our scheme to achieve the best practical performance.

References

1. Agarwal, P.K., Erickson, J.: Geometric range searching and its relatives. Adv. Discret. Comput. Geom. **23**, 1–56 (1999)
2. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM **31**(9), 1116–1127 (1998)
3. Aref, W.G., Ilyas, I.F.: SP-GiST: an extensible database index for supporting space partitioning trees. J. Intell. Inf. Syst. **17**(2–3), 215–240 (2001)
4. Arge, L., Brodal, G.S., Fagerberg, R., Laustsen, M.: Cache-oblivious planar orthogonal range searching and counting. In: Proceedings of Symposium on Computational Geometry, pp. 160–169 (2005)
5. Arge, L., Samoladas, V., Vitter, J.S.: Two-dimensional indexability and optimal range search indexing. In: Proceedings of Symposium on Principles of Database Systems, pp. 346–357 (1999)
6. Arroyuelo, D., Navarro, G.: A Lempel-Ziv text index on secondary storage. In: Proceedings of Symposium on Combinatorial Pattern Matching, pp. 83–94 (2007)
7. Baeza-Yates, R., Barbosa, E.F., Ziviani, N.: Hierarchies of indices for text searching. Inf. Syst. **21**(6), 497–514 (1996)
8. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Technical report 124, Digital Equipment Corporation, Paolo Alto CA, USA (1994)
9. Chazelle, B.: Lower bounds for orthogonal range searching. I: The reporting case. J. ACM **37**, 200–212 (1990)
10. Clark, D., Munro, I.: Efficient suffix trees on secondary storage. In: Proceedings of Symposium on Discrete Algorithms, pp. 383–391 (1996)
11. Chien, Y.F., Hon, W.K., Shah, R., Vitter, J.S.: Geometric Burrows-Wheeler transform: linking range searching and text indexing. In: Proceedings of Data Compression Conference, pp. 252–261 (2008)

12. Chiu, S.Y., Hon, W.K., Shah, R., Vitter, J.S.: I/O-efficient compressed text indexes: from theory to practice. In: Proceedings of Data Compression Conference, pp. 426–434 (2010)
13. Ferragina, P., Grossi, R.: The string B-tree: a new data structure for string searching in external memory and its application. J. ACM **46**(2), 236–280 (1999)
14. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM **52**(4), 552–581 (2005)
15. Ferragina, P., Venturini, R.: A simple storage scheme for strings achieving entropy bounds. In: Proceedings of Symposium on Discrete Algorithms, pp. 690–696 (2007)
16. Fischer, J., Gagie, T., Kopelowitz, T., Lewenstein, M., Mäkinen, V., Salmela, L., Välimäki, N.N.: Forbidden patterns. In: Proceedings of Latin American Theoretical Informatics, pp. 327–337 (2012)
17. Gagie, T., Gawrychowski, P.: Linear-space substring range counting over polylogarithmic alphabets. (2012). CoRR. arXiv:1202.3208 [cs.DS]
18. González, R., Navarro, G.: A compressed text index on secondary memory. In: Proceedings of International Workshop on Combinatorial Algorithms, pp. 80–91 (2007)
19. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proceedings of Symposium on Discrete Algorithms, pp. 841–850 (2003)
20. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. **35**(2), 378–407 (2005)
21. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proceedings of International Conference on Management of Data, pp. 47–57 (1984)
22. Hellerstein, J.M., Naughton, J.F., Pfeffer, A.: Generalized search trees for database systems. In: Proceedings of International Conference on Very Large Data Bases, pp. 562–573 (1995)
23. Hon, W.K., Lam, T.W., Shah, R., Lung, S.L., Vitter, J.S.: Succinct index for dynamic dictionary matching. In: Proceedings of Symposium on Algorithms and Computation, pp. 1034–1043 (2009)
24. Hon, W.K., Lam, T.W., Shah, R., Lung, S.L., Vitter, J.S.: Compressed index for dictionary matching. In: Proceedings of Data Compression Conference, pp. 23–32 (2008)
25. Hon, W.K., Shah, R., Vitter, J.S.: Ordered pattern matching: towards full-text retrieval. Technical report TR-06-008, Purdue University (2006)
26. Hon, W.K., Shah, R., Thankachan, S.V., Vitter, J.S.: On entropy-compressed text indexing in external memory. In: Proceedings of International Symposium on String Processing and Information Retrieval, pp. 75–89 (2009)
27. Hon, W.K., Ku, T.H., Shah, R., Thankachan, S.V., Vitter, J.S.: Compressed text indexing with wildcards. In: Proceedings of International Symposium on String Processing and Information Retrieval, pp. 267–277 (2011)
28. Hon, W.K., Ku, T.H., Shah, R., Thankachan, S.V., Vitter, J.S.: Compressed dictionary matching with one errors. In: Proceedings of Data Compression Conference, pp. 113–122 (2011)
29. Hon, W.K., Shah, R., Vitter, J.S.: Compression, indexing, and retrieval for massive string data. In: Proceedings of Symposium on Combinatorial Pattern Matching, pp. 260–274 (2010)
30. Jacobson, G.: Space-efficient static trees and graphs. In: Proceedings of Symposium on Foundations of Computer Science, pp. 549–554 (1989)
31. Kanth, K.V.R., Singh, A.K.: Optimal dynamic range searching in non-replicating index structures. In: Proceedings of International Conference on Database Theory, pp. 257–276 (1999)
32. Kärkkäinen, J., Ukkonen, E.: Sparse suffix trees. In: Proceedings of International Conference on Computing and Combinatorics, pp. 219–230 (1996)
33. Kolpakov, R., Kucherov, G., Starikovskaya, T.A.: Pattern matching on sparse suffix trees. In: International Conference on Data Compression, Communications and Processing (2011). doi:10.1109/CCP.2011.45
34. Mäkinen, V., Navarro, G.: Compressed full-text indexes. ACM Comput. Surv. **39**(1) (2007)
35. Mäkinen, V., Navarro, G.: Dynamic entropy-compressed sequences and full-text indexes. Technical report TR/DCC-2006-10, University of Chile (2006)
36. Mäkinen, V., Navarro, G.: Position-restricted substring searching. In: Proceedings of Latin American Theoretical Informatics Symposium, pp. 703–714 (2006)
37. Mäkinen, V., Navarro, G., Sadakane, K.: Advantages of backward searching-efficient secondary memory and distributed implementation of compressed suffix arrays. In: Proceedings of Symposium on Algorithms and Computation, pp. 681–692 (2004)
38. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993)
39. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM **23**(2), 262–272 (1976)

40. Munro, J.I.: Tables. In: Proceedings of Conference on Foundations of Software Technology and Theoretical Computer Science, pp. 37–42 (1996)
41. Russo, L.M.S., Navarro, G., Oliveira, A.L.: Fully compressed suffix trees. ACM Trans. Algorithms **7**(4), 53 (2011)
42. Sadakane, K.: Compressed suffix trees with full functionality. Theory Comput. Syst. 589–607(2007)
43. Samet, H.: The quadtree and related hierarchical data structures. ACM Comput. Surv. **16**(2), 187–260 (1984)
44. Subramanian, S., Ramaswamy, S.: The P-range tree: a new data structure for range searching in secondary memory. In: Proceedings of Symposium on Discrete Algorithms, pp. 378–387 (1995)
45. Thankachan, S.V.: Compressed indexes for aligned pattern matching. In: Proceedings of International Symposium on String Processing and Information Retrieval, pp. 410–419 (2011)
46. Weiner, P.: Linear pattern matching algorithms. In: Proceedings of Symposium on Switching and Automata Theory, pp. 1–11 (1973)
47. Willard, D.E.: Log-logarithmic worst-case range queries are possible in space $\theta(N)$. Inf. Process. Lett. **17**(2), 81–84 (1983)
48. Yu, C.C., Hon, W.K., Wang, B.F.: Efficient data structures for orthogonal range successor problem. In: Proceedings of International Computing and Combinatorics Conference, pp. 96–105 (2009)