2006

Compressed Text Indexing and Range Searching

Yu-Feng Chien

Wing-Kai Hon

Rahul Shah

Jeffrey S. Vitter
Kansas University, jsv@ku.edu

Report Number:
06-021

COMPRESSED TEXT INDEXING AND RANGE SEARCHING

Yu-Feng Chien
Waing-Kai Hon
Raul Shah
Jeffrey Scott Vitter

CSD TR #06-021
December 2006

Compressed Text Indexing and Range Searching

Yu-Feng Chien* Wing-Kai Hon* Rahul Shah† Jeffrey Scott Vitter†

Abstract

We introduce two transformations `Text2Points` and `Points2Text` that, respectively, convert text to points in space and vice-versa. With these transformations, data structural problems in pattern matching and geometric range searching can be linked. We show strong connections between space versus query time trade-offs in these fields. Thus, the results in range searching can be applied to compressed indexing and vice versa. In particular, we show that for a given equivalent space, pattern matching queries can be done using 2-D range searching and vice-versa with query times within a factor of $O(\log n)$ of each other. This two-way connection enables us not only to design new data structures for compressed text indexing, but also to derive new lower bounds.

For compressed text indexing, we propose alternative data structures based on our `Text2Points` transform and 4-sided orthogonal query structures in 2-D. Currently, all proposed compressed text indexes are based on the Burrows-Wheeler transform (BWT) or its inverse [16, 17, 20, 22, 42]. We observe that our `Text2Points` transform is related to BWT on blocked text, and hence we also call it *geometric* BWT. With this variant, we solve some well-known open problems in this area of compressed text indexing. In particular, we present the first external memory results for compressed text indexing. We give the first compressed data structures for position-restricted pattern matching [27, 34]. We also show lower bounds for these problems and for the problem of text indexing in general. These are the first known lower bounds (hardness results) in this area.

*Department of Computer Science, National Tsing Hua University, 101 Kuang Fu Road, Sec. 2, Hsinchu, Taiwan 300. Email: {cyf, wkhon}@cs.nthu.edu.tw

†Department of Computer Sciences, Purdue University, West Lafayette, Indiana 47907-2066, USA. Email: {rahul, jsv}@cs.purdue.edu

1 Introduction

Pattern matching and range searching are both very well researched areas in the field of data structures (see [32] and [1, 37] for surveys). Suffix trees and suffix arrays are widely used as pattern matching data structures. Given a text T consisting of n characters from alphabet Σ, the problem is to build an index on T which can answer pattern matching queries efficiently. Suffix trees and suffix arrays can answer these queries but they take $O(n \log n)$ bits of space, which could be a lot more than the optimal $n \log |\Sigma|$ bits required to store the text. A fundamental problem in the area of compressed text indexing is that of designing an $O(n \log |\Sigma|)$-bit data structure to answer pattern matching queries. The input of a pattern matching query Q_{match} is a pattern P, and the query returns the set (or the cardinality of the set) $Q_{match}(T) = \{i \mid T[i..(i + |P| - 1)] = P\}$. An index should support these queries in $O(|P| + \text{polylog}(n) + |Q_{match}(T)|)$ time. We call this data structure problem the $\mathcal{SA}$ (suffix array) problem. In the particular case where the data structure is restricted to use only $O(n \log |\Sigma|)$ bits (that is, linear to the space for storing the original T), we shall call it the $\mathcal{CSA}$ (compressed suffix array) problem. The seminal papers by Grossi and Vitter [22] and Ferragina and Manzini [16] first proposed solutions to $\mathcal{CSA}$ and started the relatively new field of compressed text indexing.

An extension to pattern matching is the problem of position-restricted pattern matching [34]. These queries can be used as building blocks for many other complex text retrieval queries [27]. Here, the text T is given and the input of a query Q_{pr_match} consists of a pattern P along with positions i and j. The query returns the set (or the cardinality of the set) $Q_{pr\_match}(T) = Q_{match}(T) \cap [i, j]$.

The field of orthogonal range searching is comparatively old. Many classical results on data structures and lower bounds exist under various models including pointer machine, RAM, external memory, and cache-oblivious models [1, 4]. Many practical data structures (without worst-case theoretical bounds) also exist for this in the database literature [3, 23, 26, 43]. In this paper, we shall focus on orthogonal range searching with axis-parallel (hyper-)rectangles in dimensions 2 (and 3) efficiently. We are given a set of n points by their x and y coordinates: $S = \{(x_1, y_1), (x_2, y_2), .., (x_n, y_n)\}$. The query Q_{range} specifies a rectangle $(x_\ell, x_r, y_\ell, y_r)$. The answer to query is given by $Q_{range}(S) = \{(x_i, y_i) \in S \mid x_\ell \leq x_i \leq x_r,\ y_\ell \leq y_i \leq y_r\}$. Two specific versions of this query have been considered: counting and reporting. We shall also consider similar queries in dimension 3. We call the problems of designing data structures on S for efficient orthogonal range queries the $\mathcal{RS2D}$ problem for the 2-D case and the $\mathcal{RS3D}$ problem for the 3-D case.

Similar to the Burrows-Wheeler transform (BWT), which transforms a text into another text, we define `Text2Points`, which transforms a text into a set of points. We alternatively call it *geometric* BWT (GBWT) because this transformation (i) maps into points and (ii) is related to BWT on blocked text. Unlike BWT, which needs to use positional placement of text characters, GBWT can maintain position information explicitly within $O(n \log |\Sigma|)$ bits and hence it is more amenable for other models like external memory. We also define `Points2Text`, which transforms a set of points into a text. Both `Text2Points` and `Points2Text` preserve the space up to a constant factor. These transforms allow the data structures for $\mathcal{SA}$ and $\mathcal{RS2D}$ to be used interchangeably for each other. We show that the existence of a data structure for set S in $\mathcal{RS2D}$ taking $O(n)$ words (or, $O(n \log u)$ bits)[††] is equivalent to the existence of a data structure taking $O(n \log |\Sigma|)$ bits for a text T in $\mathcal{SA}$. Then, $\mathcal{SA}$ can be reduced to $\mathcal{RS2D}$: the query $Q_{match}(T)$ can be converted into $O(\log_{|\Sigma|} n)$ queries $Q_{range}(S)$ on the corresponding $\mathcal{RS2D}$ structure. Also, $\mathcal{RS2D}$ can be reduced to $\mathcal{SA}$ as we show that each query $Q_{range}(S)$ can be converted to $O(\log^2 n)$ queries $Q_{match}(T)$ using the corresponding $\mathcal{SA}$ structure. We wish to note that although the mapping is space preserving, it does introduce space blow-ups by small constants and is not a bijection. These are actually two separate mappings.

The field of succinct or compressed data structures has grown in its importance in recent years. The emphasis is to build data structures that use an amount of space no more than the size of the original input data (or, better yet, the size of the compressed representative of the input data) while seamlessly allowing queries as if the data were not compressed. For text indexing, such data structures have been obtained using the Burrows-Wheeler Transform (BWT) or its inverse. When answering pattern matching query in these approaches, one needs to navigate across a function Φ (or its inverse) separately to match each character in the pattern. The structure and permutation generated by this Φ function were mysterious (even after some progress in characterizing the suffix array [24]). Thus, in the external memory model, it was never possible to achieve an additive term like $|P|/B$ in query I/Os, where B is a memory block size (measured in words). In our GBWT, since we explicitly have position information, we do not need to decode and chase Φ function (or its inverse) multiple times and hence the $|P|/B$ additive term can be achieved. We use a sparse suffix array, a range searching data structure, and a

[††]We shall assume machine word size is of $\log u$ bits, with $n \leq u$.

four-russians table to achieve our external memory result.

Although data structures for $\mathcal{RS}2\mathcal{D}$ have been used earlier to solve part of the $\mathcal{SA}$ problem, to the best of our knowledge, we are the first to show the reverse connectivity from $\mathcal{RS}2\mathcal{D}$ to $\mathcal{SA}$. This shows the optimality of many results already known in compressed text indexing. It can also be used to obtain new lower bounds and answer some of the open problems of this area conclusively.

1.1 Related work

Suffix tree [38, 47] and suffix arrays [35] are well known data structures that answer pattern matching queries. However, although the text size is $n \log |\Sigma|$ until very recently no optimal space data structures were known [16, 22]. Various other results have followed like Sadakane's CSA [42] and XBW for labeled trees [15]. Many papers have improved the initial results (see survey by Mäkinen and Navarro [32]). This field has also encouraged the development of some succinct data structures for rank-select dictionaries [41] for bit vectors and multi-character rank-select dictionaries [19, 20]. As some later papers have pointed out, the wavelet tree data structure [20] (for multi-character dictionaries) is an alternative data structure for $\mathcal{RS}2\mathcal{D}$, which achieves k-order compression [33].

Range query structures have been used for many pattern matching queries. They have been especially used for answering 1-error approximate string matching. They have also been used for many other applications including colored range search [28], document retrieval [39], and retrieving pairs of patterns [14]. However, they have rarely been used to get compression. The use of data structures for $\mathcal{RS}2\mathcal{D}$ in [22] also seems to be primarily focusing towards reporting answers rather than achieving compression.

Before the prominence of BWT-based techniques, Kärkkäinen and Ukkonen [31] attempted to achieve compressed data structures by using sparse suffix arrays; that is, the suffix arrays indexed only a subset of the suffixes of the text rather than all the suffixes. They also explored the use of data structures for $\mathcal{RS}2\mathcal{D}$ for this [30]. However, all of these attempts seemed to have missed the four-russians technique for solving the short patterns case. We put this missing piece together to achieve the goal of compressed text indexing and show that navigating across the BWT or Φ-function is not essential to achieve $O(n \log |\Sigma|)$-bit data structure. Thus, we show that the complexity of compressed text indexing is dominated only by the complexity (space-time trade-offs) of $\mathcal{RS}2\mathcal{D}$. In addition, it is recently shown that high-order entropy compression can be achieved just by blocking [18]; this in turn shows that the positional variant of our GBWT can also achieve high-order entropy compressed text indexes.

The problem of compressed text indexing ($\mathcal{CSA}$) in external memory has been a well-known open problem in this area [40]. Also, the question of which other text queries can be answered within the $O(n \log |\Sigma|)$-bit space has not been well explored. The question of developing compressed data structure for the position-restricted queries was left open [27, 34]. No good lower bounds have been known in this field except for the one by Demaine and López-Ortiz [12], which only goes to say that the data structures need to be at least the size of the input text. Our transforms will open up many ideas for deriving lower bounds in geometry to text indexing.

Orthogonal range searching has a long history (see the survey in [1]) in geometry, and many classic results are known. The ones relevant to our interest are mainly in 2-D queries and 3-D queries. For $\mathcal{RS}2\mathcal{D}$, the best data structures in RAM are by Chazelle [9, 10] and by Grossi and Italiano [21]: If linear space (using $(\epsilon^{-1} + O(1))n$ words) is allowed, one can solve count queries in $O(\log n)$ time and reporting queries in $O(\log n + occ \log^{\epsilon} n)$ time (it is not surprising that bounds for CSA look similar), or alternatively, reporting queries in $O(\sqrt{n} + occ)$ time; if $O(n \log^{\epsilon} n)$ words are allowed, one can answer the queries in $O(\log n + occ)$ time. For $\mathcal{RS}3\mathcal{D}$, the best reporting structure in RAM model is by Alstrup, Brodal, and Rauhe [2], which takes $O(n \log^{\epsilon} n)$ words to answer report queries in $O(\log n + k)$ time. The lower bounds on the pointer machine model are shown by Chazelle [11].

The $\mathcal{RS}2\mathcal{D}$ problem is also well researched in the external memory model and the cache-oblivious model. Hellerstein et al [25] showed that query bounds having an additive term occ/B are impossible to achieve using linear-sized structures. For non-replicating structures, the lower bound of $\Omega(\sqrt{n/B} + occ/B)$ was shown by Kanth and Singh [29]. Thus, for linear-sized solutions for $\mathcal{RS}2\mathcal{D}$ (which is needed for $\mathcal{CSA}$), good reporting bounds cannot be achieved, however, counting can be done easily. Constrained by these results, we achieve $O(|P|/B + \log_{|\Sigma|} n \log_B n)$ I/Os for our counting query in the external memory model. Each of the occurrences can be reported for an additional $O(\log_B n)$ I/Os, adding the term $occ \log_B n$ to the total reporting bound. For upper bound results, in the external memory model, there is a data structure taking $O(n(\log(n/B))/(\log \log_B n))$ words which can answer the queries in $O(\log_B n + occ/B)$ I/Os [5]; in the cache-oblivious model, there is a data structure taking $O(n \log^2 n / \log \log n)$ words which can answer the queries in $O(\log_B n + occ/B)$ I/Os [4]. We show a more compact structure in the cache-oblivious model with a somewhat weaker query bound using our transformation.

Although linear-space *theoretical* structures for $\mathcal{RS}2\mathcal{D}$ are not very encouraging, many *practical* structures exist in database community like R-trees, kd-trees, and Quadtrees [3, 23, 26, 43]. This makes our transform very encouraging for practical applications.

1.2 Our contribution and results

We summarize the following contributions:

1. We propose two space-preserving transforms `Text2Points` and `Points2Text`. These transforms are simple, quickly computable, and invertible. The `Text2Points` transform can also be called the *Geometric Burrows-Wheeler Transform* (GBWT).
2. We show that using `Text2Points` (or GBWT) transform one can achieve an $O(n \log |\Sigma|)$-bit text index answering queries in $O(|P| + \log_\Sigma n \log n + occ \log n)$ time. The positional variant of GBWT can be used to achieve high-order entropy compressed text index taking $O(nH_k) + o(n \log |\Sigma|)$ bits of space that can answer *count* pattern matching queries in $O(|P| \log n + \log^2 n)$ time.
3. We develop a data structure taking $O(n \log |\Sigma|)$ bits that can answer position-restricted pattern matching queries in $O(|P| + \text{polylog}(n) + occ)$ time if $P > (\log^{2+\epsilon} n)/(\log |\Sigma|)$.
4. For general patterns, we show that a data structure answering position-restricted queries in $O(\text{polylog}(n) + occ)$ time is $O(n \log_{|\Sigma|})$ bits are unlikely unless RS3D can be imporved.
5. We develop the first external memory data structure taking $O(n \log |\Sigma|)$ bits that can answer the pattern matching queries in $O(|P|/B + (\log_{|\Sigma|} n)(\log_B n) + occ \log_B n)$ I/Os.
6. We show that an external memory data structure answering pattern matching queries in $O((\log_B n)^c + occ/B)$ I/Os, for any c, must occupy $\Omega((n \log |\Sigma|)(\log n / \log \log_B n))$ bits in the worst case.
7. We obtain a cache-oblivious data structure for $\mathcal{RS}2\mathcal{D}$ with $O((\log^2 n)(\log_B n) + occ/B)$ I/Os and taking $O(n \log n)$ words of space.

1.3 Organization of the Paper

The organization of the paper is as follows. Section 2 summarizes existing results that will be applied later. Section 3 defines the two transformations `Text2Points` and `Points2Text`. Sections 4, 5, and 6 present our findings on compressed text indexing and range search. We conclude in Section 7.

2 Preliminaries

In this section, we first describe the well-known suffix tree [38, 47], suffix array [35], and Burrows-Wheeler Transform [8], and introduce some notations to ease later discussions. We then describe the wavelet tree [20, 34] for searching an array of integers. After that, we summarize the performance of two existing external memory data structures, namely the String B-tree (SBT) [13] and a 4-sided range query index [25], as they will be used as the key building blocks in our external memory results. Recall that B is the size of a disk page (measured in words).

2.1 The Suffix Tree, Suffix Array, and Burrows-Wheeler Transform

Let T be a text of n characters, denoted by $T[1..n]$. The substrings of T in the form $T[i..n]$ (for $i = 1, 2, \ldots, n$) are called the *suffixes* of T. The *suffix tree* is an ordered compact trie storing the n suffixes of T, such that the ith leftmost leaf represents the ith smallest suffix in the lexicographical order. The *suffix array* $SA[1..n]$ is an array of n integers, where $SA[i]$ stores the starting position of the ith smallest suffix in the lexicographical order.

If a pattern P appears in T, P is the prefix of some suffix of T, say $T[i..n]$; in this case, we say P occurs at position i. In [35], it is shown that all occurrences of P in T correspond to the positions stored in a contiguous region of SA. That is, one can find ℓ and r (with $\ell \leq r$) such that $SA[\ell], SA[\ell + 1], \ldots, SA[r]$ stores all positions where P occurs. In this paper, we call $[\ell, r]$ the *SA range* of P in T, and we have the following lemma relating the suffix tree, suffix array, and suffix range.

Lemma 1 *Given a text T with $|T| = n$. We can index T using both the suffix tree and the suffix array in $O(n \log n)$ bits such that for any input pattern P, we can find the SA range $[\ell, r]$ of P in T using $O(|P|)$ time. In addition, each SA value can be reported in $O(1)$ time.* □

Based on the SA of T, the *Burrows-Wheeler transform* of T is a text $BWT[1..n]$ such that $BWT[i] = T[SA[i] - 1]$.[‡] In other words, $BWT[i]$ is the character preceding the ith smallest suffix. The BWT is shown to be more compressible than the original text [36], which is used in the bzip2 compression scheme [44] and in the design of compressed text indexes such as the CSA and the FM-index [16, 22].

2.2 Simple Wavelet Tree as a data structure for $\mathcal{RS}2\mathcal{D}$

Given an array $A[1..m]$ of m integers drawn from $[1, n]$, a simple implementation of the wavelet tree [20, 34] supports on an input range $[\ell, r]$ and an input value y, finding all z's in $[\ell, r]$ such that $A[z] = y$ in $O((occ+1) \log n)$ time, where occ denotes the number of z's in the output. Briefly speaking, the wavelet tree is organized as $\log n$ layers of m bits such that the ith layer corresponds to the ith most-significant bit of all values in A. By 'traversing' the wavelet tree from layer 1 to layer i according to the first i bits in binary representation of y, we can maintain all positions z in $[\ell, r]$ such that the first i bits in the binary representation of $A[z]$ matches those of y. It takes constant time to traverse each layer. After traversing $\log n$ layers, we obtain the desired set of z's, and each z can then be retrieved in $O(\log n)$ time.

In fact, the 'traversal' in the wavelet tree can be generalized easily so that we achieve the following results [27].

Lemma 2 *We can index A in $O(m \log n)$ bits such that on an input range $[\ell, r]$ and an input bound $[x, y]$, we can output all $z \in [\ell, r]$ such that $x \leq A[z] \leq y$ in $O((1 + occ) \log n)$ time (or, $O((1 + occ) \log_B n)$ I/Os in the external memory model), where occ denotes the number of such z in the output.* □

2.3 The String B-tree

String B-tree (SBT) [13] is an external-memory index on a given text which, once created, can support subsequent pattern searching queries efficiently. The performance of SBT is as follows.

Lemma 3 *Suppose we have the SBT for T with $|T| = n$. Then for any input pattern P, we can find the SA range $[\ell, r]$ of P in T using $O(|P|/B + \log_B n)$ I/Os. The corresponding SA values can be reported in $O((r - \ell)/B)$ I/Os. The space occupancy of the SBT is $O(n/B)$ pages, or $O(n \log n)$ bits.* □

Remark. The above lemma only summarizes the minimal functionalities of the SBT that are needed in this paper. The actual performance of the String B-tree is much more general, and it can be used to index a dynamic collection of texts as well.

2.4 Four-sided Range Query

Let S be a set of n points in the two-dimensional plane $\mathbb{R}^2$, and each point may be associated with a piece of satellite information. A *four-sided range query* consists of two input ranges (x_1, x_2) and (y_1, y_2), and outputs all points (x, y) (and its satellite information) in S with $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$.

When each point and its satellite information can be represented by $\Theta(\log n)$ bits, there are two existing external-memory data structures for indexing S and support efficient four-sided range queries. The first one is by Arge et al. [5] whose result is summarized below.

Lemma 4 *In the external memory model, we can index a set of n points in $\mathbb{R}^2$ using $O((n/B)(\log(n/B)/(\log \log_B n))$ pages (which is equivalent to, $O(n(\log^2 n)/(\log \log_B n))$ bits) such that each four-sided range query can be answered in $O(\log_B n + k/B)$ I/Os, where k is the number of points in the output.* □

The second one is the O-tree proposed by Kanth and Singh [29] which is a linear-space structure with data points not replicated. The lemma below summarizes its performance.

[‡]In the special case where $SA[i] = 1$, we set $BWT[i] = T[n]$.

Lemma 5 *In the external memory model, we can index a set of n points in $\mathbb{R}^2$ using $O(n/B)$ pages (that is, $O(n \log n)$ bits) such that each four-sided range query can be answered in $O(\sqrt{n/B} + k/B)$ I/Os, where k is the number of points in the output.* □

3 The Two Transformations

This section describes two transformations `Text2Points` and `Points2Text` which are fundamental in deriving all results in this paper.

3.1 The `Text2Points` transform

Given an input text T of length n with characters drawn from an alphabet Σ, we first define the `Text2Points` with a blocking factor d to transform T into a set of n/d points in $\mathbb{N}^3$. Then, we describe how to obtain an $O(n \log |\Sigma|)$-bit index that supports efficient pattern searching query. Our idea is simple: convert T into another text T' by blocking $d = \log_{|\Sigma|} n$ characters, and construct the suffix tree (and the suffix array) of T' instead. When this suffix tree is augmented with a suitable data structure for `Text2Points` of T with blocking factor d, we show that pattern searching query can be answered efficiently. Details are as follows.

3.1.1 Definition of `Text2Points`

Given a text T drawn from $\{1, 2, \ldots, |\Sigma|\}$ and a blocking factor d, `Text2Points`(T, d) is a set S of n/d points (x_i, y_i, z_i) in 3 dimensions.[‡‡] 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 $\text{bin}(c)$, has $d \log |\Sigma|$ bits. Let $\overleftarrow{c}$ be the character such that $\text{bin}(\overleftarrow{c})$ is the reverse bit-string of $\text{bin}(c)$ of length $d \log |\Sigma|$ bits, and we call $\overleftarrow{c}$ the *reverse character* of c.

The `Text2Points`(T,d) is simply the set of n/d points $S = \left\{(SA'[i], \overleftarrow{T'[SA'[i]-1]}, i) \;\middle|\; 1 \le i \le n/d\right\}$. Note that when the points in S are sorted (in increasing order) in the z-coordinates, the corresponding y-coordinates will be similar to the BWT of T', except that each character is replaced by its reverse character. Thus, we shall alternatively call `Text2Points` as the *geometric* BWT (GBWT).

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.

3.1.2 Constructing compressed text index

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 $\delta = 1/4$.[§] To obtain an $O(n \log |\Sigma|)$-bit text index, we first construct a data structure Δ consisting the suffix tree and 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 `Text2Points`(T,d), sort them in the z-coordinates, and get the modified Burrows-Wheeler transform BWT_{mod} of T' by listing the corresponding y-coordinates. That is, $BWT_{mod}[i] = \overleftarrow{BWT[i]}$. After that, we construct the wavelet tree (Lemma 2) 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 that the wavelet tree occupies $O((n/d)\delta \log n) = O(n \log |\Sigma|)$ 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 $\mathcal{RS}2\mathcal{D}$ data structure on `Text2Points`(T,d). We describe this in term of wavelet tree because it is easy to later derive high-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 with $i \bmod d = k$,

[‡‡]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.

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

where k may not be 1. We call such an occurrence an *offset-k* occurrence. Here, we require that P is longer than $\pi = d - k + 1$ so that the offset-k occurrence of P starting inside a character in T' does not end inside the same character.

Let $\widehat{P}$ 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 $\widehat{P}$ away from P. We define two characters c_{min} and c_{max} in $\{1, 2, \ldots, n\}$ as follows: Reverse the bit-string of $\widehat{P}$ and then append $\log n - \pi \log|\Sigma|$ bits of 0s to it, we obtain the $\log n$ bit-string of the character c_{min}; if we append $\log n - \pi \log|\Sigma|$ bits of 1s instead, we obtain the $\log n$ 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',[‡‡] with the binary encoding of $\widehat{P}$ matching the suffix of the binary encoding of $T'[i'-1]$. The latter happens if and only if $c_{min} \leq \overleftarrow{T'[i'-1]} \leq c_{max}$. 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 $\widehat{P}$.
3. Search wavelet tree of BWT_{mod} to find all z's in $[\ell, r]$ such that $c_{min} \leq BWT_{mod}[z] \leq c_{max}$.
4. Use Δ to find $SA'[z]$ for all the z's in Step 3, 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 6 *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 $O(|P| + (\log n)(\log_{|\Sigma|} n) + occ \log n)$ time.*

Proof: Let occ_k denote the number of offset-k occurrences of P. Our algorithm searches Δ for d times, and each time takes $O(|P|/d)$ time to obtain the SA ranges. In total, this part takes $O(|P|)$ time. The offset-1 occurrences can be reported from Δ directly in $O(occ_1 + 1)$ time. The offset-k occurrences, with $k \neq 1$, are reported when we search the wavelet tree of BWT_{mod} based on $[\ell, r]$ and $[c_{min}, c_{max}]$, which takes $O((occ_k + 1) \log n)$ time. In total, reporting occurrences take $O(occ \log n + (\log n)(\log_{|\Sigma|} n))$ time. Combining the two terms gives the desired result. □

It remains to show how to find those occurrences of P that start and end in the *same* character of T'. We solve this by using an auxiliary data structure. First, for each character c in T', if c appears at least $\sqrt{n}$ times, we say c is a *frequent* character. Otherwise, c is an *infrequent* character.

For each frequent character c, we maintain a list of positions in T' where it appears. Because each occurs at least $\sqrt{n}$ times, there are at most $O(\sqrt{n})$ of them. We now treat each of the *distinct* frequent characters as a string of d original characters, and construct a *generalized* suffix tree on these strings.[¶] Then, on any input pattern P, we can find all distinct frequent characters containing P (and the exact position(s) where P is located inside each frequent character). Also, the corresponding pointers to their lists of positions can be returned. The time required to output all occurrences of P appearing in all frequent characters is $O(|P| + occ)$ time where occ denotes the number of occurrences. The space of this generalized suffix tree is $O(\sqrt{n} \times d \times \log n) = o(n)$ bits.

The number of distinct infrequent characters is at most $2^{\delta \log n} = n^{1/4}$, as each character is represented in $\delta \log n$ bits. In total, the number of infrequent characters (counting repeats) is at most $n^{1/4} \times \sqrt{n}$. We treat each of them as a string of d original characters and construct a generalized suffix tree for these strings. Then, on any input pattern P, we can find all occurrences of P appearing inside all (i.e., possibly *non-distinct*) infrequent characters in $O(|P| + occ)$ time, where occ denotes the number of occurrences. The space of this generalized suffix tree is $O(n^{1/4} \times \sqrt{n} \times d \times \log n) = o(n)$ bits.

Combining this with Lemma 6, we obtain the theorem below.

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 $O(|P| + (\log n)(\log_{|\Sigma|} n) + occ \log n)$ time, where occ is the number of occurrences.* □

The above theorem has demonstrated the usefulness of blocking characters in making a compressed index. Note that our index is worse than the best-known result by Grossi, Gupta, and Vitter [20] which occupies less space and supports faster query.[‖] However, our index is intuitive and requires a much simpler implementation

[‡‡] Precisely, $\tilde{P}$ occurs at $(i'-1)d + 1$ in T.

[¶] Given a set of strings $S_1, S_2, \ldots, S_k$, the generalized suffix tree is a compact trie storing all suffixes of all S_i's. Searching of a pattern is done simultaneously in all S_i's.

[‖] Precisely, the space is $O(nH_k) + o(n \log|\Sigma|)$ bits where $k = o(\log n)$ and H_k is the kth-order empirical entropy of T which is always less than $\log|\Sigma|$), and the query time is $O(|P|/\log_{|\Sigma|} n + \log^2 n + occ \log^\epsilon n)$ for any fixed ϵ.

in practice. Also, the idea of blocking is crucial when we develop an external-memory compressed text index in Section 5.

Remark. In fact, the use of GBWT does not forbid us from getting high-order compression. Here, we sketch a method to obtain a k-order compressed index supporting *count* queries. We use the same index as described above, except that we replace the suffix tree Δ by the wavelet tree of the (normal) BWT. Searching for SA ranges in Step 1 can be done by the backward search algorithm in [16] in $O(|P|/d)$ time. As we do not store Δ, we cannot *report* the occurrences of P in Step 4; nevertheless, we can still support count query in $O(|P| + \log^2 n)$ time. For the space complexity, as the wavelet tree was recently shown to be k-order compressible [33], we get k-order compression as we use two wavelet trees plus $o(n)$ bits for searching short patterns. Alternatively, it has also been shown by [18] that the zero order compression of blocked text (as in GBWT) achieves high-order compression of the original text.

3.2 The `Points2Text` transform

Given a set S of n points in $\mathbb{N}^3$, each represented in h bits, we first define the `Points2Text` to transform S into a text of length $O(nh)$ with characters drawn from an alphabet of size 4, plus an integer array of length n. Then, based on `Points2Text`, we obtain a reduction for range searching in 2-D or in 3-D to a text searching problem. This in turn allows us to obtain lower bound results for text searching problems in Section 4 and obtain upper bound results on external-memory range query in the cache-oblivious model in Section 6. Details are as follows.

3.2.1 Definition of `Points2Text`

Let $S = \{(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_n, y_n, z_n)\}$ be a set of n points in $\mathbb{N}^3$, such that the x-, y-, or z- coordinate of each point is represented naturally in binary using $h = O(\log n)$ bits. Without loss of generality, we assume that $z_1 \leq z_2 \leq \ldots \leq z_n$.

We borrow the notation from Section 3.1.1 for an integer x represented in h bits, $\overleftarrow{x}$ is the integer whose representation is the reverse of the representation of x. 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 (`0` or `1`).

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

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

In addition, we construct an array $Z[1..n]$ with $Z[i] = z_i$. The combination of the text T and the array Z is the `Points2Text` transform of S.

Naturally, for a 2-D set S' of n points in $\mathbb{N}^2$, we may form a 3-D set S such that (x_i, y_i) in S' if and only if $(x_i, y_i, 1)$ in S. Then, the `Points2Text` of S' will be defined simply as the text T in the `Points2Text` 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.

3.2.2 Reducing 2-D range query to pattern searching

Here, we demonstrate the reduction from 2-dimension 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. The general case for reducing range query in $\mathbb{R}^2$, assuming each point is represented in $h' = O(\log u)$ bits, can be handled easily by storing a sorted array of x values and a sorted array of y values, using $O(nh')$-bit extra space and $O(\log n)$ extra query time.

Let $S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ 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 $[k2^i, k(2^i + 1) - 1]$ 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 $1 \leq \ell \leq r \leq 2^h$, 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:

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

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

Theorem 2 *Given a set S of n points in $\mathbb{R}^2$, 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 searching queries on T, with each query searching a pattern of $O(\log n)$ characters.* □

Remark. We can reduce the conversion factor from $O(\log^2 n)$ to $O(\log n)$ by using pattern *range* queries (often supported by text indexes with the same query performance as pattern matching queries) which return the positions of all the suffixes which fall between patterns P_1 and P_2 in lexicographic order.

Corollary 1 *In the pointer machine model,** a text index on $T[1..n]$ supporting pattern matching query in $O(\text{polylog}(n) + occ)$ time requires $\Omega((n \log |\Sigma|)(\log n)/(\log \log n))$ bits in the worst case.*

Proof: Immediately follows from Theorem 2 and the lower bound result by Chazelle in [11]. □

4 Position-Restricted Query

Given a text T and a pattern P, and two positions i and j, the *position-restricted query* finds all occurrences of P in T whose starting positions are between i and j. This section presents the upper bound and lower bound results on indexing T for answering position restricted queries.

4.1 Upper Bound

We use a similar blocking technique as in Lemma 6. Let $T[1..n]$ be a text with characters drawn from $\{1, 2, \ldots, |\Sigma|\}$. Let $d = \log^{2+\epsilon} n / \log |\Sigma|$ for some fixed $\epsilon > 0$ and $T'[1..n/d]$ be a meta-text formed by blocking every d characters in T into a meta-character. Let SA' denote the suffix array of T'. Also, we re-use the notations $\overleftarrow{c}$, $\widehat{P}$, $\tilde{P}$, c_{min}, and c_{max} in Section 3.1 analogously, except with the above new value of d.

We construct a data structure Δ consisting of the suffix tree and suffix array of T'. Also, we obtain the set of points $S = \texttt{Text2Points}(T,d)$ and construct an index I for S such that RS3D can be answered in $O(\log n + k)$ time [2]. The sizes of both data structures are $O(n \log |\Sigma|)$ bits.

As P is longer than d, any offset-k occurrence of P with starting position between i and j in the original text T must have $\tilde{P}$ occurring at some position x in T', $\widehat{P}$ matching the "suffix" of $T'[x-1]$, and x between $i' = \lceil (i + |\widehat{P}|)/d \rceil$ and $j' = \lceil (j + |\widehat{P}|)/d \rceil$. Thus, all offset-$k$ occurrences can be found as follows:

1. Search $\tilde{P}$ in Δ to obtain the SA range $[\ell, r]$ of $\tilde{P}$ in T'.
2. Construct c_{min} and c_{max} based on $\widehat{P}$.
3. Compute i' and j'.
4. Search I to find all points (x, y, z) such that $x \in [i', j']$, $y \in [c_{min}, c_{max}]$, and $z \in [\ell, r]$.
5. The x values of all points obtained in Step 4 correspond to the offset-k occurrences of P. (Precisely, P appears at positions $(x-1)d + k$ in T for all x.)

The position-restricted occurrences of P can thus be obtained by finding all offset-k occurrences of P in the above process, for $k = 1, 2, \ldots, d$. The total time to obtain all SA ranges for d times is $O(|P|)$. The total time to search I for d times is $O(d \log^2 n + occ)$, where occ is the number of position-restricted occurrences of P. Combining the two terms gives the following theorem.

**This model is originally developed by Tarjan [46], such that memory are accessed by following a series of pointers, as opposed to random access in RAM.

Theorem 3 *For a fixed $\epsilon > 0$, we can index T in $O(n \log |\Sigma|)$ bits such that for any input pattern P longer than $(\log^{2+\epsilon} n)/(\log |\Sigma|)$ and input positions i and j, we can support the position-restricted query in $O(|P| + (\log^{4+\epsilon} n)/(\log |\Sigma|) + occ)$ time, where occ is the number of occurrences.* □

4.2 Lower Bound

We use a similar reduction technique as we show Theorem 2. Here, we reduce the 3-dimension range query about n points in $[1,n] \times [1,n] \times [1,n]$ to position-restricted query and obtain the desired lower bound result. The general case for reducing range query in $\mathbb{R}^3$, when each point is represented in $h' = O(\log u)$ bits, can be handled easily with $O(nh')$-bit extra space and $O(\log n)$ extra query time.

Let $S = \{(x_i, y_i, z_i) \mid 1 \leq i \leq z\}$ be a set of n points in $\mathbb{N}^3$. We perform `Points2Text` transform on S to obtain a text T and an array Z, where we assume $z_1 \leq z_2 \leq \ldots z_n$ and $Z[i] = z_i$. Recall that T is in the form

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

Let $e = 2h + 2$ denote the length of the string $\langle \overleftarrow{x_i} \rangle \# \langle y_i \rangle \star$. On input ranges $[x_{left}, x_{right}]$, $[y_{bottom}, y_{top}]$, and $[z_{front}, z_{back}]$, let i denote the minimum k with $Z[k] \geq z_{front}$ and j denote the maximum k with $Z[k] \leq z_{back}$. Then, finding all points in S that fall inside the ranges can be done by searching the substring representing (x, y) in T for all $x \in [x_{left}, x_{right}]$ and $y \in [y_{bottom}, y_{top}]$ with positions restricted by $(i-1)e + 1$ and $(j-1)e + 1$. Again by Observation 1, we can limit the number of position-restricted pattern queries to $O(\log^2 n)$. This gives the following theorem and corollary.

Theorem 4 *Given a set S of n points in $\mathbb{R}^2$, each represented in $h' = O(\log u)$ bits, we can construct a sorted array for each of the x-, y-, and z- coordinates, and a text T of length $O(n \log n)$ with characters chosen from an alphabet of size 4; then, a 3-D orthogonal range query on S can be answered by $O(\log^2 n)$ position-restricted pattern searching queries on T, with each query searching a pattern of $O(\log n)$ characters.* □

Corollary 2 *In the pointer machine model, a text index on $T[1..n]$ supporting position-restricted pattern matching query in $O(\text{polylog}(n) + occ)$ time requires $\Omega((n \log |\Sigma|)(\log n / \log \log n)^2)$ bits in the worst case.*

Proof: Immediately follows from Theorem 4 and the lower bound result by Chazelle in [11]. □

Remark. Assuming the current best result in RAM model of $O(n \log^{1+\epsilon} n)$ words of space for $O(\text{polylog}(n) + occ)$ query time cannot be beaten, the data structure for position-restricted query requires at least $\Omega(n \log^{1+\epsilon} n)$ bits of space. Thus, the goal of $\mathcal{CSA}$ will be unlikely to be achieved.

5 Compressed Text Indexes in External Memory

This section presents the upper bound and lower bound results on compressed indexing of T to answer pattern matching queries, under the external memory model. We denote B as the size of a disk page (measured in words).

5.1 Upper Bound

Our first result simply replaces each data structure used in Section 3.1.2 by its external memory counter-part. That is, we replace Δ of T' by the string B-tree of T' and the wavelet tree of BWT_{mod} by the external memory wavelet tree of BWT_{mod} (or by $\mathcal{RS2D}$ structure on `Text2Points`); for the suffix trees inside the data structures that search short patterns, they are replaced by string B-trees as well. By Lemma 3 and the external memory result in Lemma 2, we obtain the following theorem:

Theorem 5 *In the external memory model, 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, where occ is the number of occurrences.* □

Sometimes, the additive term $occ \log_B n$ in the query I/O may not be desirable. In the following, we attempt to reduce this additive term, ideally, to occ/B, under more constraints.

First, we block the text T using a new blocking factor $d = (\log^2 n)/((\log|\Sigma|)(\log\log_B n))$. 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 Lemma 4 to store the points $(i, BWT_{mod}[i])$. 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 6 *In the external memory model, we can index a text T in $O(n\log|\Sigma|)$ bits such that whenever P is longer than $d = (\log^2 n)/((\log|\Sigma|)(\log\log_B n))$, finding all occurrences of a pattern P in T can be done in $O(|P|/B + d\log_B n + occ/B)$ I/Os, where occ is the number of occurrences.* □

Similarly, if the blocking factor is $d = \log_{|\Sigma|} n$, we can use the four-sided query index I' of Lemma 5 to replace the wavelet tree. This gives the following theorem:

Theorem 7 *In the external memory model, we can index a text T in $O(n\log|\Sigma|)$ bits such that whenever P is longer than $d = \Theta(\log_{|\Sigma|} n)$, finding all occurrences of a pattern P in T can be done in $O(|P|/B+\sqrt{n/B}\log_{|\Sigma|} n+ occ/B)$ I/Os, where occ is the number of occurrences.* □

Remark. Reporting occurrences of patterns of length at most $0.25\log_{|\Sigma|} n$ in the data structure for short patterns actually takes $O(occ/B + occ/\sqrt{n} + 1)$ I/Os. Thus, if we combine this with the data structure for Theorem 7, and with the assumption that $B \leq \sqrt{n}$, Theorem 7 holds for all pattern lengths.

Remark. In our bound $|P|/B$, P is measured in terms of number of characters and B in terms number of words. Similar to the best known RAM bound of $|P|/\log_{|\Sigma|} n$, we can indeed achieve $|P|/(B\log_{|\Sigma|} n)$ as our bounds using Φ-function [22] and LCP structure [42]. The result is mainly technical and not the focus of our paper.

5.2 Lower Bound

Subramanian and Ramaswamy [45] showed that any external memory data structure that can answer 2-D orthogonal range query in $O((\log_B n)^c + occ/B)$ I/Os, for any c, must use at least $\Omega((n\log n)/(\log\log_B n))$ words. Combining this result with Theorem 2, we have the following theorem.

Corollary 3 *An external memory index on $T[1..n]$ supporting pattern matching query in $O((\log_B n)^c + occ/B)$ I/Os, for any c, requires $\Omega((n\log|\Sigma|)(\log n)/(\log\log_B n))$ bits in the worst case.* □

6 Cache-Oblivious Four-Sided Range Query Index

The cache-oblivious model is an external memory model such that the memory size M and the page size B are not known in advance. This section presents a cache-oblivious index on 2-D points for answering four-sided range queries. Here, we apply the cache-oblivious string dictionary developed by Brodal and Fagerberg [7] whose performance is as follows.

Lemma 7 *Let $T[1..n]$ be a text. In the cache-oblivious model, we can index T in $O(n/B)$ disk pages (or $O(n)$ words) such that finding occurrences of a pattern P in T requires $O((|P| + occ)/B + \log_B n)$ I/Os, where occ is the number of occurrences.*

Immediately, we obtain the following theorem by combining the above lemma with Theorem 2.

Theorem 8 *Let S be a set of n points in $\mathbb{R}^2$, with each point represented in $h' = O(\log u)$ bits. In the cache-oblivious model, we can index S in $O(n\log n/B)$ disk pages (or $O(n\log n)$ words) such that any four-sided range query on S can be answered with $O(\log^2 n\log_B n + k/B)$ I/Os, where k is the size of the output.*

Proof: (Omitted.) We maintain the sorted lists of x-coordinates, y-coordinates, and z-coordinates with three cache-oblivious B-trees [6], and transform each point in S to a point $[1,n]\times[1,n]\times[1,n]$ in S' so that each point is represented in $O(\log n)$ bits. We then construct $T =$ `Points2Text` of S' and use Lemma 7 to index T. A four-sided query on S is first translated to an equivalent four-sided query on S' using the B-trees. Then, by Theorem 2 and Lemma 7, we can answer a four-sided range query on S' is $O(\log^2 n\log_B n + k/B)$. The space usage is $O(n/B)$ disk pages for the B-trees, and $O(n\log n/B)$ pages for the index on T. □

Remark. The result above is a direct application of [7]. The result above can be improved to $O(\log n + k/B)$ I/Os by using pattern range queries and suffix links. In many applications, k/B is the dominant factor.

7 Conclusions and Future Work

We have introduced two transforms for converting text into set of points and vice-versa. Our transforms reduce the problems of compressed text indexing to those of range searching and vice-versa. Our first transform can be seen as geometric variant of the Burrows-Wheeler transform. With this, we develop new compressed indexes which while matching the performance of earlier ones based on BWT also solve some of the new problems. Our second transform can be used to derive lower bounds for compressed text indexing. We expect that our transforms will have practical impact. Many practical range searching structures can be used for compressed text indexing. It may be a worthwhile line of research to engineer the best range searching structures for text indexing.

Many other problems remain open. For example, can one develop compressed index for 2-D pattern matching with square patterns in the matrix of text? What are the best space bounds for approximate matching? We believe our transforms will greatly help in understanding of the complexity of these problems.

References

[1] P. K. Agarwal and J. Erickson. Geometric Range Searching and Its Relatives. *Advances in Discrete and Computational Geometry*, 23:1–56, 1999.

[2] S. Alstrup, G. S. Brodal, and T. Rauhe. New Data Structures for Orthogonal Range Searching. In *Proceedings of Symposium on Foundations of Computer Science*, pages 198–207, 2000.

[3] W. G. Aref and I. F. Ilyas. SP-GiST: An Extensible Database Index for Supporting Space Partitioning Trees. *Journal of Intelligent Information Systems*, 17(2–3):215–240, 2001.

[4] L. Arge, G. S. Brodal, R. Fagerberg, and M. Laustsen. Cache-Oblivious Planar Orthogonal Range Searching and Counting. In *Proceedings of Symposium on Computational Geometry*, pages 160–169, 2005.

[5] L. Arge, V. Samoladas, and J. S. Vitter. Two-Dimensional Indexability and Optimal Range Search Indexing. In *Proceedings of Symposium on Principles of Database Systems*, pages 346–357, 1999.

[6] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-Oblivious B-Trees. *SIAM Journal on Computing*, 35(2):341.

[7] G. S. Brodal and R. Fagerberg. Cache-Oblivious String Dictionaries. In *Proceedings of Symposium on Discrete Algorithms*, pages 581–590, 2006.

[8] M. Burrows and D. J. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. Technical Report 124, Digital Equipment Corporation, Paolo Alto, CA, USA, 1994.

[9] B. Chazelle. Filtering Search: A New Approach to Query-Answering. *SIAM Journal on Computing*, 15:703–724, 1986.

[10] B. Chazelle. A Functional Approach to Data Structures and Its Use in Multidimensional Searching. *SIAM Journal on Computing*, 17:427–462, 1988.

[11] B. Chazelle. Lower Bounds for Orthogonal Range Searching, I: The Reporting Case. *Journal of the ACM*, 37:200–212, 1990.

[12] E. D. Demaine and A. López-Ortiz. A Linear Lower Bound on Index Size for Text Retrieval. In *Proceedings of Symposium on Discrete Algorithms*, pages 289–294, 2001.

[13] P. Ferragina and R. Grossi. The String B-tree: A New Data Structure for String Searching in External Memory and Its Application. *Journal of the ACM*, 46(2):236–280, 1999.

[14] P. Ferragina, N. Koudas, S. Muthukrishnan, and D. Srivastava. Two-Dimensional Substring Indexing. volume 66, pages 763–774, 2003.

[15] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring Labeled Trees for Optimal Succinctness, and Beyond. In *Proceedings of Symposium on Foundations of Computer Science*, pages 184–196, 2005.

[16] P. Ferragina and G. Manzini. Indexing Compressed Text. *Journal of the ACM*, 52(4):552–581, 2005.

[17] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An Alphabet-Friendly FM-index. In *Proceedings of International Symposium on String Processing and Information Retrieval*, pages 150–160, 2004.

[18] P. Ferragina and R. Venturini. A Simple Storage Scheme for Strings Achieving Entropy Bounds. To appear in *Proceedings of Symposium on Discrete Algorithms*, 2007.

[19] A. Golynski, J. I. Munro, and S. S. Rao. Rank/Select Operations on Large Alphabets: A Tool for Text Indexing. In *Proceedings of Symposium on Discrete Algorithms*, pages 368–373, 2006.

[20] R. Grossi, A. Gupta, and J. S. Vitter. High-Order Entropy-Compressed Text Indexes. In *Proceedings of Symposium on Discrete Algorithms*, pages 841–850, 2003.

[21] R. Grossi and G. F. Italiano. Efficient Splitting and Merging Algorithms for Order Decomposable Problems. *Information and Computation*, 154(1):1.

[22] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[23] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. pages 47–57, 1984.

[24] M. He, J. I. Munro, and S. S. Rao. A Categorization Theorem on Suffix Arrays with Applications to Space Efficient Text Indexes. In *Proceedings of Symposium on Discrete Algorithms*, pages 23–32, 2005.

[25] J. H. Hellerstein, E. Koustsoupias, and C. H. Papadimitriou. On the Analysis of Indexing Schemes. In *Proceedings of Symposium on Principles of Database Systems*, pages 249–256, 1997.

[26] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *Proceedings of International Conference on Very Large Data Bases*, pages 562–573, 1995.

[27] W.-K. Hon, R. Shah, and J. S. Vitter. Ordered Pattern Matching: Towards Full-Text Retrieval. Technical Report TR-06-008, Purdue University, March 2006.

[28] L. C. K. Hui. Color Set Size Problem with Application to String Matching. In *Proceedings of Symposium on Combinatorial Pattern Matching*, pages 230–243, 1992.

[29] K. V. R. Kanth and A. K. Singh. Optimal Dynamic Range Searching in Non-replicating Index Structures. In *Proceedings of International Conference on Database Theory*, pages 257–276, 1999.

[30] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv Parsing and Sublinear-Size Index Structures for String Matching (Extended Abstract). In *Proceedings of South American Workshop on String Processing*, pages 141–155, 1996.

[31] J. Kärkkäinen and E. Ukkonen. Sparse Suffix Trees. In *Proceedings of International Conference on Computing and Combinatorics*, pages 219–230, 1996.

[32] V. Mäkinen and G. Navarro. Compressed Full-Text Indexes. To appear in *ACM Computing Surveys*, 2006.

[33] V. Mäkinen and G. Navarro. Dynamic Entropy-Compressed Sequences and Full-Text Indexes. Technical Report TR/DCC-2006-10, University of Chile, July 2006.

[34] V. Mäkinen and G. Navarro. Position-Restricted Substring Searching. In *Proceedings of LATIN*, pages 703–714, 2006.

[35] U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[36] G. Manzini. An Analysis of the Burrows-Wheeler Transform. *Journal of the ACM*, 48(3):407–430, 2001.

[37] J. Matousek. Geometric Range Searching. *ACM Computing Surveys*, 26(4):421–461, 1994.

[38] E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[39] S. Muthukrishnan. Efficient Algorithms for Document Retrieval Problems. In *Proceedings of Symposium on Discrete Algorithms*, pages 657–666, 2002.

[40] S. J. Puglisi, W. F. Smyth, and A. Turpin. Inverted Files Versus Suffix Arrays for Locating Patterns in Primary Memory. In *Proceedings of International Symposium on String Processing and Information Retrieval*, pages 122–133, 2006.

[41] R. Raman, V. Raman, and S. S. Rao. Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees and Multisets. In *Proceedings of Symposium on Discrete Algorithms*, pages 233–242, 2002.

[42] K. Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *Journal of Algorithms*, 48(2):294–313, 2003. A preliminary version appears in ISAAC 2000.

[43] H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[44] J. Seward. The `bzip2` and `libbzip2` Official Home Page, 1996. `http://sources.redhat.com/bzip2/`.

[45] S. Subramanian and S. Ramaswamy. The P-range Tree: A New Data Structure for Range Searching in Secondary Memory. In *Proceedings of Symposium on Discrete Algorithms*, pages 378–387, 1995.

[46] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1987.

[47] P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of Symposium on Switching and Automata Theory*, pages 1–11, 1973.