Practical Succinct Text Indexes in External Memory

Hongwei Huo*, Xiaoyang Chen*, Yuhao Zhao*, Xiaojin Zhu*, and Jeffrey Scott Vitter†

*Xidian University
Department of Computer Science
Xi'an, Shaanxi, 710071, China
hwhuo@mail.xidian.edu.cn

†The University of Mississippi
Department of Computer & Information Science
University, MS 38677-1848, USA
jsv@OleMiss.edu

Abstract

Chien et al. [1, 2] introduced the geometric Burrows-Wheeler transform (GBWT) as the first succinct text index for I/O-efficient pattern matching in external memory; it operates by transforming a text T into point set S in the two-dimensional plane. In this paper we introduce a practical succinct external memory text index, called mKD-GBWT. We partition S into σ^2 subregions by partitioning the x-axis into σ intervals using the suffix ranges of characters of T and partitioning the y-axis into σ intervals using characters of T, where σ is the alphabet size of T. In this way, we can represent a point using fewer bits and perform a query in a reduced region so as to improve the space usage and I/Os of GBWT in practice. In addition, we plug a crit-bit tree into each node of string B-trees to represent variable-length strings stored. Experimental results show that mKD-GBWT provides significant improvement in space usage compared with the state-of-the-art indexing techniques. The source code is available online [3].

I. Introduction

The explosive growth of textual big data — including text documents, email or text messages, and genome sequences — imposes great challenges in data processing and information retrieval systems. Full-text indexes provide an important solution for retrieval and search in textual big data. A full-text index is a data structure that stores a text string in preprocessed form so that it can support fast string matching queries. As a popular full-text index, suffix arrays [4] can answer pattern matching queries in optimal or almost optimal time; however for big data applications, the space usage, which is $\Theta(n \log n)$ bits of space, is excessive since it is much bigger than the size $O(n \log \sigma)$ of the original text, where σ is the alphabet size.

The compressed suffix array (CSA) [5] and the FM-index [6] achieve a space usage related to the compressed size (i.e., higher-order entropy) of the original text, and they provide fast random access to any part of the original text. The query time is proportional to the query pattern size plus the product of the output size and a small polylog function of n. However, for massive data, the CSA and the FM-index do not fit into internal memory well, and thus they must be stored in secondary storage such as disk drives or in the cloud. The external memory model introduced by Aggarwal and Vitter [7, 8] uses the number of I/O operations to measure the performance of an external memory algorithm.

Queries using the CSA and the FM-index tend to perform random memory accesses and show poor locality, and thus they can incur large I/O costs. Nevertheless, some fast compressed indexes for secondary memory do exist [1, 2, 9–12]. Mäkinen

2375-0359/18/$31.00 ©2018 IEEE
DOI 10.1109/DCC.2018.00030

et al. [12] proposed an external memory index based upon compressed suffix array taking $nH_0 + O(n \text{ loglog } \sigma)$ bits of space, and supporting pattern search in $O(|P| \log_B n + occ \log n)$ I/Os, where $|P|$ is the query pattern size, B is the block size measured in words, and σ is the alphabet size. Arroyuelo and Navarro [9] proposed an index based on Lempel-Ziv compression that takes $8nH_k + o(n \log \sigma)$ bits and claimed to support pattern matching queries in practice in about several dozen disk accesses. González and Navarro [11] presented an external memory index that requires $O((n \log n)H_k \log(1/H_k))$ bits of space and answers a pattern matching query in $O(|P| + occ/B)$ I/Os.

Ferragina and Grossi [13] introduced an efficient external memory data structure for string search, called string B-tree (SBT), that takes $O(n \log n)$ bits of space and supports pattern matching queries in optimal $O(|P|/B + \log_B n + occ/B)$ I/Os.

Chien et al. [1, 2] introduced the first succinct external memory index called geometric Burrows-Wheeler transform (GBWT) with efficient query in $O(|P|/B + \log_\sigma n \log_B n + occ \log_B n)$ I/Os, and they showed that this index is further improved to achieve the high-order entropy-compressed space of $O(nH_k) + o(n \log \sigma)$ bits. In addition, they also gave implementations [10] of the GBWT for randomly generated texts of up to 4 million characters with alphabet size = 4.

In this paper we introduce a practical succinct external memory text index, called mKD-GBWT. It uses $O(n \log \sigma)$ bits space and finds all occurrences of pattern P in T in $O(|P|/B + \log_\sigma n \log_B n + \sqrt{n/B} + occ/B)$ I/Os in the worst case. Experimental results show that our index provides significantly better space performance than the state-of-the-art indexing techniques for various sizes of texts and alphabet sizes. The source code is available online [3].

II. Preliminaries

A. Text Indexing and Suffix Arrays

The suffix tree [14] and suffix array [4] are popular index structures that provide efficient solutions to many problems involving pattern matching and pattern discovery for string processing. Let T denote a text of size n over an alphabet Σ of size σ. A prefix of T is a substring of the form $T[1..i]$, and a suffix of T is a substring of the form $T[j..n]$, where $1 \leq i,\ j \leq n$. The suffix tree of T is a compact trie that stores the n suffixes of T.

The suffix array $SA[1, n]$ of T is an array of n integers that gives the sorted order of the suffixes of T. $SA[i]$ is the starting position in T of the ith smallest suffix of T in lexicographical order, denoted by $\leq_L$. Given a pattern P, if it appears in the jth position of T, then P must be a prefix of the suffix $T[j..n]$. All the starting positions in T where P occurs create a contiguous range in SA. That is, $SA[l], SA[l+1], \cdots, SA[r]$ stores the start positions where P occurs in T, for some $1 \leq l \leq r \leq n$. We refer to $[l, r]$ as the suffix range of pattern P. Suffix trees and suffix arrays use $\Theta(n \log n)$ bits of space, which can be much larger than the text itself.

The text index problem is to design a data structure for T so that for any query pattern P, we can efficiently report all occurrences of P in T. The compressed (or succinct) text index problem is to design a data structure whose space usage is

i	$SA'[i]$	$T'[SA'[i] \mathinner{..} n/d]$	c_i	c'_i	S
1	1	aak ask rha kas rhk aka skr	–	–	{(1, –),
2	6	aka skr	rhk	khr	(2, khr),
3	2	ask rha kas rhk aka skr	aak	kaa	(3, kaa),
4	4	kas rhk aka skr	rha	ahr	(4, ahr),
5	3	rha kas rhk aka skr	ask	ksa	(5, ksa),
6	5	rhk aka skr	kas	sak	(6, sak),
7	7	skr	aka	aka	(7, aka)}

Figure 1: The GBWT S of $T =$ aak ask rha kas rhk aka skr for $d = 3$.

provably close to the entropy-compressed size of T while still maintaining fast query functionality.

B. The KD-tree

Orthogonal range searching in the two-dimensional setting can be described as follows: Given a set S with N points and a query axis-aligned rectangle R, the problem is to report all points lying within R.

A kd-tree [15] built upon S is a balanced binary tree. Using a kd-tree built upon S of N points, we can answer an orthogonal range query in $O(\sqrt{N} + occ)$ time, where occ is the number of reported points. We can externalize the kd-tree trivially — simply create a leaf node when the number of points drops below B. Lemma 1 gives the external memory kd-tree performance [16].

Lemma 1 ([16]). *A kd-tree on a set S of N points in two-dimensional plane occupies $\Theta(N/B)$ disk blocks or $\Theta(N \log N)$ bits, and answers an orthogonal range query in $O(\sqrt{N/B} + occ/B)$ I/Os, where occ is the number of reported points.*

C. The GBWT Transform

The geometric Burrow-Wheeler transform (GBWT) is a variant of the Burrows-Wheeler transform (BWT) [17] that converts a text T into a collection of points S in the two-dimensional plane. GBWT can support pattern matching on T by using the SBT built upon the sampled suffixes of T and the index structure built upon S. GBWT keeps relative position information between characters explicitly and hence it has better locality than the BWT, more suitable for external memory.

Given a text T of n characters from an alphabet Σ, we group each d contiguous characters of T into a meta-character, except for the last group which may contain less than d characters, then we obtain a new text T' of length n/d. Each meta-character of T' corresponds to d characters of T. Therefore, the suffix starting from the position i of T' corresponds to the suffix starting from position $(i-1)d+1$ of T. We use SA' to denote the suffix array of T' of length n/d.

The GBWT of T is a two-dimensional point set $S = \{(i, c'_i) \mid 1 \leq i \leq n/d\}$ built upon T', where c'_i represents the reverse of the meta-character $c_i = T'[SA'[i] - 1]$ preceeding the ith lexicographically smallest suffix $SA'[i]$ of T'. Figure 1 shows an example of GBWT for string $T =$ aak ask rha kas rhk aka skr for $d = 3$, where T' consists of seven meta-characters, and the GBWT S of T is shown in the last column.

III. mKD-GBWT

In this section, we introduce a new succinct text index in external memory, called mKD-GBWT. The mKD-GBWT index consists mainly of two data structures: a new representation of the SBT of T' in which a crit-bit tree [18] is used to represent each SBT node, denoted by cSBT, and the range search structure (multiple kd-trees) built over S. The first data structure is used to report positions of T, which corresponds to positions of meta-characters of T' where P occurs and the second one is used to report positions in T where P occurs within meta-characters.

A. Pattern Matching

Given a text T of length n over an alphabet Σ of size σ, and an query pattern P, the pattern matching query of P on T answers all the positions where P occurs in T. Assume that $|P| \geq d$, where d is a group size. For a pattern matching query of P in cSBT over T', we can only report positions $(i-1)d+1$ in T, which correspond to position i of some meta-characters where P occurs in T'. As a result, in order to be able to support a full-text pattern search, we need to build the orthogonal range search structure over S to find the positions within some meta-character of T' where P occurs.

Each meta-character $c_i = T'[k](= T[l..r])$ of length d of T' has d suffixes, where $k = SA'[i] - 1$, $l = (k-1)d + 1$ and $r = l + d - 1$. For each $2 \leq j \leq d$, if $c_i[j..d]$ is a prefix of P and the suffix $P[d-j+2..m]$ of P is a prefix of the suffix $T'[k+1..n/d]$ (i.e., $P[d-j+2..m]$ occurs at position $k+1$ of T'), then the jth suffix of the meta-character c_i corresponds to an occurrence of P in T.

The key point is how to find the positions within some meta-character of T' where P occurs. Let $P_{k1} = P[1..d-j+1]$ and $P_{k2} = P[d-j+2..m]$ where $2 \leq j \leq d$. If the jth suffix of meta-character c_i is an occurring position of P in T, it must satisfy the following two conditions:

1. P_{k2} is a prefix of $T'[k+1..n/d]$ for $k = SA'[i] - 1$; and
2. $c_i[j..d] = P_{k1}$ where $c_i = T'[k]$.

For condition 1, we can search for P_{k2} on the cSBT of T' to find the suffix range $[s, e]$ of P_{k2} in T'. That is, for any $s \leq i \leq e$, P_{k2} is a prefix of suffixes $T'[SA'[i]..n/d]$. If the jth suffix $c_i[j..d]$ of meta-character $c_i = T'[SA'[i]-1]$ is alphabetically equal to P_{k1} for $s \leq i \leq e$, i.e., $c_i[j..d] = P_{k1}$, then condition 2 is satisfied. Then we must have $P'_{k1} \cdot C^{j-1}_{min} \leq_L c'_i \leq_L P'_{k1} \cdot C^{j-1}_{max}$ for $2 \leq j \leq d$, where P'_{k1} and c'_i are the respective reverse order of P_{k1} and meta-character c_i, C^{j-1}_{min} is the string of length $j-1$ consisting of repetitions of the minimum character in T, C^{j-1}_{max} is the string of length $j-1$ consisting of repetitions of the maximum character in T, and '$\cdot$' denotes the simple concatenation of two strings.

Notice that c'_i is a y-coordinate of the point having x-coordinate of i ($s \leq i \leq e$) in S. Therefore, for query pattern P, we can build the query rectangle $Q_j = [s, e] \times [P'_{k1} \cdot C^{j-1}_{min}, P'_{k1} \cdot C^{j-1}_{max}]$ for $2 \leq j \leq d$ and perform orthogonal range searching on the index data structure on S to find all the points in the query rectangle Q_j. These points must satisfy both conditions 1 and 2.

If points (i, c'_i) satisfy conditions 1 and 2 for some $s \leq i \leq e$, that is, the jth suffix of the meta-character c_i corresponds to a position where pattern P occurs in T, then we need to query the cSBT of T' to obtain the value of $SA'[i]$ (notice that the leaves on the cSBT of T' store the suffix array $SA'[1..n/d]$ of T'). Then we know $d(SA'[i]-1)+1$ is the position where suffix $T'[SA'[i]..n/d]$ occurs in T. Thus $d(SA'[i]-1)-d+j)$ is the position where P occurs in T.

For the example T in Figure 1 for $d = 3$ and pattern $P = \text{akas}$, let $P_{k1} = \text{ak}$ and $P_{k2} = \text{as}$ when $j = 2$, we know the suffix range of P_{k2} is $[s, e] = [3, 3]$ by querying the cSBT of T' and $SA'[3] = 2$. And then we build query rectangle $Q_2 = [s, e] \times [P'_{k1} \cdot C_{\min}, P'_{k1} \cdot C_{max}] = [3, 3] \times [\text{kaa}, \text{kas}]$ and perform range search to report point $(3, \text{ka})$. Hence $d(SA'[i]-1)+1 = 4$ is the position where P_{k2} occurs in T. Therefore, $d(SA'[i]-1)-d+j) = 2$ for $j = 2$ is one position that P occurs in T.

B. Partitioned Range Search Structures

Let S be a set of n/d points in two-dimensional plane obtained by applying GBWT on T. We partition the x-axis into σ intervals by the suffix range $[s_x, e_x]$ of character x of T, and partition the y-axis into σ intervals by character y of T, and then we obtain σ^2 subregions, where $x, y \in \Sigma$. We build a kd-tree $KDT_{x,y}$ for the points in each subregion individually. Then we obtain σ^2 kd-trees.

For the query rectangle $Q_j = [s, e] \times [P'_{k1} \cdot C^{j-1}_{min}, P'_{k1} \cdot C^{j-1}_{max}]$ of P for $2 \leq j \leq d$, where $[s, e]$ is the suffix range obtained by searching P_{k2} on the cSBT of T', we observe the following property: For the query rectangle Q_j and P, we can always determine a unique subregion $R_{x,y} = [s_x, e_x] \times [y \cdot C^{d-1}_{min}, y \cdot C^{d-1}_{max}]$, where $[s_x, e_x]$ is the suffix range in T' of character $x = P[d-j+2]$, and character $y = P[d-j+1]$ such that Q_j is enclosed in $R_{x,y}$.

By this property, when searching P, we can determine directly its region $R_{x,y}$ by $x = P[d-j+2]$ and $y = P[d-j+1]$, which encloses the query rectangle Q_j of P, and perform an orthogonal range search on the corresponding kd-tree $KDT_{x,y}$ built upon $R_{x,y}$ to report the points in Q_j, which are the positions where P occurs within some meta-characters of T'. In this way, we can avoid a range search on the whole region R over S.

Searching on the reduced subregion $R_{x,y}$ instead of the whole region R may reduce the required I/Os. On the other hand, building a single kd-tree over S, we need $\log(n/d)$ bits and $d \log \sigma$ bits to represent x- and y-coordinates of a point, respectively, since there are n/d points in S while building a kd-tree over $R_{x,y}$, individually, we only need $\log(e_c - s_c)$ bits and $(d-1)\log \sigma$ bits to represent the x- and y-coordinates of a point, respectively, where $[s_c, e_c]$ is the suffix range of character c and $1 \leq s_c, e_c \leq n/d$. Thus using multiple kd-trees may take up less space in practice.

C. Query Algorithms

The mKD-GBWT consists of two parts: a crit-bit tree representation of the SBT of T'(cSBT) and multiple kd-trees built upon σ^2 subregions. So we store two index files in the external memory for the mKD-GBWT. The first index stores an implementation of the sparse string B-tree and the second index stores multiple kd-trees. Given a pattern P, we perform pattern matching queries on the mKD-GBWT index in two steps. First, we search for P on the cSBT of T' to obtain a suffix range of P and collect these SA's values in the suffix range stored in the cSBT in lines 2–4. These values of $SA's$ are the start positions of meta-characters where P occurs in T'. Second, we use the cSBT of T' to find the suffix range of P_{k2} in line 13, and construct query rectangle $[s - beg, e - beg] \times [y_l, y_h]$, where $y_l = P'_{k3} \cdot C^{j-1}_{min}$, $y_h = P'_{k3} \cdot C^{j-1}_{max}$, and P'_{k3} is a reverse of P_{k3} $(= P[1..d-j])$, and locate the corresponding kd-tree built upon $R_{r\times s}$ to find all the points occurred in the rectangle so as to find positions where P occurs within some meta-characters of T'. Algorithm 1 describes how to perform a pattern matching query using the mKD-GBWT.

```
Algorithm 1: $PMQuery(T, P)$
1  $result \leftarrow \varnothing$
2  $[s, e] \leftarrow cSBTSearch(T', P)$
3  for $i \leftarrow s$ to $e$ do
4      $result \leftarrow result \cup \{d(SA'[i] - 1) + 1\}$
5  for $j \leftarrow 2$ to $d$ do
6      $S_1 \leftarrow \varnothing$
7      $P_{k2} \leftarrow P[d - j + 2..m]$
8      $P_{k3} \leftarrow P[1..d - j]$
9      $beg \leftarrow ST[P[d - j + 2]]$
10     for $k \leftarrow 1$ to $j - 1$ do
11         $C_{min}[k] \leftarrow c_{min}$
12         $C_{max}[k] \leftarrow c_{max}$
13     $[s, e] \leftarrow cSBTSearch(T', P_{k2})$
14     if $[s, e] \neq \varnothing$ then
15         $[y_l, y_h] \leftarrow [P'_{k3} \cdot C_{min}, P'_{k3} \cdot C_{max}]$
16         Computing $R_{r\times s}$ such that $\Sigma[r] = P[d - j + 2]$ and $\Sigma[s] = P[d - j + 1]$
17         $S_1 \leftarrow KDTSearch([s - beg, e - beg] \times [y_l, y_h], KDT_{r\times s})$
18     for each $(x, y, SA'[x]) \in S_1$ do
19         $result \leftarrow result \cup \{d(SA'[x] - 1) - d + j\}$
20 return $result$
```

We can improve the efficiency of the mKD-GBWT with some carefully considered modifications to the implementation. For example, we can divide the query rectangles of P into groups so that the query rectangles located in the same region $R_{r\times s}$ are packed to the same group to improve the query I/Os.

We plug a crit-bit tree [18] into each SBT node containing some suffixes of T, denoted as cSBT and store it in a disk block of size B. When searching for P in the cSBT, we need to load the crit-bit tree into memory and perform a pattern matching of P on the crit-bit tree to find the positions where corresponding suffixes are stored in the crit-bit tree. We then load its child nodes according to these positions to proceed the search in the cSBT.

IV. Performance

In this section, we give the performance of mKD-GBWT in Theorem 1:

Theorem 1. *Our index on a text T of length n over an alphabet Σ of size σ requires $O(n \log \sigma)$ bits, and finds all the occurrences of pattern P in T in $O(|P|/B + \log_B n \log_\sigma n + \sqrt{n/B} + occ/B)$ I/Os in the worst case, where B is the disk block size and occ is the number of reported points.*

Proof. Our index consists of two parts: one is the cSBT built upon T' of size n/d and the other is the structure of σ^2 kd-trees, where d is the group size. By [13], the space usage by the first part takes $O((n/d)\log(n/d))$ bits, which is $O(n \log \sigma)$ bits for $d = O(\log_\sigma n)$. Next we consider the space occupied by the σ^2 kd-trees $KDT_{x,y}$ built over the points in each of σ^2 subregions, where $x, y \in \Sigma$. Assume that $KDT_{x,y}$ built upon subregion $R_{x,y}$ contains n_i points such that $\sum_{i=1}^{\sigma^2} n_i = n/d$. By Lemma 1, the total space required by the σ^2 kd-trees $KDT_{x,y}$ is bounded by $\sum_{i=1}^{\sigma^2} O(n_i/B) = O(\sum_{i=1}^{\sigma^2} n_i/B) = O(n/(dB))$ disk blocks, which is $O(n \log \sigma)$ bits since $\sum_{i=1}^{\sigma^2} n_i = n/d$. The total space occupied by the mKD-GBWT is therefore $O(n \log \sigma)$ bits.

Now we consider the I/Os required to perform pattern matching queries in our index. By [13], performing a pattern matching query of P in cSBT of T' requires $O(|P|/(B\log_\sigma(n/d)) + \log_B(n/d))$ I/Os, and thus performing d pattern matching queries on cSBT of Algorithm 1 requires a total of $O(d|P|/(B\log_\sigma(n/d)) + d\log_B(n/d))$ I/Os, which is $O(|P|/B + \log_B n \log_\sigma n)$ I/Os. By Lemma 1, performing an orthogonal range query in the kd-tree $KDT_{x,y}$ with n_i points takes $O(\sqrt{n_i/B} + occ_i/B)$ I/Os. Thus performing $d-1$ orthogonal range queries in Algorithm 1 requires search on at most $d-1$ distinct $KDT_{x,y}$ structures. The number of I/Os required is therefore bounded by

$$\begin{aligned}
\sum_{i=1}^{d-1} O\big(\sqrt{n_i/B} + occ_i/B\big) &= \sum_{i=1}^{d-1} O\big(\sqrt{n_i/B}\big) + \sum_{i=1}^{d-1} O(occ_i/B) \\
&= O\big(\sqrt{n_1/B} + \sqrt{n_2/B} + \cdots + \sqrt{n_{d-1}/B}\big) + O(occ/B) \\
&= O\big(\sqrt{d-1}\sqrt{(n_1 + n_2 + \cdots + n_{d-1})/B}\big) + O(occ/B) \\
&= O\big(\sqrt{(d-1)(n/d)/B}\big) + O(occ/B) < O(\sqrt{n/B}) + O(occ/B).
\end{aligned}$$

The second equality is due to the fact that $\sum_{i=1}^{d-1} occ_i \leq occ$. The first inequality follows from *Cauchy-Schwarz* inequality. The second inequality is due to the fact that $\sum_{i=1}^{\sigma\times\sigma} n_i = n/d$.

By adding the I/Os required by searching in the cSBT and multiple kd-trees, we obtain the query bound $O(|P|/B + \log_B n \log_\sigma n + \sqrt{n/B} + occ/B)$ on the I/Os in the worst case. □

V. Experimental Results and Discussion

A. Environment and Setup

We have conducted experiments on a Dell Tower 7910 workstation that has 24 Intel Xeon(R) CPU E5-2630 v3 @2.4GHZ cores and 192GB of memory, running

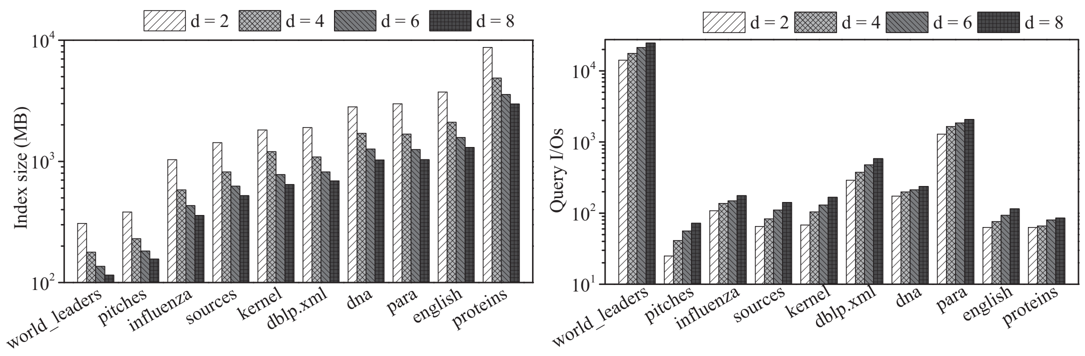

Figure 2: Effect of d on the performance of mKD-GBWT.

Ubuntu 12.04. We implement mKD-GBWT in C++ and compile it using g++ 4.8.1. The disk block size B is set to 4KB. Table 1 summarizes some general characteristics of the tested data sets, which come from the *Pizza & Chilli Corpus* and the *Pizza & Chili Highly Repetitive Corpus*. For each tested data set, we randomly choose 10000 query patterns of length 10 and measure the average number of I/Os incurred for a query.

Table 1: Statistics of the tested data sets

File	size (MB)	σ	File	size (MB)	σ
world_leaders	44.79	89	dblp.xml	282.42	97
pitches	53.25	133	dna	385.22	16
influenza	147.67	15	para	409.38	5
sources	201.1	230	english	500	226
kernel	246.01	160	proteins	1129.2	27

B. Effect of d

As far as we know, the group size d is a key parameter in mKD-GBWT. Thus, in this section we evaluate the effect of d on the performance of mKD-GBWT, and show the results in Figure 2.

By Figure 2, we know that the space of mKD-GBWT decreases as d increases. This is because that the number of sampled suffixes of T becomes smaller as d increases. As a result, the occupied space of cSBT in mKD-GBWT storing these sampled suffixes decreases. Meanwhile, the number of obtained points also decreases, and hence reduces the space occupied by the kd-trees. Note that, as d increases, the decrement of the space of mKD-GBWT is gradually getting smaller. For example, when d varies from 2 to 4, the space decreases by about 50%, while it only decreases by 20% when d increases from 6 to 8.

On the contrary, the number of query I/Os incurred by mKD-GBWT increases as d increases. Several factors may contribute to this trend: (1) as d increases, the number of suffixes P_{k2} increases, which means that we consume more I/Os to search these suffixes by using cSBT; (2) as a result of (1), we obtain more query rectangles. Thus, we may perform more range search on the kd-trees. In addition, the increment of the number of query I/Os becomes smaller as d increases in most case. Considering space and query I/Os together, we choose $d = 6$ as the default parameter for mKD-GBWT.

C. Performance of mKD-GBWT

In this section, we compare our mKD-GBWT with the state-of-the-art external text indexes, including SBT [13] and GBWT [1]. Table 2 shows the index size and query I/Os for different methods. Note that, for GBWT and mKD-GBWT, we set the group size $d = 6$.

Table 2: Index Size (in megabytes) and Query I/Os

Data sets	SBT		GBWT		mKD-GBWT	
	Size	I/Os	Size	I/Os	Size	I/Os
world_leaders	302.7	**13571**	185.8	56223.6	**135.8**	21216.7
pitches	326	**14.5**	196.4	109	**182**	56.1
influenza	941.9	**13.8**	455.8	333.7	**432.9**	150
sources	1384.4	**47.6**	839.8	309.1	**624.2**	110.1
kernel	1789.8	**55.9**	900.1	326.1	**777.8**	130.1
dblp.xml	1693.7	**239.5**	960.5	1271.3	**820.3**	476.7
dna	2802.6	**21.8**	1869.7	771.9	**1179.8**	212.8
para	2767	**1279.3**	1903.1	5329.4	**1252.9**	1845.9
english	3574.2	**15.9**	2028.5	267.2	**1574.1**	93.5
proteins	8348.8	**15.3**	4309.3	132.7	**3562.9**	80.9

As shown in Table 2, SBT consumes the most space as it has stored all suffixes of the text. Compared with GBWT, the index size of our mKD-GBWT decreases by 10%–35%. This improvement is due to the fact that (1) we replace blind tree with crit-bit tree to represent each node of the string B-tree, which can avoid to explicitly store the pointers in the blind tree; (2) we partition the whole region into σ^2 subregions, and each point located in a subregion needs fewer bits in the kd-tree built upon this subregion.

Considering query I/Os, SBT performs better than GBWT and mKD-GBWT. The number of query I/Os incurred by mKD-GBWT is only 27%–60% of that of GBWT, which achieves an excellent performance. This is because that mKD-GBWT perform the range search on subregions, while GBWT performs the one on the whole region. In summary, mKD-GBWT is superior to the alternative GBWT in terms of space and query I/Os, and takes much less space than SBT.

VI. Conclusions

Chien et al. [1] gave an improvement upon their GBWT data structure so as to achieve high-order entropy-compressed space; it used the wavelet tree [19] and a variable length blocking coding scheme, paying an extra factor of polylog function of n in query time.

While our mKD-GBWT index is not optimal in space usage, it performs quickly in practice. We still have a goal of developing a high-order entropy-compressed index while achieving fast query performance both in theory and in practice.

VII. Acknowledgments

The authors would like to thank Wing-Kai Hon and Rahul Shah for their discussions and information regarding GBWT. This work is supported in part by China NSF grants 61173025, 61373044, and 61741215 and U.S. NSF grant CCF-1017623.

References

[1] Y.-F. Chien, W.-K. Hon, R. Shah, S. V. Thankachan, and J. S. Vitter, "Geometric bwt: Compressed text indexing via sparse suffixes and range searching," *Algorithmica*, vol. 71, no. 2, pp. 258–278, 2015.

[2] Y.-F. Chien, W.-K. Hon, R. Shah, and J. S. Vitter, "Geometric burrows-wheeler transform: Linking range searching and text indexing," in *IEEE Data Compression Conference*, 2008, pp. 252–261.

[3] Y. Zhao and H. Huo, "Source code for mKD-GBWT," https://github.com/Hongweihuo-Lab/mKD-GBWT, 2016.

[4] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," *SIAM Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[5] R. Grossi and J. S. Vitter, "Compressed suffix arrays and suffix trees with applications to text indexing and string matching," *SIAM Journal on Computing*, vol. 35, no. 2, pp. 378–407, 2005.

[6] P. Ferragina and G. Manzini, "Indexing compressed text," *Journal of the ACM*, vol. 52, no. 4, pp. 552–581, 2005.

[7] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Communications of the ACM*, vol. 31, no. 9, pp. 1116–1127, 1988.

[8] J. S. Vitter, "Algorithms and data structures for external memory," *Foundations and Trends® in Theoretical Computer Science*, vol. 2, no. 4, pp. 305–474, 2008.

[9] D. Arroyuelo and G. Navarro, "A lempel-ziv text index on secondary storage," in *Proceedings of Symposium on Combinatorial Pattern Matching*, 2007, pp. 83–94.

[10] Y.-F. Chien, W.-K. Hon, R. Shah, and J. S. Vitter, "I/O-efficient compressed text indexes: From theory to practice," in *IEEE Data Compression Conference*, 2010, pp. 426–434.

[11] R. González and G. Navarro, "A compressed text index on secondary memory," *Journal of Combinatorial Mathematics and Combinatorial Computing*, vol. 71, p. 127, 2009.

[12] V. Mäkinen, G. Navarro, and K. Sadakane, "Advantages of backward searching-efficient secondary memory and distributed implementation of compressed suffix arrays," in *Proceedings of Symposium on Algorithms and Computation*, 2004, pp. 681–692.

[13] P. Ferragina and R. Grossi, "The string B-tree: a new data structure for string search in external memory and its applications," *Journal of the ACM*, vol. 46, no. 2, pp. 236–280, 1999.

[14] E. M. McCreight, "A space-economical suffix tree construction algorithm," *Journal of the ACM*, vol. 23, no. 2, pp. 262–272, 1976.

[15] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[16] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter, "Bkd-tree: A dynamic scalable kd-tree," in *International Symposium on Spatial and Temporal Databases*, 2003, pp. 46–65.

[17] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Digital Equipment Corporation, Palo Alto, CA, Tech. Rep., 1994.

[18] S. Gog, A. Moffat, J. S. Culpepper, A. Turpin, and A. Wirth, "Large-scale pattern search using reduced-space on-disk suffix arrays," *IEEE transactions on knowledge and data engineering*, vol. 26, no. 8, pp. 1918–1931, 2014.

[19] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proceedings of the 14th annual ACM-SIAM symposium on Discrete algorithms*, 2003, pp. 841–850.