# I/O-efficient Compressed Text Indexes: From Theory to Practice

Sheng-Yuan Chiu Wing-Kai Hon
*Department of Computer Science*
*National Tsing Hua University*
*Hsinchu, Taiwan*
*Email: {csy,wkhon}@cs.nthu.edu.tw*

Rahul Shah
*Department of Computer Science*
*Louisiana State University*
*Baton Rouge, LA, USA*
*Email: rahul@csc.lsu.edu*

Jeffrey Scott Vitter
*Department of Computer Science*
*Texas A&M University*
*College Station, TX, USA*
*Email: jsv@tamu.edu*

**Abstract**

Pattern matching on text data has been a fundamental field of Computer Science for nearly 40 years. Databases supporting full-text indexing functionality on text data are now widely used by biologists. In the theoretical literature, the most popular internal-memory index structures are the suffix trees and the suffix arrays, and the most popular external-memory index structure is the string B-tree. However, the practical applicability of these indexes has been limited mainly because of their space consumption and I/O issues. These structures use a lot more space (almost 20 to 50 times more) than the original text data and are often disk-resident.

Ferragina and Manzini (2005) and Grossi and Vitter (2005) gave the first compressed text indexes with efficient query times in the internal-memory model. Recently, Chien et al (2008) presented a compact text index in the external memory based on the concept of Geometric Burrows-Wheeler Transform. They also presented lower bounds which suggested that it may be hard to obtain a good index structure in the external memory.

In this paper, we investigate this issue from a practical point of view. On the positive side we show an external-memory text indexing structure (based on R-trees and KD-trees) that saves space by about an order of magnitude as compared to the standard String B-tree. While saving space, these structures also maintain a comparable I/O efficiency to that of String B-tree. We also show various space vs I/O efficiency trade-offs for our structures.

## I. Introduction

Given a text $T$ which is a string of characters drawn from an alphabet set $\Sigma$ and a pattern $P$, how can we locate all the occurrences of the pattern $P$ in $T$? This has been a fundamental problem in Computer Science since almost half a century. Many algorithms are known to solve this problem in $O(n+m)$ time where $n$ is the size of the text and $m$ is the size of the pattern [19]. When the text $T$ is already known and the patterns keep changing, the problem is best seen as a data structure/indexing problem where we can preprocess the text into an index, and the query comes in as a pattern or a set of patterns. In particular, suffix trees [22], [27] and suffix arrays [21] are two indexes widely used for this purpose, and there is still a considerable amount of research interest in database, algorithms, and pattern matching communities on these data structures even in this decade. Nevertheless, two of the main short-comings of these indexes make them insufficient for answering the needs of nowadays applications whose data volumes are huge. The first one is their space requirements, which are often 20 to 60 times the original text, and the second one is their poor locality of reference, which often induces main performance bottleneck when the data resides on disk. Each of these short-comings have been partly addressed by the research community. Firstly, to reduce space requirements, there has been a lot of recent research on compressed text indexing which is based on the Burrows-Wheeler Transform (BWT) of the text [11], [12], [13], [14]. Secondly, to improve the locality, Ferragina and Grossi designed the String B-tree [9] which achieves theoretically optimal I/O performance. However, these two short-comings have so far only been addressed separately but not simultaneously.

Even though storage space has been increasing, the need for compressing data is still very important due to emergence of device-based computing where mobile-devices have limited RAM and the data storage is distributed across some network. Also, the compressed data often fits into faster levels of memory hierarchy, thus automatically reducing I/O performance bottleneck. Recently, Chien et al [8] proposed compact text indexing in external memory which promises to achieve both good I/O performance and good compression factor at one shot. The main idea is by incorporating orthogonal range searching data structures on top of the String B-tree for answering text queries. Particularly, the focus has been in two-dimensions where one has to build an index on a set $S$ of points to answer range queries specified by a rectangle $(x_\ell, x_r, y_\ell, y_r)$ and the answer is $\{(x_i, y_i) \in S \mid x_\ell \le x_i \le x_r, y_\ell \le y_i \le y_r\}$. Although there were lower bounds shown that good I/O performance may not be possible in terms of clustering output occurrences, it was suggested that by using popular practical range query structures good performance may be obtained. The search performance relies heavily on how range searching is supported. It is thus interesting to see whether such an index is realizable, and investigate its performance when we apply the popular R-trees and KD-trees as the core range query structures. Also, of particular interest is to develop the range query structures which are tailor-made for the queries arising in above application. In this paper, we address these issues and show the first practical I/O-efficient compressed text index, which achieves a whole range of space-time trade-offs when compared with String B-tree.

Our work is based on the principle that elegant theory drives and begets many new practically applicable methods. Our work is based on the strict theoretical foundations of Burrows-Wheeler transform and combines theoretically and practically elegant structures like String B-tree/R-trees to obtain a space-saving index for pattern matching in external memory.

### A. Related Work

Text compression has been a very well-researched area. Starting from Ziv-Lempel (LZ) [28] scheme which achieves high-order entropy compression, many other schemes and variants have been developed. More recently, another compression scheme called Burrows-Wheeler Transform (BWT) [6] was introduced. This scheme was later shown to be particularly useful in text indexing applications: Ferragina and Manzini [11] introduced a new text index based on BWT which takes space nearly about the compressed text (thus much less than that of suffix trees or suffix arrays) and at the same time answering pattern matching queries efficiently. A slightly orthogonal approach was taken by Grossi and Vitter [14] which compressed the suffix arrays using a principle similar to BWT. Since then, this field has been thriving with many other variants and improvements are achieved (see the excellent survey by Navarro and Mäkinen [23]). The concept of BWT has also been applied to compress and index XML (or tree shaped) documents [10]. A good collection of practical solutions and benchmarks are collected on the Pizza&Chili website [24]. However, most of the work until now has mainly stayed focussed on internal memory (RAM model) computations.

The field of orthogonal range searching has a long history (see the comprehensive survey in [1]). Staring with lower bounds given by Chazelle [7], there has been a plethora of results attempting to address this in the internal as well as the external memory model. When $occ$ is the size of the output of the orthogonal range query, Hellerstein et al [16] showed that query bounds having the term $occ/B$ I/Os cannot be achieved if we can only use linear-sized structures (here $B$ denotes the size of disk page). For non-replicating structures, the lower bound of $\Omega(\sqrt{n/B} + occ/B)$ I/Os was shown by Kanth and Singh [18]. Arge et al [2] showed a worst-case R-tree achieving this bound. It was also shown by Arge and Vitter [3] that if we can use about $\log n$ times more space, then the optimal I/O bound of $O(\log_B n + occ/B)$ will be achievable. Nevertheless, there have been many practical structures widely used in database community which use linear space and (although with no worst-case guarantee) achieve practically good I/O performance.

Many other recent related research has been appearing in database community. In [25], Sinha et al. investigated various approaches to improve the locality of reference in suffix arrays. The data here resides on the disk. They used suffix arrays as a measure of space saving with respect to suffix trees and showed that one can achieve about three times faster in query performance by using a combination of small trie with suffix array-like blocked data structure. Results focussing on the need of practical approaches have also been appearing. In [26], [5], the focus is on the practical issues involved in suffix tree construction. Arroyuelo and Navarro [4] have attempted to present the first compressed index in the external memory based on LZ-trie based index. Their index achieved space bounds being very close to the text size, but nevertheless incurring much more I/Os for matching patterns (especially when the pattern is long). Their index achieved reasonable I/O throughput while reporting matching location: 5–10% of the disk pages examined come from the actual occurrences. However, this is still much lower than almost close to 100% throughput achieved by suffix arrays and String B-trees.

## II. Preliminaries

In this section, we introduce the String B-tree [9], which is theoretically the best text index in the external memory model. Due to limited space, we omit the explanations for the suffix trees [22], [27] (which is a compact trie containing all suffixes of a text) and the suffix arrays [21] (which is an array containing (the locations of) all suffixes of a text, sorted in lexicographical order), and assume the readers have fair understanding on these topics. For those readers who may be unfamiliar, we recommend the reference [15] for details.

### A. String B-tree

While suffix trees and suffix arrays exhibit good searching performance in the internal memory model, they do not work well when they are stored on disk, as the searching process may invoke many random I/O accesses. To improve the I/O access behavior, Ferragina and Grossi [9] proposed the String B-tree (SBT) index for a text $T$, which basically organized the entries of the suffix array with a B-tree. Each node in the SBT occupies 1 disk page, storing $\Theta(B)$ suffix array entries. As a result, SBT has $\Theta(\log_B n)$ levels.

The SA range of a pattern $P$ (which is the contiguous range in the suffix array containing all occurrences of $P$ in $T$) can be found by traversing the SBT analogous to searching a key in a B-tree. When a node of the SBT is visited, the entries in the node, which are essentially the suffixes of $T$, are then used to compare with $P$ and tell which child node to be visited

T' = acg tac gtg cgt

| pos | sorted suffix |
|---|---|
| 1 | acg tac gtg cgt |
| 4 | cgt |
| 3 | gtg cgt |
| 2 | tac gtg cgt |

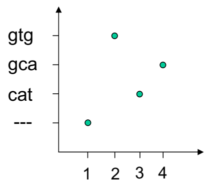
sorted suffix of T'

(a)

| $i$ | SA[$i$] | $c_i$ |
|---|---|---|
| 1 | 1 | --- |
| 2 | 4 | gtg |
| 3 | 3 | cat |
| 4 | 2 | gca |

GBWT of T'

(b)



2D representation of GBWT

(c)

Figure 1. Example of the GBWT. (a) The suffixes of $T' = \texttt{acg tac gtg cgt}$ sorted in lexicographical order. (b) The suffix array $SA'$ and the meta-characters $c_i$, in which GBWT is defined as the tuples $(i, c_i)$ for all $i$. (c) The 2-d representation of GBWT.

next. In addition, each node is coupled with a blind trie of the corresponding suffixes to further reduce the I/Os incurred in the above comparison.[1] The following lemma summarizes the performance of the SBT in [9]:

*Lemma 1:* Let $T$ be a text of $n$ characters. The SBT of $T$ can be stored in $O(n/B)$ disk pages. Then for any query pattern $P$, we can compute the suffix range of $P$ in $O(|P|/B + \log_B n)$ I/Os.

## III. GBWT and I/O-efficient Compressed Text Index

Despite the excellent I/O performance in pattern matching, a major disadvantage of the SBT is the storage requirement. While the indexed text $T$ can naively be stored in $O(n)$ bits when the alphabet size $|\Sigma|$ is a constant (more precisely, $O(n \log |\Sigma|)$ bits when the alphabet of $T$ is $\Sigma$), SBT will require $O(n \log n)$ bits; the exact ratio can be up to a factor of 20 to 60 times in practice, when the indexed texts are DNA sequences (whose alphabet size is 4). Chien et al [8] proposed a variant of the Burrows-Wheeler Transform [6], called *geometric BWT* (GBWT), and show that it can be used to design a compressed text index that works well in the external memory model. Their main idea is very simple: maintaining a subset of suffixes instead of all suffixes in the SBT. In the remainder of the section, we will first explain briefly the GBWT, then describe how to design an external-memory index based on GBWT, and show how pattern searching can then be supported.

### A. Definition of GBWT

Given a text $T$ and a blocking factor $d$, let $T'[1..n/d]$ be the text formed by blocking every consecutive $d$ characters of $T$ to form a single meta-character. For example, if $T = \texttt{acgtacgtgcgt}$ and $d = 3$, then $T'$ would be equal to

$$T' = \texttt{acg} \quad \texttt{tac} \quad \texttt{gtg} \quad \texttt{cgt},$$

consisting of 4 meta-characters. Thus, the suffix of $T'$ starting at 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 $T'$.

The GBWT of $T$ is defined as a specific set of 2-tuples $(x, c)$ based on $T'$ and $SA'$, where $x$ is an integer between 1 to $n/d$, and $c$ is a meta-character. Recall that $SA'[i]$ corresponds to the $i$th lexicographically smallest suffix in $T'$. For such a suffix, we let $c_i$ be the meta-character before it (that is, with $d$ original characters), but written in the *reverse* order. For instance, in the above example, $SA'[2] = 4$, so that cgt is the second lexicographically smallest suffix of $T'$. Then we have $c_2 = \texttt{gtg}$. The GBWT of $T$ is simply the set of tuples $S = \{(i, c_i) \mid 1 \le i \le n/d\}$. See Figure 1 for a complete example.

Indeed, each meta-character $c_i$ in a GBWT tuple can naturally be thought as an integer, whose bit representation is exactly the $d \log |\Sigma|$ bits we use to encode $c_i$. For instance, in our example, suppose the original characters $\texttt{a}, \texttt{c}, \texttt{g}, \texttt{t}$ are encoded by $\texttt{00}, \texttt{01}, \texttt{10}$, and $\texttt{11}$, respectively. Then a meta-character cat will be encoded by 100011, and can be thought as the integer 35. Consequently, the GBWT of $T$ can be represented by $n/d$ points in the 2-d plane. See Figure 1(c) for an example.

[1]The blind trie of a set of strings is a specialized compact trie where each internal node stores the length of the common prefix of the descendant strings, and each edge from a node is labeled by the first character after the common prefix.

## B. *I/O-efficient Compressed Text Index*

The fundamental component of Chien et al's compressed text index is the SBT defined on $T'$. Since there are only $n/d$ suffixes in $T'$, the space of the SBT of $T'$ becomes $O((n/d)\log(n/d))$ bits, which is only $1/d$ of the space of a standard SBT. Although we have not maintained all the suffixes, we can still support pattern searching with such an SBT, though in a very restricted form. Precisely, it can only report those occurrences of $P$ in $T$ which occur at positions of the form $d * i + 1$.

To extend the power of this SBT, we obtain the GBWT tuples, and store them in a 2-d index, such as R-tree or KD-tree; alternatively, we may represent the GBWT tuples by an integer array, and index the array by the external-memory wavelet tree [20], [17]. In either ways, efficient orthogonal range queries on the GBWT tuples can be supported.

*1) Answering Pattern Matching Queries:* We now show how to use orthogonal range queries on the GBWT tuples for pattern matching. In particular, we find all those occurrences of $P$ in $T$ whose starting position is *inside* some meta-character in $T'$. That is, those occurring at positions $i$ (in $T$) with $i \bmod d = k$, where $k$ may not be 1. We call such an occurrence an *offset-k* occurrence. Here, we require that $P$ is longer than $d$, so that its offset-$k$ occurrence does not start and end inside the same character in $T'$. For instance, let $P = \texttt{tgcg}$ and consider the example text of Figure 1. We see that $P$ indeed occurs once in $T'$, whose starting position is inside the third meta-character $\texttt{gtg}$ of $T'$. Also, because this occurrence starts at the second position inside a meta-character, we call this an offset-2 occurrence.

To find all offset-$k$ occurrences of $P$ in $T$, we first partition $P$ into two parts: (i) $\widehat{P}$, which contains the first $d - k + 1$ characters of $P$, and (ii) $\tilde{P}$, which contains the remaining characters of $P$. It is shown that we can find all offset-$k$ occurrences as follows:

Step 1. Search in SBT of $T'$ to obtain the SA range $[\ell, r]$ of $\tilde{P}$, so that $SA'[\ell..r]$ are all occurrences of $\tilde{P}$ in $T'$.

Step 2. Use the reverse of $\widehat{P}$ to construct a range $[c, c']$, and search in the 2-d index for all points within the rectangle $[\ell, r] \times [c, c']$. Output each point as an occurrence.

For example, suppose we want to find the offset-3 occurrence of $P = \texttt{cgtgc}$ in our example text $T'$. We know that whenever such an occurrence appears, it must have $\tilde{P} = \texttt{gtgc}$ appearing as a prefix of some suffix $X$ of $T'$, and $\widehat{P} = \texttt{c}$ as a suffix in the meta-character before $X$. Indeed, we see that the latter condition implies the meta-character before $X$, when written in *reverse* order, is of the form $\texttt{c??}$; equivalently, such a reverse meta-character is within the range $[\texttt{caa}, \texttt{ctt}]$.

We apply the above to find offset-$k$ occurrences for $k = 2, 3, \ldots, d$, giving the following result:

*Theorem 1:* Suppose the SBT of $T'$ and a 2-d range query index for the GBWT tuples are stored. Then all occurrences of $P$ with starting and ending positions inside different (meta-)characters of $T'$ can be found using $d$ searches in the SBT and $d$ orthogonal range queries.

Finally, we have to handle the case when $P$ is shorter than $d$, so that both its starting and ending positions can be inside the same meta-character of $T'$. In this case, we can apply the traditional inverted files for searching such patterns. In some applications where the query patterns are known to be longer than $d$, we can simply discard the inverted files to save space. Thus, a perfect choice of $d$ would be the minimum length of a query pattern one may want to search.

## IV. Implementation Details

## A. *Tuning of String B-Tree*

Recall that in the SBT search, when we visit a node, we need to compare $P$ with the suffixes in the node so as to guide the traversal. In the current process, such a comparison is aided by the blind trie in the node, and the target is to find out the relative lexicographical order of $P$ among the suffixes in the node. However, in contrast to a normal compact trie, the blind trie lacks the complete information in the edges (only the first character is stored), so that searching in the blind trie would actually invoke some random I/Os to recover the edge information.

*Heuristic 1:* Our first heuristic attempts to avoid these random I/Os completely, by explicitly storing the first 16 characters of each suffix inside the node. Consequently, if the relative lexicographical order of $P$ can be determined directly, there will be no further I/Os, and we can proceed immediately to the next node in the traversal. A minor drawback with this heuristic is that each node has to reserve the space for these first 16 characters, such that we can no longer store the same number of suffixes as before. As a result, the branching factor of the SBT will be decreased a bit, which may make the height of the tree taller.

*Heuristic 2:* Our second heuristic is a standard and well-known trick: storing the first few levels of SBT explicitly in the internal memory. While the size of an SBT could be huge, the size of the first few levels, even up to half of all levels, is only $O(\sqrt{n})$ space. This is quite manageable for indexing texts in most applications. As a result, the traversal in these levels will not incur any I/O.

### B. *Tuning of Range Query Index*

While R-tree and KD-tree are general-purpose indexes for 2-d range query, the points in our application are usually skewed. Precisely, let us consider the ranges of $x$ and $y$ coordinates in a common setting. The $x$ direction in our case represents the various suffixes of $T'$, so that the range can be up to millions of values. On the other hand, the $y$ direction in our case represents the meta-characters, so that when $|\Sigma|$ and $d$ are small (say, $|\Sigma| = 4$ for DNA texts and $d = 4$), the range may only be up to a few thousands. As a result, all the points we index fall into a very narrow stripe, and the traditional indexes like R-tree or KD-tree may fail to perform well.

*Heuristic 3:* Our third heuristic is to fine tune the criteria in the R-tree or KD-tree construction, such as ordering the x-cuts and y-cuts in KD-tree, so that a query invoked in our application can be answered more effectively.

### C. *Inverted Index for Short Patterns*

The GBWT index handles all pattern occurrences which cross at least one meta-character boundary. To handle those occurrences which appear inside of a meta-character, we first generate a generalized suffix tree of all (distinct) meta-characters–there are at most $|\Sigma|^d$ of them. For each substring $s$ of a distinct meta-character $c$, we store a pointer from $s$ to $c$ in the generalized suffix tree. The search in the suffix tree follows standard procedure, and once we match $P$ and stop at a location $\rho$, we can identify all meta-characters $c$'s that contain $P$ as its substring, and proceed to find the occurrences of those $c$'s. These occurrences of $c$'s can readily be reported by searching the sparse SBT. However, in case the meta-character $c$ occurs rarely in $T$ (much less than $B$ occurrences), then the list of all its positions in $T$ is directly maintained within the generalized suffix tree so as to avoid a random I/O access in the sparse SBT.

There are about $|\Sigma|^d$ meta-characters to be indexed (for DNA text with $d = 4$, this number is about $4^4$ = 256 entries). Generally, this index takes a very small fraction of space. This index always reports $occ$ occurrences in at most $O(occ/B)$ I/Os.

## V. Experimental Results

We implemented our data structures using the `C` programming language, compiled with the `gcc` compiler version (4.2.4). All the experiments are simulation results, where we organize our memory logically into disk pages and the performance of a query is measured by the number of accessed disk pages. Our simulation is tested in an `Intel(R) Pentium(R) 4 CPU 3.40 GHz` machine, with 1 GBytes of RAM. The OS is `Ubuntu 4.2.4`. The disk page size $B$ is 1024 bytes.

### A. *The Settings*

We compared the performance of the following indexes:

1) `ST`: The suffix tree under naive blocking strategy. Within each disk page where a subtree (or a forest of subtrees) of the suffix tree is stored, we avoid using pointers to indicate the parent-child relationship between the nodes. Instead, the subtree structure is stored by the parentheses encoding, saving 1.75 bytes per node on average.[2]
2) `ST + SA`: The suffix tree under naive blocking strategy and the suffix array. When query is performed, we traverse the suffix tree using the input pattern to some location $\rho$ in the tree as before. However, we do not perform the simple tree traversal to report the occurrences. Instead, we obtain the range of the suffix array that is below $\rho$, and report the desired occurrences by accessing the corresponding range in the suffix array. There is an overhead of 8 bytes per suffix tree node for storing the range information.
3) `FSBT`: The full version of the String B-tree that contains all suffixes. Each SBT node is stored in a disk page, and the tree structure of the blind trie is stored by the parentheses encoding. This saves 1.75 bytes per blind trie node on average.
4) `FSBT + SA`: The full version of String B-tree and the suffix array. When query is performed, we use the SBT to report the suffix range of the input pattern $P$. Afterwards, we report the desired occurrences by accessing the corresponding range in the suffix array.
5) `SSBT-`$d$ `+ Rtree`: The sparse version of the String B-tree with R-tree. The number of characters, $d$, in a meta-character is either 2, 4, or 8.
6) `SSBT-`$d$ `+ KD-tree`: The sparse version of the String B-tree with KD-tree. The number of characters, $d$, in a meta-character is either 2, 4, or 8.
7) `SSBT-`$d$ `+ Wavelet`: The sparse version of the String B-tree with wavelet tree. The number of characters, $d$, in a meta-character is either 2, 4, or 8.

[2]We can encode the structure of a tree as follows: Perform a pre-order traversal. During the traversal, we write a ‘`(`’ when we visit a node for the first time, and write a ‘`)`’ when we visit a node for the last time.

Table I
EFFECT OF HEURISTICS ON PERFORMANCE OF SBT. THE ENTRIES INDICATE THE AVERAGE NUMBER OF I/OS TO REPORT THE SUFFIX RANGE OF THE QUERY PATTERN.

| | Normal | Heuristic 1 | Heuristic 2 | Both |
|---|---|---|---|---|
| $d = 1$ | 15.15 | 9.20 | 11.27 | 5.32 |
| $d = 2$ | 18.45 | 9.32 | 12.15 | 3.02 |
| $d = 4$ | 22.75 | 16.63 | 11.44 | 5.32 |
| $d = 8$ | 32.04 | 28.54 | 12.26 | 9.12 |

Table II
EFFECT OF HEURISTIC ON PERFORMANCE OF KD-TREE. THE ENTRIES INDICATE THE AVERAGE I/OS PER QUERY TO OUTPUT OCCURRENCES BY THE RESPECTIVE KD-TREE.

| Pattern Type | Default KD-tree (I/Os) | Tuned KD-tree (I/Os) |
|---|---|---|
| Set-1000 | 13 | 11 |
| Set-3000 | 25 | 21 |
| Set-10000 | 60 | 53 |

All texts in our experiment are randomly generated texts with alphabet size = $4$. The lengths of the texts include 1M, 2M, and 4M. For the pattern queries, apart from testing patterns with different lengths, we also test patterns selected from the following sets:

1) Set-0: Patterns with no occurrences in the text.
2) Set-1000: Patterns with average occurrences = 1000.
3) Set-3000: Patterns with average occurrences = 3000.
4) Set-10000: Patterns with average occurrences = 10000.

### B. *Tuning of SBT*

The main purpose of having SBT (both FSBT and SSBT) in our indexes is to obtain the suffix range of an input pattern $P$. The searching I/O in such a procedure is independent of the number of occurrences of $P$. In our first experiment, we performed various fine tunings (Heuristic 1, or Heuristic 2, or both as described in Section IV) and compared the resulting I/O performance to obtain the suffix range when searching patterns with different lengths.

We tested the performance on a single input text which contains 4M characters. The text is indexed by `FSBT`, `SSBT`-2, `SSBT`-4, and `SSBT`-8. We select at random 1024 query patterns of length 25 from the input text. The results are shown in Table I.

As Heuristic 1 and Heuristic 2 both target at reducing the I/Os from searching the SBT, a great improvement (about 60–80% reduction) was observed for all different kinds of SBT when both heuristics are applied. This confirmed the practicality of the two heuristics. Thus from here onwards, we shall assume all SBTs to include both heuristics.

### C. *Tuning of KD-tree*

As mentioned in Section IV, we may improve the performance of the 2D index if we take the distribution of points into consideration. Table V-C reports the performance of KD-tree with the default implementation and with tuning using Heuristic 3. In our experiment, we chose a random text of 4M characters which was indexed by `SSBT`-4. The KD-tree was used for outputting the occurrences. The pattern queries are selected from Set-1000, Set-3000, and Set-10000, respectively.

The above shows that fine tuning of KD-tree consistently gives minor improvements in query I/Os. Nevertheless, since fine tuning of KD-tree only re-organize the partition of the points, it will not add any extra space so that it should be used. Note that we have not performed tuning on R-tree though we expect similar improvements can be achieved.

### D. *Empirical Space*

All the indexes we compared are linear-space indexes. We have constructed these indexes over texts of different lengths, including 1M, 2M, and 4M. The following is a close approximation to the space we found empirically. The space of `ST`, `FSBT`, `SSBT`-$d$ are measured in the number $n$ of characters, while the space of `R-tree`, `KD-tree`, and `Wavelet` are measured in terms of the size of the `SSBT`.

Table III
SPACE OF THE INVERTED INDEX FOR SHORT PATTERNS.

| Meta-char | Worst-Case (bytes) | Empirical (bytes) |
|---|---|---|
| $d = 2$ | 8M | 1K |
| $d = 4$ | 4M | 100K |
| $d = 8$ | 102M | 102M |

| | | | |
|---|---|---|---|
| `ST`: | $17n$ bytes | `SSBT`-$d$: | $9n/d$ bytes |
| `ST+ SA`: | $21n$ bytes | `R-tree`: | $1.1 \times$ size of `SSBT` |
| `FSBT`: | $11n$ bytes | `KD-tree`: | $1.4 \times$ size of `SSBT` |
| `FSBT + SA`: | $15n$ bytes | `Wavelet`: | $0.55 \times$ size of `SSBT` |

For the space of the inverted index for short patterns, it consists of two parts: The generalized suffix tree and the list of pointers to the text. Table III shows the theoretical space (worst-case) and the empirical space of the inverted index when indexing a 4M text, for $d = 2, 4, 8$.

When $d = 2$ or $d = 4$, most of the meta-characters will have more than $\Theta(B)$ occurrences in $T$, making the list of positions in the generalized suffix tree very small. (Here, we set $\Theta(B) = 256$ since 1024 bytes can store at most 256 positions.) On the other hand, when $d = 8$ and text length = 4M, most of the meta-characters occur only a few times in $T$, so that the list of positions include all possible $n/d$ positions of $T$, making the empirical space very bad.

Note that because the inverted index for $d = 8$ is huge, we should either use $d = 8$ when we only search for patterns longer than 8, or when the input text is very long so that we are willing to pay for the extra 102M bytes anyway.

## *E. Effect on Pattern Length*

We experimented with various pattern sizes for empty queries, but our results show no significant difference in I/Os. Thus our I/Os are invariant of pattern size as we expected. Note that this is a significant improvement over the LZ-index by Arroyuelo and Navarro [4] where the number of I/Os scales up heavily with larger pattern sizes.

## *F. Effect on Number of Occurrences*

When the number of occurrences of a pattern is high, the query I/Os is dominated by the I/O spent in occurrence reporting. In this part, we performed experiments to investigate the space/time tradeoff of various indexes concerning the occurrence reporting. We chose a random text of 4M characters, indexed with various indexes, and tested for pattern queries from Set-0, Set-1000, Set-3000, and Set-10000.

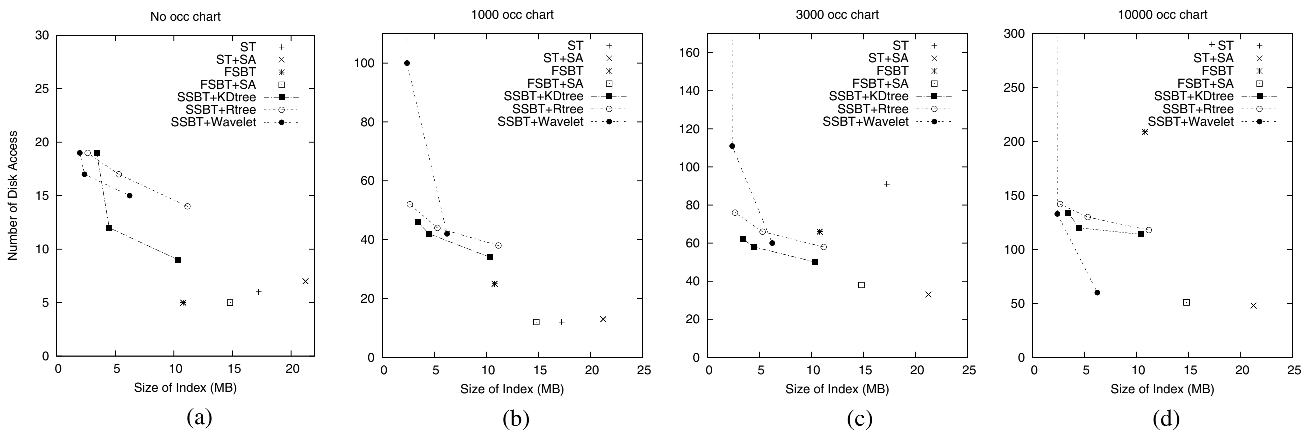


Figure 2. Performances of Indexes on Various Number of Occurrences

Figure 2(a) shows the query I/Os when the pattern searching ends up with no output. This case basically separates the I/Os required for matching pattern apart form I/Os required to output the occurrences (which can be an overshadowing factor). The three points in each index correspond to $d = 2, 4, 8$ cases. With $d = 8$ our indexes are up to 3 or 4 times smaller than the String B-tree. However, the number of I/Os go up by about 3 to 4 times as well. For the analysis, we can roughly count the space for FSBT as twice of suffix array (SA). Our sparse indexes–SSBT + 2D range index—take about $2 * |\texttt{FSBT}|/d$ space, which is $4 * |\texttt{SA}|/d$ space. Among our indexes we can see that SSBT+KD-tree with $d = 4$ offers a very good space-time trade-off.

The subsequent figures show the case when I/Os are dominated by reporting outputs. First we start with the moderate 1000 output case. In this case there are about 1000 positions matching the query. We can first observe that our indexes perform about 2–3 times more I/Os compared to suffix-array-based indexes. The main reason for the factor of 2 being that our point is a tuple and hence costs 8 bytes per point as against 4 bytes for SA value. Apart from that our indexes perform well in terms of the throughput. In the case of 10000 occurrences the worst of our index takes about 130 I/Os which amounts to about 66% of the best possible (80 I/Os). In the best case we achieve about 85% of I/O throughput. This is in stark contrast with [4] whose throughput is about 5–10%. We also note that I/Os for wavelet-tree-based index is much higher as expected.

There is a further interesting point to note. In the case of 10000 occurrences, SSBT + Wavelet with $d = 2$ has query performance very close to that of FSBT + SA, or ST + SA. The reason is that the wavelet tree in our experiment only has 2 levels, and in case $d$ is small, the I/O for outputting occurrences becomes $O(4^d + occ/B)$ instead of $O(occ \log_B n)$.

## Acknowledgments

The authors wish to thank the anonymous reviewers for their careful reading and their constructive comments, which have considerably improved the quality of the paper.

## 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] L. Arge, M. de Berg, H. J. Haverkort, K. Yi. The Priority R-tree: A Practically Efficient and Worst-Case Optimal R-tree. *ACM Transactions on Algorithms*, 4(1), 2008.

[3] L. Arge and J. S. Vitter. Optimal External Memory Interval Management. *SIAM Journal on Computing*, 32(6):1488–1508, 2003.

[4] D. Arroyuelo and G. Navarro. A Lempel-Ziv Text Index on Secondary Storage. In *Proceedings of Symposium on Combinatorial Pattern Matching*, pages 83–94, 2007.

[5] S. J. Bedathur and J. R. Haritsa. Engineering a Fast Online Persistent Suffix Tree Construction. In *Proceedings of International Conference on Data Engineering*, 2004.

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

[7] B. Chazelle. Lower Bounds for Orthogonal Range Searching, I: The Reporting Case. *Journal of the ACM*, 37(2):200–212, 1990.

[8] Y.-F. Chien, W.-K. Hon, R. Shah, and J. S. Vitter. Geometric Burrows-Wheeler Transform: Linking Range Searching and Text Indexing. In *Proceedings of Data Compression Conference*, pages 252–261, 2008.

[9] 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.

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

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

[12] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed Representations of Sequences and Full-Text Indexes. *ACM Transactions on Algorithms*, 3(2), 2007.

[13] 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.

[14] 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.

[15] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, NY, USA, 1997.

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

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

[18] 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.

[19] D. E. Knuth, J. H. Morris, and V. B. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

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

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

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

[23] G. Navarro and V. Mäkinen. Compressed Full-Text Indexes. *ACM Computing Surveys*, 39(1), 2007.

[24] Pizza&Chili Corpus – Compressed Indexes and their Testbeds. `http://pizzachili.di.unipi.it/`, 2005.

[25] R. Sinha, S. J. Puglisi, A. Moffat, and A. Turpin. Improving Suffix Arrays Locality for Fast Pattern Matching on Disk. In *Proceedings of International Conference on Management of Data*, pages 661–672, 2008.

[26] S. Tata, R. A. Hankins, and J. M. Patel. Practical Suffix Tree Construction. In *Proceedings of International Conference on Very Large Data Bases*, pages 36–47, 2004.

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

[28] J. Ziv and A. Lempel. Compression of Individual Sequences via Variable Length Coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.