Compression, Indexing, and Retrieval for Massive String Data *

Wing-Kai Hon[1], Rahul Shah[2], and Jeffrey Scott Vitter[3]

[1] National Tsing Hua University, Taiwan, wkhon@cs.nthu.edu.tw
[2] Louisiana State University, USA, rahul@csc.lsu.edu
[3] Texas A&M University, USA, jsv@tamu.edu

Abstract. The field of compressed data structures seeks to achieve fast search time, but using a compressed representation, ideally requiring less space than that occupied by the original input data. The challenge is to construct a compressed representation that provides the same functionality and speed as traditional data structures. In this invited presentation, we discuss some breakthroughs in compressed data structures over the course of the last decade that have significantly reduced the space requirements for fast text and document indexing. One interesting consequence is that, for the first time, we can construct data structures for text indexing that are competitive in time and space with the well-known technique of inverted indexes, but that provide more general search capabilities. Several challenges remain, and we focus in this presentation on two in particular: building I/O-efficient search structures when the input data are so massive that external memory must be used, and incorporating notions of relevance in the reporting of query answers.

1 Introduction

The world is drowning in data! Massive data sets are being produced at unprecedented rates from sources like the World-Wide Web, genome sequencing, scientific experiments, business records, image processing, and satellite imagery. The proliferation of data at massive scales poses serious challenges in terms of storing, managing, retrieving, and mining information from the data.

Pattern matching — in which a pattern is matched against a massively sized text or sequence of data — is a traditional field of computer science that forms the basis for biological databases and search engines. Previous work has concentrated for the most part on the internal memory RAM model. However, we are increasingly having to deal with massive data sets that do not easily fit into internal memory and thus must be stored on secondary storage, such as disk drives, or in a distributed fashion in a network.

Suffix trees and suffix arrays, which are the traditional data structures used for pattern matching and a variety of other string processing tasks, are often

* Supported in part by Taiwan NSC grant 96–2221–E–007–082–MY3 (W. Hon) and USA National Science Foundation grant CCF–0621457 (R. Shah and J. S. Vitter).

"bloated" in that they require much more space than that occupied by the uncompressed input data. Moreover, the input data are typically highly compressible, often by a factor of 5–10. When compared with the size of the input data in compressed form, the size of suffix trees and suffix arrays can be prohibitively large, often 20–150 times larger than the compressed data size. This extra space blowup results in increased memory resources and energy usage, slower data access (because the bloated data must reside in the slower levels of the memory hierarchy), and reduced bandwidth.

1.1 Key Themes in this Presentation

In this presentation we focus on some emerging themes in the area of pattern matching for massive data. One theme deals with the exciting new field called *compressed data structures*, which addresses the bloat exhibited by suffix trees and suffix arrays. There are two simultaneous goals: space-efficient compression and fast indexing. The last decade has seen much progress, both in theory and in practice. A practical consequence is that, for the first time, we have space-efficient indexing methods for pattern matching and other tasks that can compete in terms of space and time with the well-known technique of inverted indexes [73, 52, 74] used in search engines, while offering more general search capabilities. Some compressed data structures are in addition *self-indexing* [61, 19, 20, 28], and thus the original data can be discarded, making them especially space-efficient. The two main techniques we discuss — compressed suffix array (CSA) and FM-index — are self-indexing techniques that require space roughly equal to the space occupied by the input data in compressed format.

A second theme deals with external memory access in massive data applications [1, 71, 70], in which we measure performance in terms of number of I/Os. A key disadvantage of CSAs and the FM-index is that they do not exhibit locality of reference and thus do not perform well in terms of number of I/Os. If the input data are so massive that the CSA and FM-index do not fit in internal memory, their performance is slowed significantly. There is much interesting work on compressed data structures in external memory (e.g., [2, 4, 27, 17, 16, 38, 48, 55]), but major challenges remain.

The technique of sparsification allows us to reduce space usage but at the same time exploit locality for good I/O performance and multicore utilization. We discuss sparsification in two settings: One involves a new transform called the geometric Burrows-Wheeler transform (GBWT) [9, 34] that provides a link between text indexing and the field of range searching, which has been studied extensively in the external memory setting. In this case, a sparse subset of suffix array pointers are used to reduce space, and multiple offsets in the pattern must be searched, which can be done especially fast on multicore processors. The other setting introduces the notion of relevance in queries so that only the most relevant (or top-k) matches [53, 6, 64, 69, 36] are reported. The technique of sparsification provides approximate answers quickly in a small amount of space [36].

Besides the external memory scenario, other related models of interest worth exploring include the cache-oblivious model [25], data streams model [54], and practical programming paradigms such as multicore [65] and MapReduce [11].

2 Background

2.1 Text Indexing for Pattern Matching

We use $T[1\,..\,n]$ to denote an input string or text of n characters, where the characters are drawn from an alphabet Σ of size σ. The fundamental task of text indexing is to build an index for T so that, for any query pattern P (consisting of p characters), we can efficiently determine if P occurs in T. Depending upon the application, we may want to report all the *occ* locations of where P occurs in T, or perhaps we may merely want to report the number *occ* of such occurrences.

The string T has n suffixes, starting at each of the n locations in the text. The ith suffix, which starts at position i, is denoted by $T[i\,..\,n]$. The *suffix array* [26, 49] $SA[1\,..\,n]$ of T is an array of n integers that gives the sorted order of the suffixes of T. That is, $SA[i] = j$ if $T[j\,..\,n]$ is the ith smallest suffix of T in lexicographical order. Similarly, the inverse suffix array is defined by $SA^{-1}[j] = i$. The *suffix tree* ST is a compact trie on all the suffixes of the text [51, 72, 68]. Suffix trees are often augmented with suffix links. The suffix tree can list all *occ* occurrences of P in $O(p + occ)$ time in the RAM model. Suffix arrays can also be used for pattern matching. If P appears in T, there exist indices ℓ and r such that $SA[\ell], SA[\ell + 1], \ldots, SA[r]$ store all the starting positions in text T where P occurs. We can use the longest common prefix array to improve the query time from $O(p \log n + occ)$ to $O(p + \log n + occ)$ time.

Suffix trees and suffix arrays use $O(n)$ words of storage, which translates to $O(n \log n)$ bits. This size can be much larger than that of the text, which is $n \log \sigma$ bits, and substantially larger than the size of the text in compressed format, which we approximate by $nH_k(T)$, where $H_k(T)$ represents the kth-order empirical entropy of the text T.

2.2 String B-trees

Ferragina and Grossi introduced the *string B-tree* (SBT) [16], an elegant and efficient index in the external memory model. The string B-tree acts conceptually as a B-tree over the suffix array; each internal node does B-way branching. Each internal node is represented as a "blind trie" with B leaves; each leaf is a pointer to one of the B child nodes. The blind trie is formed as the compact trie on the B leaves, except that all but the first character on each edge label is removed. When searching within a node in order to determine the proper leaf (and therefore child node) to go to next, the search may go awry since only the first character on each edge is available for comparison. The search will always end up at the right place when the pattern correctly matches one of the leaves, but in the case where there is no match and the search goes awry, a simple scanning of the original text can discover the mistake and find the corrected position where the pattern belongs. Each block of the text is never scanned more than once and thus the string B-tree supports predecessor and range queries in $O(p/B + \log_B n + occ/B)$ I/Os using $O(n)$ words (or $O(n/B)$ blocks) of storage.

3 Compressed Data Structures

In the field of compressed data structures, the goal is to build data structures whose space usage is provably close to the entropy-compressed size of the text. A simultaneous goal is to maintain fast query performance.

3.1 Wavelet Trees

The *wavelet tree*, introduced by Grossi et al. [28, 24], has become a key tool in modern text indexing. It supports rank and select queries on arrays of characters from Σ. (A rank query $rank(c, i)$ counts how many times character c occurs in the first i positions of the array. A select query $select(c, j)$ returns the location of the jth occurrence of c.) In a sense, the wavelet tree generalizes the rank and select operations from a bit array [59, 57] to an arbitrary multicharacter text array T, and it uses $nH_0(T) + t + O(n/\log_\sigma n)$ bits of storage, where n is the length of the array T, and t is the number of distinct characters in T.

The wavelet tree is conceptually a binary tree (often a balanced tree) of logical bit arrays. A value of 0 (resp., 1) indicates that the corresponding entry is stored in one of the leaves of the left (resp., right) child. The collective size of the bit arrays at any given level of the tree is bounded by n, and they can be stored in compressed format, giving the 0th-order entropy space bound. When $\sigma = O(\operatorname{polylog} n)$, the height and traversal time of the wavelet tree can be made $O(1)$ by making the branching factor proportional to σ^ϵ for some $\epsilon > 0$ [21].

Binary wavelet trees have also been used to index an integer array $A[1\,..\,n]$ in linear space so as to efficiently support *position-restricted queries* [35, 45]: given any index range $[\ell, r]$ and values x and y, we want to report all entries in $A[\ell\,..\,r]$ with values between x and y. We can traverse each level of the wavelet tree in constant time, so that the above query can be reported in $O(occ \log t)$ time, where occ denotes the number of the desired entries.

Wavelet tree also work in the external memory setting [35]. Instead of using a binary wavelet tree, we can increase the branching factor and obtain a B-ary (or $\sqrt{B}$-ary) wavelet tree so that each query is answered in $O(occ \log_B t)$ I/Os.

3.2 Compressed Text Indexes

Kärkkäinen [37] exploited Lempel-Ziv compression to develop a text index that, in addition to the text, used extra space proportional to the size of the text (later improved to $O(nH_k(T)) + o(n \log \sigma)$ bits). Query time was quadratic in p plus the time for p 2D range searches. Subsequent work focused on achieving faster query times of the form $O((p + occ) \operatorname{polylog} n)$, more in line with that provided by suffix trees and suffix arrays. In this section we focus on two parallel efforts — compressed suffix arrays and the FM-index — that achieve the desired goal.

Compressed Suffix Array (CSA). Grossi and Vitter [30, 31] introduced the *compressed suffix array* (CSA), which settled the open problem of whether it was possible to simultaneously achieve fast query performance and break the $(n \log n)$-space barrier. In addition to the text, it used space proportional to the text size, specifically, $2n \log \sigma + O(n)$ bits, and answered queries in $O(p/\log_\sigma n + occ \log_\sigma n)$ time. The key idea was to store a sparse representation of the full

suffix array, namely, the values that are multiples of 2^j for certain j. The *neighbor function* $\Phi(i) = SA^{-1}[SA[i] + 1]$ allows suffix array values to be computed on demand from the sparse representation in $O(\log_\sigma n)$ time.

Sadakane [61, 62] showed how to make the CSA *self-indexing* by adding auxiliary data structures so that the Φ function was entire and defined for all i, which allowed the text values to be computed without need for storing the text T. Queries took $O((p + occ) \log n)$ time. Sadakane also introduced an entropy analysis, showing that its space was bounded by $nH_0(T) + O(n \log \log \sigma)$ bits.[4]

Grossi et al. [28] gave the first self-index that provably achieved asymptotic space optimality (i.e., with constant factor of 1 in the leading term). It used $nH_k(T)+o(n)$ bits and achieved $O(p \log \sigma+occ(\log^4 n)/((\log^2 \log n) \log \sigma))$ query time.[5] For $0 \le \epsilon \le 1/3$, there are various tradeoffs, such as $\frac{1}{\epsilon} nH_k(T) + o(n)$ bits of space and $O(p/\log_\sigma n + occ(\log^{2\epsilon/(1-\epsilon)} n) \log^{1-\epsilon} \sigma)$ query time. The Φ function is encoded by representing a character in terms of the contexts of its following k characters. For each character c in the text, the suffix array indices for the contexts following c form an increasing sequence. The CSA achieves high-order compression by encoding these increasing sequences in a context-by-context manner, using 0th-order statistics for each context. A wavelet tree is used to reduce redundancy in the sequence encodings.

FM-index. In parallel with the development of the CSA, Ferragina and Manzini introduced the elegant *FM-index* [19, 20], based upon the *Burrows-Wheeler transform* (BWT) [7, 50] data compressor. The FM-index was the first self-index shown to have both fast performance and space usage within a constant factor of the desired entropy bound for constant-sized alphabets. It used $5nH_k(T) + O(n^\epsilon \sigma^{\sigma+1} + n\sigma/\log n) + o(n)$ bits and handled queries in $O(p + occ \log^\epsilon n)$ time. The BWT of T is a permutation of T denoted by T_{bwt}, where $T_{\text{bwt}}[i]$ is the character in the text immediately preceding the ith lexicographically smallest suffix of T. That is, $T_{\text{bwt}}[i] = T[SA[i]-1]$. Intuitively, the sequence $T_{\text{bwt}}[i]$ is easy to compress because adjacent entries often share the same higher-order context. The "last to first" function LF is used to walk backwards through the text; $LF(i) = j$ if the ith lexicographically smallest suffix, when prepended with its preceding character, becomes the jth lexicographically smallest suffix.

The FM-index and the CSA are closely related: The LF function and the CSA neighbor function Φ are inverses. That is, $SA[LF(i)] = SA[i] - 1$; equivalently $LF(i) = SA^{-1}[SA[i] - 1] = \Phi^{-1}(i)$. A partition-based implementation and analysis of the FM-index, similar to the context-based CSA space analysis described above [28], reduced the constant factor in the FM-index space bound to 1, achieving $nH_k(T) + o(n)$ bits and various query times, such as $O(p + occ \log^{1+\epsilon} n)$ [21, 29]. Intuitively, the BWT T_{bwt} (and the CSA lists) can be partitioned into contiguous segments, where in each segment the context of subsequent text characters is the same. The context length may be fixed (say, k)

[4] We assume for convenience in this presentation that the alphabet size satisfies $\sigma = O(\text{polylog } n)$ so that the auxiliary data structures are negligible in size.

[5] We assume that $k \le \alpha \log_\sigma n - 1$ for any constant $0 \le \alpha \le 1$, so that the kth-order model complexity is relatively small.

or variable. We can code each segment of T_{bwt} (or CSA lists) using the statistics of character occurrences for that particular context. A particularly useful tool for encoding each segment is the wavelet tree, which reduces the coding problem from encoding vectors of characters to encoding bit vectors. Since each individual partition (context) is encoded by a separate wavelet tree using 0th-order compression, the net result is higher-order compression. This idea is behind the notion of "compression boosting" of Ferragina et al. [14].

Simpler implementations for the FM-index and CSA achieve higher-order compression without explicit partitioning into separate contexts. In fact, the original BWT was typically implemented by encoding T_{bwt} using the move-to-front heuristic [19, 20]. Grossi et al. [24] proposed using a single wavelet tree to encode the entire T_{bwt} and CSA lists rather than a separate wavelet tree for each partition or context. Each wavelet tree node is encoded using run-length encoding, such as Elias's γ or δ codes [12]. (The γ code represents $i > 0$ with $2\lfloor \log i \rfloor + 1$ bits, and the δ code uses $\lfloor \log i \rfloor + 2\lfloor \log(\log i + 1) \rfloor + 1$ bits.) Most analyses of this simpler approach showed higher-order compression up to a constant factor [50, 24, 44, 13]. The intuition is that encoding a run of length i by $O(\log i)$ bits automatically tunes itself to the statistics of the particular context.

Mäkinen and Navarro [46] showed how to use a single wavelet tree and achieve a space bound with a constant factor of 1, namely, $nH_k(T) + o(n)$ bits. They used a compressed block-based bit representation [59, 57] to encode each bit array within the single wavelet tree. A similar bound can be derived if we instead encode each bit array using δ coding, enhanced with rank and select capabilities, as done by Sadakane [61, 62]; however, the resulting space bound contains an additional additive term of $O(n \log H_k(T)) = O(n \log \log \sigma)$ bits, which arises from the $2 \log \log i$ term in δ encoding. This additive term increases the constant factor in the linear space term $nH_k(T)$ when the entropy or alphabet size is bounded by a constant, and under our assumptions on σ and k, it is bigger than the secondary $o(n)$ term. Mäkinen and Navarro [46] also apply their boosting technique to achieve high-order compression for dynamic text indexes [8, 47].

Extensions. In recent years, compressed data structures has been a thriving field of research. The CSA and FM-index can be extended to support more complex queries, including dictionary matching [8], approximate matching [40], genome processing [22, 41], XML subpath queries [18], multilabeled trees [3], and general suffix trees [63, 60, 23]. Puglisi et al. [58] showed that compressed text indexes provide faster searching than inverted indexes. However, they also showed that if the number of occurrences (matching locations) are too many, then inverted indexes perform better in terms of document retrieval. The survey by Navarro and Mäkinen [55] also discusses index construction time and other developments, and Ferragina et al. [15] report experimental comparisons.

4 Geometric Burrows-Wheeler Transform (GBWT)

Range search is a useful tool in text indexing (see references in [9]). Chien et al. [9] propose two transformations that convert a set S of points (x_1, y_1), (x_2, y_2), $\ldots$, (x_n, y_n) into text T, and vice-versa. These transformations show a two-way

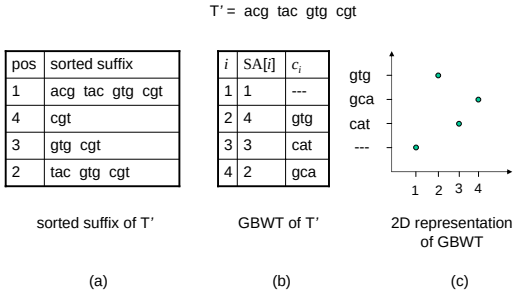

Fig. 1. Example of the GBWT for text $T =$ `acgtacgtgcgt`. The text of metacharacters is $T' =$ `acg tac gtg cgt`. (a) The suffixes of T' sorted into lexicographical order. (b) The suffix array SA' and the reverse preceding metacharacters c_i; the GBWT is the set of tuples (i, c_i), for all i. (c) The 2D representation of GBWT.

connectivity between problems in text indexing and orthogonal range search, the latter being a well-studied problem in the external memory setting and in terms of lower bounds. Let $\langle x \rangle$ be the binary encoding of x seen as a string, and let $\langle x \rangle^R$ be its reverse string. For each point (x_i, y_i) in S, the first transform constructs a string $\langle x_i \rangle^R \# \langle y_i \rangle \$$. The desired text T is formed by concatenating the above string for each point, so that $T = \langle x_1 \rangle^R \# \langle y_1 \rangle \$ \langle x_2 \rangle^R \# \langle y_2 \rangle \$ \ldots \langle x_n \rangle^R \# \langle y_n \rangle \$$. An orthogonal range query on S translates into $O(\log^2 n)$ pattern matching queries on T. This transformation provides a framework for translating (pointer machine as well as external memory) lower bounds known for range searching to the problem of compressed text indexing. An extended version of this transform, which maps 3D points into text, can be used to derive lower bounds for the position-restricted pattern matching problem.

For upper bounds, Chien et al. introduced the *geometric Burrows-Wheeler transform* (GBWT) to convert pattern matching problems into range queries. Given a text T and blocking factor d, let $T'[1 \ldots n/d]$ be the text formed by blocking every consecutive d characters of T to form a single metacharacter, as shown in Figure 1. Let $SA'[1 \ldots n/d]$ be the sparse suffix array of T'. The GBWT of T consists of the 2D points (i, c_i), for $1 \leq i \leq n/d$, where c_i is the reverse of the metacharacter that precedes $T'[SA'[i]]$. The parameter d is set to $\frac{1}{2} \log_\sigma n$ so that the data structures require only $O(n \log \sigma)$ bits.

To perform a pattern matching query for pattern P, we find, for each possible offset k between 0 and $d-1$, all occurrences of P that start k characters from the beginning of a metacharacter. For $k \neq 0$, this process partitions P into $\langle \widehat{P}, \widetilde{P} \rangle$, where $\widetilde{P}$ matches a prefix of a suffix of T', and $\widehat{P}$ has length k and matches a suffix of the preceding metacharacter. By reversing $\widehat{P}$, both subcomponents must match prefixes, which corresponds to a 2D range query on the set S of 2D points defined above. The range of indices in the sparse suffix array SA' can be found by a string B-tree, and the 2D search can be done using a wavelet tree or using alternative indexes, such as kd-trees or R-trees.

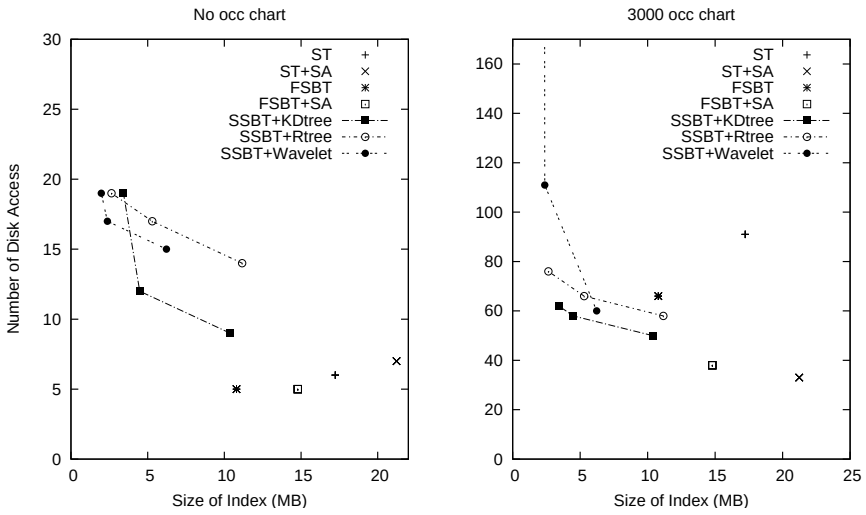

Fig. 2. I/Os per query. On the left, there is no output (i.e., the searches are unsuccessful). On the right, there are 3,000 occurrences on average per query.

If the pattern is small and fits entirely within a metacharacter, table lookup techniques (akin to inverted indexes) provide the desired answer using a negligible amount of space. The resulting space bound for GBWT is $O(n \log \sigma)$ bits, and the I/O query bound is the same as for four-sided 2D range search, namely, $O(p/B + (\log_\sigma n) \log_B n + occ \log_B n)$ or $O(p/B + \sqrt{n/B} \log_\sigma n + occ/B)$ [34]. Faster performance can often be achieved in practice using kd-trees or R-trees [10].

Hon et al. [34] introduce a variable-length sparsification so that each metacharacter corresponds to roughly d bits in *compressed* form. Assuming $k = o(\log_\sigma n)$, this compression further reduces the space usage from linear to $O(nH_k(T)+n)+o(n \log \sigma)$ bits of blocked storage. The query time for reporting pattern matches is $O(p/(B \log_\sigma n) + (\log^4 n)/\log\log n + occ \log_B n)$ I/Os.

4.1 Experimental Results for GBWT

In Figure 2, we compare the pattern matching performance of several indexes:

1. `ST`: Suffix tree (with naive blocking) and a parenthesis encoding of subtrees.
2. `ST + SA`: Suffix tree (with naive blocking strategy) and the suffix array.
3. `FSBT:` Full version of string B-tree containing all suffixes. The structure of each blind trie uses parentheses encoding, saving ≈ 1.75 bytes per trie node.
4. `FSBT + SA`: Full version of string B-tree and the suffix array.
5. `SSBT(d) + Rtree`: Sparse version of the string B-tree with the R-tree 2D range search data structure. Metacharacter sizes are $d = 2, 4, 8$.
6. `SSBT(d) +` k`d-tree`: Sparse version of the string B-tree with the kd-tree range search data structure. Metacharacter sizes are $d = 2, 4, 8$.
7. `SSBT(d) + Wavelet`: Sparse version of the string B-tree with the wavelet tree used for 2D queries. Metacharacter sizes are $d = 2, 4, 8$.

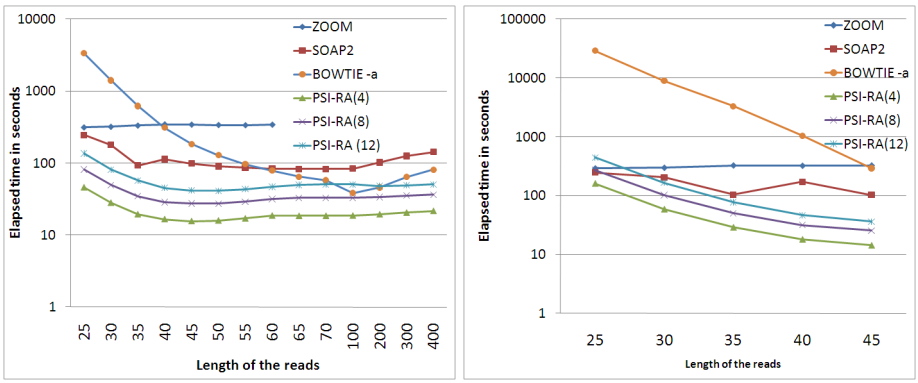

Fig. 3. Finding all exact matches of 1 million short read patterns P with the human genome. The left graph uses short read patterns sampled from the target genome; the right graph uses short read patterns obtained from the SRR001115 experiment [56].

The first four indexes are not compressed data structures and exhibit significant space bloat; however, they achieve relatively good I/O performance. The latter three use sparsification, which slows query performance but requires less space.

4.2 Parallel Sparse Index for Genome Read Alignments

In this section, we consider the special case in the internal memory setting in which P is a "short read" that we seek to align with a genome sequence, such as the human genome. The human genome consists of about 3 billion bases (A, T, C, or G) and occupies roughly 800MB of raw space. In some applications, the read sequence P may be on the order of 30 bases, while with newer equipment, the length of P may be more than 100. We can simplify our GBWT approach by explicitly checking, for each match of $\widetilde{P}$, whether $\widehat{P}$ also matches. We use some auxiliary data structures to quickly search the sparse suffix array SA' and employ a backtracking mechanism to find approximate matches. The reliability of each base in P typically degrades toward the end of P, and so our algorithm prioritizes mismatches toward the end of the sequence.

Figure 3 gives timings of short read aligners for a typical instance of the problem, in which all exact matches between each P and the genome are reported:

1. `SOAP2` [42]: Index size is 6.1 GB, based on 2way-BWT, run with parameters -r 1 -M 0 -v 0 (exact search).
2. `BOWTIE` [41]: Index size 2.9 GB, based upon BWT, run with -a option.
3. `ZOOM` [43]: No index, based on a multiple-spaced seed filtering technique, run with -mm 0 (exact search).
4. `Ψ-RA(4)`: Index size 3.4 GB, uses sparse suffix array with sparsification factor of $d = 4$ bases, finds all occurrences of the input patterns.
5. `Ψ-RA(8)`: Index size 2.0 GB, uses sparse suffix array with sparsification factor of $d = 8$ bases, finds all occurrences of the input patterns.
6. `Ψ-RA(12)`: Index size 1.6 GB, uses sparse suffix array with sparsification factor of $d = 12$ bases, finds all occurrences of the input patterns.

The size listed for each index includes the space for the original sequence data. Our simplified *parallel sparse index read aligner* (a.k.a. Ψ-RA) [39] achieves relatively high throughput compared with other methods. The experiments were performed on an Intel i7 with eight cores and 8GB memory. The Ψ-RA method can take advantage of multicore processors, since each of the d offset searches can be trivially parallelized. However, for fairness in comparisons, the timings used a single-threaded implementation and did not utilize multiple cores.

5 Top-k Queries for Relevance

Inverted indexes have several advantages over compressed data structures that need to be considered: (1) Inverted indexes are highly space-efficient, and they naturally provide the demarcation between RAM storage (dictionary of words) and disk storage (document lists for the words). (2) They are easy to construct in external memory. (3) They can be dynamically updated and also allow distributed operations [74]. (4) They can be easily tuned (by using frequency-ordered or PageRank-ordered lists) to retrieve top-k most relevant answers to the query, which is often required in search engines like Google.

Top-k query processing is an emerging field in databases [53, 6, 64, 69, 36]. When there are too many query results, certain notions of relevance may make some answers preferable to others. Database users typically want to see those answers first. In the problem of top-k document retrieval, the input data consist of D documents $\{d_1, d_2, \ldots, d_D\}$ of total length n. Given a query pattern P, the goal is to list which documents contain P; there is no need to report where in a document the matches occur. If a relevance measure is supplied (such as frequency of matches, proximity of matches, or PageRank), the goal is to output only the most relevant matching documents. The problem could specify an absolute threshold K on the relevance, in which case all matching documents are reported whose relevance value is $\geq K$; alternatively, given parameter k, the top-k most relevant documents are reported.

Early approaches to the problem did not consider relevance and instead reported all matches [53, 64, 69]. They used a generalized suffix tree, and for each leaf, they record which document it belongs to. On top of this basic data structure, early approaches employed either a chaining method to link together entries from the same document or else a wavelet tree built over the document array. As a result, these data structures exhibit significant bloat in terms of space usage.

Hon et al. [36] employ a more space-conscious approach. They use a suffix tree, and every node of the suffix tree is augmented with additional arrays. A relevance queries can be seen as a $(2, 1, 1)$-query in 3D, where the two x-constraints come from specifying the subtree that matches the pattern P, the one-sided y-constraint is for preventing redundant output of the same document, and the one-sided z-constraint is to get the highest relevance scores. This $(2, 1, 1)$-query in 3D can be converted to at most p $(2, 1)$-queries in 2D, which in turn can be answered quickly using range-maximum query structures, thus achieving space-time optimal results. The result was the first $O(n)$-word index that takes $O(p + k \log k)$ time to answer top-k queries.

Preliminary experimental results show that for 2MB of input data, the index size is 30MB and can answer top-k queries in about 4×10^{-4} seconds (for $k = 10$). This implementation represents a major improvement because previous solutions, such as an adaptation of [53], take about 500MB of index size and are not as query-efficient. Further improvements are being explored.

Many challenging problems remain. One is to make the data structures compressed. The space usage is $\Omega(n)$ nodes, and thus $\Omega(n \log n)$ bits, which is larger than the input data. To reduce the space usage to that of a compressed representation, Hon et al. [36] employ sparsification to selectively augment only $O(n/\log^2 n)$ carefully chosen nodes of the suffix tree with additional information, achieving high-order compression, at the expense of slower search times. Other challenges include improved bounds and allowing approximate matching and approximate relevance. Thankachan et al. [67] develop top-k data structures for searching two patterns using $O(n)$ words of space with times related to 2D range search; the approach can be generalized for multipattern queries.

6 Conclusions

We discussed recent trends in compressed data structures for text and document indexing, with the goal of achieving the time and space efficiency of inverted indexes, but with greater functionality. We focused on two important challenging issues: I/O efficiency in external memory settings and building relevance into the query mechanism. Sparsification can help address both questions, and it can also be applied to the dual problem of dictionary matching, where the set of patterns is given and the query is the text [32, 33, 66, 5]. Much work remains to be done, including addressing issues of parallel multicore optimization, dynamic updates, online data streaming, and approximate matching.

References

1. A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
2. D. Arroyuelo and G. Navarro. A Lempel-Ziv text index on secondary storage. In *Proc. Symp. on Combinatorial Pattern Matching*, volume 4580 of *Lecture Notes in Computer Science*, pages 83–94. Springer, 2007.
3. J. Barbay, M. He, J. I. Munro, and S. S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 680–689, 2007.
4. R. Bayer and K. Unterauer. Prefix B-trees. *ACM Transactions on Database Systems*, 2(1):11–26, March 1977.
5. D. Belazzougui. Succinct dictionary matching with no slowdown. In *Proc. Symp. on Combinatorial Pattern Matching*, June 2010.
6. Bialynicka-Birula and R. Grossi. Rank-sensitive data structures. In *Proc. Intl. Symp. on String Processing Information Retrieval*, volume 12 of *LNCS*, 2005.
7. M. Burrows and D. Wheeler. A block sorting data compression algorithm. Technical report, Digital Systems Research Center, 1994.
8. H. L. Chan, W. K. Hon, T. W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2), 2007.

9. Y.-F. Chien, W.-K. Hon, R. Shah, and J. S. Vitter. Geometric Burrows-Wheeler transform: Linking range searching and text indexing. In *Proc. IEEE Data Compression Conf.*, pages 252–261, 2008.
10. S.-Y. Chiu, W.-K. Hon, R. Shah, and J. S. Vitter. I/o-efficient compressed text indexes: From theory to practice. In *Proc. IEEE Data Compression Conf.*, pages 426–434, 2010.
11. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. Symp. on Operating Systems Design and Implementation*, pages 137–150. USENIX, December 2004.
12. P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21:194–203, 1975.
13. P. Ferragina, R. Giancarlo, and G. Manzini. The myriad virtues of wavelet trees. *Information and Computation*, 207(8):849–866, 2009.
14. P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52(4):688–713, July 2005.
15. P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13:article 1.12, 2008.
16. 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*, 46(2):236–280, March 1999.
17. P. Ferragina, R. Grossi, A. Gupta, R. Shah, and J. S. Vitter. On searching compressed string collections cache-obliviously. In *Proc. ACM Conf. on Principles of Database Systems*, pages 181–190, Vancouver, June 2008.
18. P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 184–196, 2005.
19. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. IEEE Symp. on Foundations of Computer Science*, volume 41, pages 390–398, November 2000.
20. P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.
21. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2), May 2007. Conference version in *SPIRE 2004*.
22. P. Ferragina and R. Venturini. Compressed permuterm index. In *Proc. ACM SIGIR Conf. on Res. and Dev. in Information Retrieval*, pages 535–542, 2007.
23. J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.
24. L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: Experiments on suffix arrays and trees. *ACM Transactions on Algorithms*, 2(4):611–639, 2006. Conference versions in *SODA 2004* and *DCC 2004*.
25. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. IEEE Symp. on Foundations of Computer Science*, volume 40, pages 285–298, 1999.
26. G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In *Information Retrieval: Data Structures And Algorithms*, chapter 5, pages 66–82. Prentice-Hall, 1992.
27. R. González and G. Navarro. A compressed text index on secondary memory. In *Proc. Intl. Work. Combinatorial Algorithms*, pages 80–91, Newcastle, Australia, 2007. College Publications.
28. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, January 2003.
29. R. Grossi, A. Gupta, and J. S. Vitter. Nearly tight bounds on the encoding length of the Burrows-Wheeler transform. In *Proc. Work. on Analytical Algorithmics and Combinatorics*, January 2008.

30. R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. ACM Symp. on Theory of Computing*, volume 32, pages 397–406, May 2000.
31. 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(32):378–407, 2005.
32. W.-K. Hon, T.-W. Lam, R. Shah, S.-L. Tam, and J. S. Vitter. Compressed index for dictionary matching. In *Proc. IEEE Data Compression Conf.*, pages 23–32, 2008.
33. W.-K. Hon, T.-W. Lam, R. Shah, S.-L. Tam, and J. S. Vitter. Succinct index for dynamic dictionary matching. In *Proc. Intl. Symp. on Algorithms and Computation*, LNCS. Springer, December 2009.
34. W.-K. Hon, R. Shah, S. V. Thankachan, and J. S. Vitter. On entropy-compressed text indexing in external memory. In *Proc. Intl. Symp. on String Processing Information Retrieval*, volume 5721 of *LNCS*. Springer, August 2009.
35. W.-K. Hon, R. Shah, and J. S. Vitter. Ordered pattern matching: Towards full-text retrieval. In *Purdue University Tech Rept*, 2006.
36. W.-K. Hon, R. Shah, and J. S. Vitter. Space-efficient framework for top-k string retrieval problems. In *Proc. IEEE Symp. on Foundations of Computer Science*, Atlanta, October 2009.
37. J. Kärkkäinen. *Repetition-Based Text Indexes*. Ph.d., University of Helsinki, 1999.
38. J. Kärkkäinen and S. S. Rao. Full-text indexes in external memory. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies*, chapter 7, pages 149–170. Springer, Berlin, Germany, 2003.
39. M. O. Külekci, W.-K. Hon, R. Shah, J. S. Vitter, and B. Xu. A parallel sparse index for read alignment on genomes, 2010.
40. T.-W. Lam, W.-K. Sung, and S.-S. Wong. Improved approximate string matching using compressed suffix data structures. *Algorithmica*, 51(3):298–314, 2008.
41. B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):article R25, 2009.
42. R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang. SOAP2: An improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.
43. H. Lin, Z. Zhang, M. Q. Zhang, B. Ma, and M. Li. ZOOM: Zillions of oligos mapped. *Bioinformatics*, 24(21):2431–2437, 2008.
44. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.
45. V. Mäkinen and G. Navarro. Position-restricted substring searching. In *Proc. Latin American Theoretical Informatics Symp.*, pages 703–714, 2006.
46. V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. Intl. Symp. on String Processing Information Retrieval*, volume 4726 of *LNCS*, pages 229–241. Springer, October 2007.
47. V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3):article 12, June 2008.
48. V. Mäkinen, G. Navarro, and K. Sadakane. Advantages of backward searching—efficient secondary memory and distributed implementation of compressed suffix arrays. In *Proc. Intl. Symp. on Algorithms and Computation*, volume 3341 of *LNCS*, pages 681–692. Springer, 2004.
49. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.
50. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3), 2001. Conference version in *SODA 1999*.
51. E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

52. A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, October 1996.
53. S. Muthukrishnan. Efficient Algorithms for Document Retrieval Problems. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 657–666, 2002.
54. S. Muthukrishnan. *Data Streams: Algorithms and Applications.* Foundations and Trends in Theoretical Computer Science. now Publishers, Hanover, MA, 2005.
55. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
56. NCBI short read archive SRR001115, `http://www.ncbi.nlm.nih.gov/`.
57. M. Patrascu. Succincter. In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 305–313, 2008.
58. S. J. Puglisi, W. F. Smyth, and A. Turpin. Inverted files versus suffix arrays for locating patterns in primary memory. In *Proc. Intl. Symp. on String Processing Information Retrieval*, volume 4209 of *LNCS*, pages 122–133. Springer, 2006.
59. R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):article 43, 2007.
60. L. Russo, G. Navarro, and A. Oliveira. Fully-compressed suffix trees. In *Proc. Latin American Theoretical Informatics Symp.*, volume 4957 of *LNCS*, pages 362–373, 2008.
61. K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. Intl. Symp. on Algorithms and Computation*, number 1969 in LNCS, pages 410–421. Springer, December 2000.
62. K. Sadakane. New text indexing functiionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
63. K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.
64. K. Sadakane. Succinct Data Structures for Flexible Text Retrieval Systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.
65. A. C. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh. Parallelism via multithreaded and multicore CPUs. *IEEE Computer*, 43(3):24–32, March 2010.
66. A. Tam, E. Wu, T. W. Lam, and S.-M. Yiu. Succinct text indexing with wildcards. In *Proc. Intl. Symp. on String Processing Information Retrieval*, pages 39–50, August 2009.
67. S. V. Thankachan, W.-K. Hon, R. Shah, and J. S. Vitter. String retrieval for multi-pattern queries, 2010.
68. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, September 1995.
69. N. Välimäki and V. Mäkinen. Space-Efficient Algorithms for Document Retrieval. In *Proc. Symp. on Combinatorial Pattern Matching*, pages 205–215, 2007.
70. J. S. Vitter. *Algorithms and Data Structures for External Memory.* Foundations and Trends in Theoretical Computer Science. now Publishers, Hanover, MA, 2008.
71. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.
72. P. Weiner. Linear pattern matching algorithm. In *Proc. IEEE Symp. on Switching and Automata Theory*, volume 14, pages 1–11, Washington, DC, 1973.
73. I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann, Los Altos, CA, 2nd edition, 1999.
74. J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), 2006.