Indexing Compressed Text

PAOLO FERRAGINA, ROSSANO VENTURINI
University of Pisa, Pisa, Italy

Synonyms

Compressed full-text indexing; Compressed suffix array; Compressed suffix tree; Compressed and searchable data format

Definition

Given a text $T[1,n]$, the *Compressed Text Indexing* problem requires to building an indexing data structure over T that takes space close to the empirical entropy of the input text and answers queries on the occurrences of an arbitrary pattern $P[1,p]$ in T without any significant slowdown with respect to uncompressed indexes. There are three main queries: `count`(P), that returns the number of pattern occurrences in T, `locate`(P), that returns the starting positions of all pattern occurrences in T, and `extract`(i, j), that retrieves the substring $T[i, j]$.

Historical Background

String processing and searching tasks are at the core of modern web search, information retrieval (IR), data base and data mining applications. Most of text manipulations required by these applications involve, sooner or later, *searching* those (long) texts for (short) patterns or *accessing* portions of those texts for subsequent processing/mining tasks. Despite the increase in processing speed of current CPUs and memories/disks, sequential text searching long ago ceased to be a viable approach, and indexed text searching has become mandatory.

A *(full-)text index* is a data structure built over a text $T[1,n]$, drawn from an alphabet Σ of size σ, which significantly speeds up sequential searches for *arbitrary* pattern strings, at the cost of some additional space. *Suffix trees* and *suffix arrays* are the most well-known full-text indexes [5]. The *suffix tree* of a text T is a trie (or digital tree) built on all the n text suffixes $T[i, n]$, where unary paths are compacted to ensure $O(n)$ overall size. The suffix tree has n leaves, one per text suffix, and each internal node corresponds to a unique substring of T that occurs more than once. The suffix tree can count the *occ* occurrences of any pattern $P[1,p]$ in time $O(p)$ by descending in the suffix tree according to the symbols of P, and it can locate these occurrences in optimal $O(occ)$ time by traversing the subtree of the node reached by counting. The suffix tree, however, uses much more space than the text itself because it requires $\Theta(n \log n)$ bits, whereas the text needs $n\lceil \log \sigma \rceil$ bits (logarithms are in base 2). In practice, a suffix tree requires from 10 to 20 times the text size, if carefully engineered [5].

The *suffix array* is a compact version of the suffix tree, obtained by storing in $SA[1,n]$ the starting positions of the suffixes of T listed in lexicographical order. This data structure still requires $\Theta(n \log n)$ bits in the worst case, but the constant hidden in the big-Oh notation is small in practice, namely it is no more than 4. SA can be obtained by traversing the leaves of the suffix tree, or it can be built directly in optimal linear time via ad-hoc sorting methods [5]. Since any substring of T is the prefix of a text suffix, finding all pattern occurrences boils down to finding all text suffixes that start with P. These suffixes form a lexicographic interval in SA that can be binary searched in $O(p \log n)$ time, as each comparison in the binary search requires examining up to p symbols of the pattern and of a text suffix. The time complexity can be improved to $O(p + \log n)$ by using an auxiliary data structure that doubles the space requirement of the suffix array, or it can be further reduced to $O(p + \log \sigma)$ by a proper sampling of the indexed suffixes (cfr. Suffix Trays). Once the interval $SA[sp,ep]$ containing all text suffixes starting with P has been identified, `count`(P) is answered by returning the value $occ = ep - sp + 1$, and `locate`(P) is answered by retrieving the entries $SA[sp]$, $SA[sp + 1]$,...,$SA[ep]$.

The use of full-text indexes is not limited to (full-) text searching over one single text. It can be easily extended to multiple texts, and can also be used to support (prefix, suffix, or substring) queries over a dictionary $\mathcal{D}$ of strings having variable length. This problem is called *Dictionary Indexing* and occurs frequently in the implementation of IR and data mining applications. It can be solved via a (compressed) index built on a string S_D which is obtained by concatenating all dictionary strings, separated with a special symbol #. A prefix search for P in $\mathcal{D}$ can be implemented by counting/locating the query pattern $\#P$ in S_D; a suffix search can be implemented by searching for $P\#$ in S_D; substring searches are directly executed on S_D.

Foundations

The large space occupancy of full-text indexes has driven programmers to resort to inverted indexes to solve their searching operations on large textual datasets, and some researchers have actually concluded that the increased query power of full-text indexes has to be paid by additional storage space. Fortunately, a recent body of research showed that *compressed* full-text indexes can be designed by deploying algorithmic techniques and mathematical tools which lie at the crossing point of three distinct fields – data compression, algorithmics and databases (see e.g., [9,13,14]). Most of these indexes can be classified into two families – FM-indexes (FMI) and Compressed Suffix Arrays (CSA) – and achieve efficient query times and space close to the one achievable by the best known compressors, like gzip or bzip2. In theory, these indexes require $O(nH_k(T)) + o(n \log \sigma)$ bits of space, where $H_k(T)$ is the kth order empirical entropy of T (see Table 1). This bound is appealing because it can be sublinear in n, for highly compressible texts, and $nH_k(T)$ is the classic Information-Theoretic lower bound to the storage complexity of T by means of any kth order compressor, like gzip and bzip2 (recall that `extract`$(1,n) = T$).

The FM-Index Family

These compressed indexes were introduced by Ferragina and Manzini in [9], who devised a way to orchestrate in efficient time and space the relation that exists between the suffix array data structure and the *Burrows-Wheeler Transform* (shortly, BWT [4]). The BWT is a reversible transformation that permutes the symbols of the input string T into a new string `bwt`(T) which is easier to compress, and can be computed in three steps (see Fig. 1):

1. Append at the end of T a special symbol \$ smaller than any other symbol of Σ;
2. Form a *conceptual* matrix $\mathcal{M}(T)$ whose rows are the cyclic rotations of string $T\$$ in lexicographic order;
3. Set string `bwt`(T) to the last column L of the sorted matrix $\mathcal{M}(T)$.

Every column of $\mathcal{M}(T)$, hence also the transformed string L, is a permutation of $T\$$. In particular the first column of $\mathcal{M}(T)$, call it F, is obtained by lexicographically sorting the symbols of $T\$$ (or, equivalently, the symbols of L). Note that the sorting of the rows of

Indexing Compressed Text. Table 1. Best known complexities for the time (in big-Oh) and space (in bits) required by the main families of compressed full-text indexes

Index	Count	Locate	Extract	Space	References
FMI	p	$occ \cdot \mathtt{polylog}(n)$	$\ell + \mathtt{polylog}(n)$	$nH_k(T) + o(n)$	[8]
CSA	$p/\log_\sigma n + \mathtt{polylog}(n)$	$occ \cdot \mathtt{polylog}(n)$	$\ell/\log_\sigma n + \mathtt{polylog}(n)$	$\gamma^{-1} nH_k(T) + o(n)$	[11]
LZ-INDEX	$p^2 \log p + p \log n + occ$	$occ \cdot \log n$	$\ell(1 + \epsilon^{-1}/\log_\sigma \ell)$	$(2 + \epsilon)nH_k(T) + o(n \log \sigma)$	[1]

Here $\epsilon > 0$ and $0 < \gamma < \frac{1}{3}$ are constants fixed in advance before the data structures are built; ℓ is the number of text symbols to be retrieved by `extract`; $H_k(T)$ is the k-th order empirical entropy of text T [14]. The reported complexities are worst-case and hold for $\sigma = O(\mathtt{polylog}(n))$ assuming that $k \leq \alpha \log_\sigma n$ with $0 < \alpha < 1$ except for LZ-INDEX in which $k = o(\log_\sigma n)$. For more precise bounds (e.g., coefficients in $\mathtt{polylog}(n)$ terms and the case $\sigma = \Omega(\mathtt{polylog}(n))$ and for a thoughtful comparison of these indexes and their numerous variants, the reader is referred to [14].

$\mathcal{M}(T)$ is essentially equal to the sorting of the suffixes of T, because of the presence of the special symbol \$. This shows that: (i) symbols preceding the same substring (*context*) in T are grouped together in L, and thus give raise to clusters of nearly identical symbols; (ii) there is an obvious relation between $\mathcal{M}(T)$ and SA. Property (1) is the key for devising modern data compressors, Property (2) is crucial for designing compressed indexes and, additionally, suggests a way to compute the BWT through the construction of the suffix array of T: $L[0] = T[n]$ and, for any $1 \leq i \leq n$, set $L[i] = T[SA[i] - 1]$.

Burrows and Wheeler [4] devised two properties for the invertibility of the BWT:

1. Since the rows in $\mathcal{M}(T)$ are cyclically rotated, $L[i]$ *precedes* $F[i]$ in the original string T.
2. For any $c \in \Sigma$, the ℓth occurrence of c in F and the ℓth occurrence of c in L correspond to the *same* symbol of the string T.

As a result, the original text T can be obtained backwards from L by resorting to a function LF that maps row indexes to row indexes, and is defined as follows: if the BWT maps $T[j-1]$ to $L[i']$ and $T[j]$ to $L[i]$, then $LF(i) = i'$ (so LF implements a sort of *backward* step over T) [9]. Now, since the first row of $\mathcal{M}(T)$ is \$T, it can be stated that $T[n] = L[0]$ and, in general, $T[n-i] = L[LF^i(0)]$, for $i = 1,\ldots,n-1$.

Starting from these basic properties, Ferragina and Manzini [9] proposed a way to combine the compressibility of the BWT with the indexing power of the suffix array. In particular, they have shown that searching operations on T can be reduced to counting queries of *single* symbols in L, now called `rank` operations. For any symbol $c \in \Sigma$ and position i in L, the query

		F		L
mississippi\$		\$	mississipp	i
ississippi\$m		i	\$mississip	p
ssissippi\$mi		i	ppi\$missis	s
sissippi\$mis		i	ssippi\$mis	s
issippi\$miss		i	ssissippi\$	m
ssippi\$missi	$\Longrightarrow$	m	ississippi	\$
sippi\$missis		p	i\$mississi	p
ippi\$mississ		p	pi\$mississ	i
ppi\$mississi		s	ippi\$missi	s
pi\$mississip		s	issippi\$mi	s
i\$mississipp		s	sippi\$miss	i
\$mississippi		s	sissippi\$m	i

Indexing Compressed Text. Figure 1. Example of Burrows-Wheeler transform for $T = \mathtt{mississippi}$. The matrix on the right has the rows sorted in lexicographic order. The output of the BWT is the column $L = \mathtt{ipssm\$pissii}$.

$\mathtt{rank}_c(L, i)$ returns how many times the symbol c appears in $L[1,i]$. An FM-index then consists of three key tools: a compressed representation of $\mathtt{bwt}(T)$ that supports efficient `rank` queries, a small array $C[c]$ which tells how many symbols smaller than c appear in T (this takes $O(\sigma \log n)$ bits), and the so called *backward search* algorithm which carefully orchestrates the former two data structures in order to implement efficiently the `count` query. More precisely, FMI searches the pattern $P[1,p]$ backwards in p steps, which eventually identify the interval of text suffixes that are prefixed by P or, equivalently, the interval of rows of $\mathcal{M}(T)$ that are prefixed by P. This is done by maintaining, inductively for $i = p, p-1,\ldots,1$, the interval $SA[sp_i, ep_i]$ that stores all text suffixes that are prefixed by the pattern suffix $P[i, p]$. At the beginning it is $i = p$, and so $SA[sp_p, ep_p]$ corresponds to all suffixes which are prefixed by the last symbol $P[p]$: hence, it is enough to set $sp_p = C[P[p]] + 1$ and $ep_p = C[P[p] + 1]$. At any other step, the algorithm has inductively computed $SA[sp_{i+1}, ep_{i+1}]$, and thus it can derive the next interval of suffixes prefixed by $P[i, m]$ by setting $sp_i = C[P[i]] + \mathtt{rank}_{P[i]}(L, sp_{i+1} - 1) + 1$ and $ep_i = C[P[i]] + \mathtt{rank}_{P[i]}(L, ep_{i+1})$. These two computations are actually mapping (via *LF*) the first and last occurrences (if any) of symbol $P[i]$ in the substring $L[sp_{i+1}, ep_{i+1}]$ to their corresponding occurrences in *F*. (Indeed, [9] showed that any *LF* computation boils down to a `rank` query on *L*.) As a result, the backward-search algorithm requires to solve $2p$ rank queries on $L = \mathtt{bwt}(T)$ in order to find out the (possibly empty) range $SA[sp, ep]$ of text suffixes prefixed by *P*. `count`(P) can be then solved by returning the value $occ = ep_1 - sp_1 + 1$.

Conversely, `locate` and `extract` need some extra information about the underlying suffix array, this impacts onto the space occupancy of the FMI. Recall that `locate`(P) requires to return, for any $i \in [sp, ep]$, the position $pos(i) = SA[i]$. For space reasons *SA* cannot be stored explicitly so that, for a fixed parameter $\mu = \lceil \log^{1+\in} n \rceil$, FMI samples the rows of $\mathcal{M}(T)$ which correspond to text suffixes that start at positions of the form $1 + j \cdot \mu$. Each such pair $\langle$row, position$\rangle$ is stored explicitly in a data structure $\mathcal{S}$ that supports membership queries in constant time (on the *row*-component). Now, given a row index *i*, the value $pos(i)$ can be derived immediately from $\mathcal{S}$, if *i* is a sampled row; otherwise, the algorithm computes $j = LF^t(i)$, for $t = 1,2,...,$ until *j* is a sampled row. In this case, $pos(i) = pos(j) + t$. The sampling strategy ensures that a row in $\mathcal{S}$ is found in at most μ iterations, and thus the *occ* occurrences of the pattern *P* can be located via $O(\mu \cdot occ)$ `rank` queries. The algorithm for `extract`(i,i') requires a similar approach and takes no more than $(i' - i + \mu + 1)$ rank queries.

The net result is that the space and time complexities of FMI depend on the value μ and on the performance guaranteed by the data structure used to compute rank queries on the BWT-string. The extra space required by the data structures added to support `locate` and `extract` is bounded by $O((n \log n)/\mu)$ bits, which is $o(n)$ whenever $\in > 0$. The real challenge thus consists of representing $\mathtt{bwt}(T)$ in a compressed form and answering efficiently rank queries over it. Actually, all implementations of FMI differentiate themselves by the strategy used to solve this problem, as the alphabet size grows. Today, the literature offers many solutions, the most efficient ones are summarized below.

Lemma 1 *Let* $T[1,n]$ *be a string over an alphabet of size* σ*, and let* $L = \mathtt{bwt}(T)$.

1. *For* $\sigma = O(\mathtt{polylog}(n))$*, there exists a data structure which supports* `rank` *queries on L in* $O(1)$ *time using* $nH_k(T) + o(n)$ *bits of space, for any* $k \leq \alpha \log_\sigma n$ *and* $0 < \alpha < 1$*, and retrieves any symbol of L in the same time bound* [10, *Theorem 5*].
2. *For general* Σ*, there exists a data structure which supports* `rank` *queries on L in* $O(\log \log \sigma)$ *time, using* $nH_k(T) + n\ o(\log \sigma)$ *bits of space, for any* $k \leq \alpha \log_\sigma n$ *and* $0 < \alpha < 1$*, and retrieves any symbol of L in the same time bound* [2, *Theorem 4.2*].

By plugging this Lemma into the FMI data structure, one derives a compressed full-text index that supports efficiently the three full-text queries – namely, `count`, `locate`, `extract`– and occupies space approaching the *k*th order empirical entropy of *T* (see Table 1).

In practice, there are various implementations of FMI, whose engineering choices mainly refer to the way the `rank`-data structure built on $\mathtt{bwt}(T)$ is compressed and scales with the alphabet size of the indexed text. The site Pizza&Chili (see below) reports several implementations for FMI that mainly boil down to the following trick: $\mathtt{bwt}(T)$ is split into blocks (of equal or variable length) and values of $\mathtt{rank}_c$ are precomputed for all block beginnings and all symbols $c \in \Sigma$. A query $\mathtt{rank}_c(L, i)$ is answered by summing up the answer available for the beginning of the block that contains $L[i]$ plus the rest of the occurrences of *c* in that block – they are obtained either by sequentially decompressing the block or by using a proper compressed data structure built on it (e.g., the Wavelet Tree of [12]). The former approach favors compression, the latter favors query speed.

The CSA family. These compressed indexes were introduced by Grossi and Vitter [13], who showed how to compactly represent the suffix array *SA* in $O(n \log \sigma)$ bits and still be able to access any of its entries in efficient time. Their solution is based on a function Ψ, which is the inverse of the function LF introduced for BWT:

$$\Psi(i) = \begin{cases} i' \text{ such that } SA[i'] = SA[i] + 1 & (\text{if } SA[i] > n) \\ i' \text{ such that } SA[i'] = 1 & (\text{if } SA[i] = n) \end{cases}$$

In other words, $\Psi(i)$ refers to the position in the suffix array of the text suffix that follows $SA[i]$ in T, namely, the text suffix which is one-symbol shorter. The compact storage of SA proposed by Grossi and Vitter is based on a hierarchical decomposition that deploys Ψ. To represent $SA_0 = SA$ they use three vectors: B_0, Ψ_0 and SA_1. The binary vector $B_0[1,n]$ marks the entries of SA_0 which are even (suffixes). The vector $\Psi_0[1,\lceil n/2 \rceil]$ stores the values Ψ(i) for which $SA[i]$ is odd (hence $B_0[i] = 0$). The vector $SA_1[1,\lceil n/2 \rfloor]$ is a "halved" version of SA_0, in that it contains the even elements of SA_0 divided by 2. Surprisingly enough, these three vectors suffice to retrieve any entry SA[i]. Of course, it is easy to determine whether $SA[i]$ is even or odd by simply looking at $B_0[i]$. If $SA[i]$ is odd, the following suffix $SA[i] + 1 = SA[\Psi(i)]$ is even, and its suffix-array position can be determined as $\Psi(i) = \Psi_0(\mathtt{rank}_0(B_0,i))$. If $SA[i]$ is even, it is enough to look at its *halved* value stored at $SA_1[\mathtt{rank}_1(B_0,i)]$. The three vectors Ψ_0, B_0 and SA_1 form the first level of the hierarchical decomposition of SA. This idea is applied recursively on SA_1 which is replaced by three other vectors: Ψ_1, B_1 and SA_2. This goes on until SA_h can be represented within O(n) bits, namely when $h = \lceil \log \log n \rceil$. Accessing $SA[i]$ takes h time. By storing the text T, in additional $n\lceil \log \sigma \rceil$ bits, one can search for a pattern P via the classic binary-search, now on the compacted SA. Grossi and Vitter proposed to store vectors B in compressed form via proper `rank`-data structures (see [14] and references therein), and deployed the *piecewise increasing property* for Ψ – namely, if $T[SA[i]] = T[SA[i + 1]]$, then $\Psi(i) < \Psi(i + 1)$ – to store each level of Ψ within $\frac{1}{2} n \log \sigma$ bits, still preserving constant time lookup to any level of Ψ. Other time/space tradeoffs are possible by using different numbers of levels. Essentially, not all the levels are represented and the function Ψ is used to jump from one represented level to the next represented one.

Recently, CSA has been the subject of two main improvements. The first one, due to Sadakane [16], showed that the original text T can be replaced with a binary vector F such that $F[i] = 1$ iff the first symbol of the suffixes SA[i $-$ 1] and $SA[i]$ differs. Since the suffixes in SA are lexicographically sorted, one can determine the first symbol of any suffix in constant time by just executing *a* $rank_1$ query on F. This fact, combined with the retrieval of Ψ's values in constant time, allows comparing any suffix with the searched pattern $P[1,p]$ in time $O(p)$. Sadakane also provided an improved representation for Ψ achieving $nH_0(T)$ bits. Theoretically, the best variant of CSA is due to Grossi, Gupta and Vitter [12] who devised some further structural properties of Ψ that allow to come close to $nH_k(T)$ bits, still preserving the previous time complexities for all full-text queries (see Table 1). Practically, the best implementation of the CSA is the one proposed by Sadakane that actually does not use the hierarchical decomposition above, but orchestrates a compact representation of the function Ψ together with the backward search and the sampling strategy of the FMI family. This *hybrid* index is among the fastest compressed indexes to count and locate pattern occurrences over highly-compressible data.

Other Compressed Indexes

Previous families of compressed indexes based their search on the implicit or explicit availability of the suffix array data structure. Recent years have seen the design of several other approaches, the two most notable ones are the *LZ-index*, proposed by Navarro, and the *Compressed Suffix Tree*, devised by Sadakane and then improved by many other authors. The former index bases its design on the parsing of the text T via the LZ78-compression scheme, and then enriches its output by additional data structures that support efficient searches over the parsed phrases. By properly orchestrating LZ78-parsing with compressed dictionary data structures, [1] achieved interesting search and entropy-based space bounds which are not competitive theoretically with the ones obtained by FMI and CSA indexes (see Table 1) but are, nonetheless, fast in practice. As far as the compressed suffix-tree is concerned, it is worth noticing that the compression of this data structure is obtained by properly orchestrating succinct tree and succinct array encodings [15]. The total space is the one required by the CSA built on T plus no more than $6n + o(n)$ bits; all known suffix-tree operations are supported with a maximum slowdown of $O(\log n)$ time with respect to the uncompressed suffix tree.

Key Applications

Compressed full-text indexes might be used at the core of modern web search, IR, data base and data mining applications because, as Knuth observed in the Art of Computer Programming (vol. 3): "*space optimization is closely related to time optimization in a disk memory*". Data compression can not only squeeze the space overhead of an index, but also improve its speed, as remarked earlier. Several authors [3,5,11,17,18] have recently addressed these issues in various settings but, nonetheless, there is much more room for theoretical and practical improvements.

Future Directions

An open challenge concerning compressed indexes is to fasten their `locate` queries in order to achieve the optimal $O(occ)$ time bound. The best known result is due to Ferragina and Manzini [9]: each occurrence is located in constant time, and the index takes $O(nH_k(S)\log^{\epsilon} n) + o(n \log \sigma \log^{\epsilon} n)$ bits, where ϵ is any positive constant. This bound has the extra log-factor in front of the entropy term! Therefore, it is natural to ask: Is there a full-text index achieving $O(p + occ)$ query time and $O(nH_k(S)) + o(n \log \sigma)$ bits of space occupancy in the worst case? This result would be *provably better* than any known uncompressed full-text index.

Another interesting open problem consists of designing a compressed full-text index which is disk-aware or, better, memory-oblivious in that it scales optimally over all memory levels available in a modern PC. The above data structures are compressed, but their overall size may span many memory levels so that issues pertaining to proper *arrangement of data* and properly *structured algorithmic computations* come into play. The most attractive disk-aware index is the String B-tree [7]; whereas the best cache-oblivious index is the COSB-tree [3,8]. Unfortunately the former is uncompressed, whereas the latter uses a compression heuristic which does not guarantee entropy-bounds in the worst case. It would be therefore valuable, also in practice, to devise a compressed index that combines the I/O-efficiency of the (cache oblivious) String B-tree with the space efficiency of the compressed full-text indexes discussed in this entry. Some preliminary results have been devised in [8], but the ultimate goal has yet to be achieved.

Experimental Results

Site Pizza&Chili [6] provides a full experimental comparison among the major implementations of compressed indexes. The experiments mainly show that these indexes can compress a text within 40–80% of its original size, and support searches for 20,000–50,000 patterns of 20 chars each within a second, locate about 100,000 pattern occurrences per second, and decompress text symbols at a rate of about 1MB/s. The compressed indexes are from one (`count`) to three (`locate`) orders of magnitudes slower than what one can achieve with a plain suffix array, at the benefit of using up to 18 times less space. This slowdown is due to the fact that search operations in compressed indexes access the memory in a non-local way thus eliciting many cache/IO misses, with a consequent degradation of the overall time performance. Nonetheless compressed indexes achieve a (search/extract) throughput which is significant and may match the efficiency specifications of most software tools running on commodity PCs. Recently, Ferragina and Venturini [11] provided a comparison among classic and compressed indexes for the Dictionary Indexing Problem showing that, in this case, compressed indexes may be faster than classic IR approaches.

Data Sets

Calgary Corpus (http://links.uwaterloo.ca/calgary.corpus.html)
Canterbury Corpus (http://corpus.canterbury.ac.nz)
Pizza&Chili Corpus (http://pizzachili.di.unipi.it or http://pizzachili.dcc.uchile.cl) see also[18]

URL to Code

Site Pizza&Chili (http://pizzachili.di.unipi.it or http://pizzachili.dcc.uchile.cl) collects implementations of the major compressed text indexes, and various tools and datasets to test them.

Cross-references

- ▶ Managing Compressed Structured Text
- ▶ Suffix Trees
- ▶ Text Compression
- ▶ Text Index Compression
- ▶ Text Indexing & Retrieval
- ▶ Text Indexing Techniques
- ▶ Text Representation
- ▶ XML Compression

Recommended Reading

1. Arroyuelo D., Navarro G., and Sadakane K. Reducing the space requirement of LZ-index. In Proc. 17th Annual Symposium on Combinatorial Pattern Matching, 2006, pp. 319–330.
2. Barbay J., He M., Munro J.I., and Srinivasa Rao S. Succinct indexes for string, binary relations and multi-labeled trees. In

Proc. 18th Annual ACM -SIAM Symp. on Discrete Algorithms, 2007, pp. 680–689.

3. Bender M.A., Farach-Colton M., and Kuszmaul B.C. Cache-oblivious string B-trees. In Proc. 25th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 2006, pp. 233–242.
4. Burrows M. and Wheeler D. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
5. Ferragina P. String Search in External Memory: Data Structures and Algorithms, In Handbook of Computational Molecular Biology, Chapman & Hall, London, 2005.
6. Ferragina P., González R., Navarro G., and Venturini R. Compressed Text Indexes: From Theory to Practice, J. Exp. Algorithmics, 13:1.12–1.31, 2009.
7. Ferragina P. and Grossi R. The String B-tree: A new data structure for string search in external memory and its applications. J. ACM, 46(2):236–280, 1999.
8. Ferragina P., Grossi R., Gupta A., Shah R., and Vitter J.S. On searching compressed string collections cache-obliviously. In Proc. 27th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 2008, pp. 181–190.
9. Ferragina P. and Manzini G. Indexing compressed text. J. ACM, 52(4):552–581, 2005.
10. Ferragina P., Manzini G., Mäkinen V., and Navarro G. Compressed representations of sequences and full-text indexes. ACM Trans. Algorithms, 3(2), 2007.
11. Ferragina P. and Venturini R. Compressed permuterm index. In Proc. 33rd Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, 2007, pp. 535–542.
12. Grossi R., Gupta A., and Vitter J.S. High-order entropy-compressed text indexes. In Proc. 14th Annual ACM-SIAM Symp. on Discrete Algorithms, 2003, pp. 841–850.
13. Grossi R. and Vitter J.S. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput., 35(2):378–407, 2005.
14. Navarro G. and Mäkinen V. Compressed full-text indexes. ACM Comput. Surv., 39(1), 2007.
15. Sadakane K. Compressed suffix trees with full functionality. Theory Comput. Syst., 41(4):589–607, 2007.
16. Sadakane K. New text indexing functionalities of the compressed suffix arrays. J. Algorithms, 48(2):294–413, 2007.
17. Sadakane K. Succinct data structures for flexible text retrieval systems. J. Discrete Algorithms, 5(1):12–22, 2007.
18. Tam S.L., Wong C.K., Lam T.W., Sung W.K., and Yiu S.M. Compressed indexing and local alignment of DNA. Bioinformatics, 24(6):791–797, 2008.