A

Compressed Cache-Oblivious String B-tree

PAOLO FERRAGINA, Department of Computer Science, University of Pisa
ROSSANO VENTURINI, Department of Computer Science, University of Pisa

In this paper we study three variants of the well-known prefix-search problem for strings, and design solutions for the cache-oblivious model which improve the best known results. Among these contributions, we close (asymptotically) the classic problem which asks for the detection of the set of strings which share the longest common prefix with a queried pattern by providing an I/O-optimal solution which matches the space lower bound for tries up to a constant multiplicative factor of the form $(1 + \epsilon)$, for $\epsilon > 0$. Our solutions hinge upon a novel compressed storage scheme which adds the ability to decompress *prefixes* of the stored strings I/O-optimally to the elegant locality-preserving front coding (Bender *et al.* 2006) still preserving its space bounds.

CCS Concepts: •**Theory of computation** → **Data structures design and analysis;**

Additional Key Words and Phrases: Pattern matching, data compression, compressed index, indexing data structure, string dictionary

ACM Reference Format:
Paolo Ferragina and Rossano Venturini, 2016. Compressed Cache-Oblivious String B-tree. *ACM Trans. Algor.* V, N, Article A (January 2015), 17 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The Prefix Search Problem is probably the most well-known problem in data-structure design for strings. It asks for the preprocessing of a set $\mathcal{S}$ of K strings, with total length N, in such a way that, given a query-pattern P, (the range of) all the strings in $\mathcal{S}$ which have P as a prefix can be returned efficiently in time and space. This easy-to-formalize problem is the backbone of many other algorithmic applications, and it has recently received revamped interest because of its Web-search (e.g., auto-completion search) and Internet-based (e.g., IP-lookup) applications. In this paper we concentrate on the case of *binary* strings and state that, if strings are not binary and Σ is the alphabet of their symbols, then we can *binarize* them all by changing every symbol into its $(\log |\Sigma|)$-binary encoding and still make our algorithms and results hold.

In order to establish and contextualize our results, we need to survey the main achievements in this topic and highlight their missing algorithmic points. The first solution to the prefix-search problem dates back to Fredkin [1960], who introduced and deployed the notion of the (compacted) trie. The trie structure became famous in the 1980s-'90s due to its suffix-based version, known as the Suffix Tree, which dominated

This work was partially supported by the MIUR of Italy under project PRIN ARS Technomedia 2012, and the SoBigData EU Project.
A preliminary version of this paper was published in the Proceedings of the 21st Annual European Symposium on Algorithms (ESA) 2013 [Ferragina and Venturini 2013].
Authors' address: Paolo Ferragina and Rossano Venturini, Department of Computer Science, University of Pisa, Largo Bruno Pontecorvo 3, I-56127 Pisa, Italy.

DOI: http://dx.doi.org/10.1145/0000000.0000000

first the string-matching scene [Apostolico 1985] and then bioinformatics [Gusfield 1997].

Starting from the Oxford English Dictionary initiative [Frakes and Baeza-Yates 1992], and the subsequent advent of the Web, it became crucial to design tries able to manage large sets of strings. It was immediately clear that laying out a trie in a disk memory with page size B words, each of $\Theta(\log N)$ bits, which supports I/O-efficient path traversals was not an easy task. After a decade, Demaine et al. [2004] showed that any layout of an arbitrary tree (and thus a trie) in a disk memory needs a large number of I/Os to traverse those downward paths.

The turning point in disk-efficient prefix search was the design of the string B-tree data structure [Ferragina and Grossi 1999] that was able to achieve $O(\log_B K + \mathsf{Scan}(P))$ I/Os, where $\mathsf{Scan}(P) = O(1 + \frac{|P|}{B \cdot \log N})$ indicates the number of I/Os needed to examine the input pattern of $|P|$ bits. String B-trees provide I/O-efficient analogues of tries and suffix trees, with the specialty of introducing some *redundancy* in the representation of the classic trie, thus allowing the author to get around the lower bounds by Demaine et al. [2004]. Also, the I/O-bound is optimal whenever the alphabet size is large and the data structure is required to support the search for the lexicographic position of P among the strings $\mathcal{S}$. The space usage is $O(K \log N + N)$ bits, which is uncompressed, and this takes into account the explicit storage of the (binary) strings and their pointers. The string B-tree is based upon a careful organization of a B-tree layout of string pointers, plus the use of one Patricia Trie [Morrison 1968] in each B-tree node which organizes its strings (pointers) in optimal space and supports prefix searches in $O(1)$ string accesses. In addition, the string B-tree is *dynamic* in that it allows the efficient insertion/deletion of individual strings from $\mathcal{S}$. As for B-trees, the data structure needs to know B in advance.

Brodal and Fagerberg [2006] made one step further by removing the dependance on B, and thus designed a static trie-like data structure in the *cache-oblivious* model [Frigo et al. 2012]. This structure is basically a trie over the indexed strings *plus* a few paths which are *replicated multiple times*. This *redundancy* is the essential feature to get around the lower bounds by Demaine et al. [2004], and it basically comes at no additional asymptotic space cost. Overall this solution solves the prefix-search by guaranteeing the same I/O- and space-bounds of the string B-tree, simultaneously over all values of B. In order to reduce the space usage, Bender et al. [2006] designed the (randomized) cache-oblivious string B-tree (abbreviated as COSB). It achieves the improved space of $(1+\epsilon)\mathsf{FC}(\mathcal{S}) + O(K \log N)$ bits, where $\mathsf{FC}(\mathcal{S})$ is the space required by the front-coded storage of the strings in $\mathcal{S}$ (see Section 2), and ϵ is a positive user-defined parameter that controls the trade-off between space usage and I/O-complexity of the query/update operations. COSB supports searches in $O(\log_B K + (1 + \frac{1}{\epsilon})(\mathsf{Scan}(P) + \mathsf{Scan}(succ(P))))$ I/Os, where $succ(P)$ is the successor of P in the ordered $\mathcal{S}$.[1] The solution is randomized, so the I/O bounds hold with high probability, and, more importantly for our subsequent discussions, the term $O((1 + \frac{1}{\epsilon})\mathsf{Scan}(succ(P)))$ may degenerate into $\Theta((1 + \frac{1}{\epsilon})\sqrt{N}/B)$ for some sets of strings.

Subsequently, Ferragina et al. [2008] proposed an improved in space cache-oblivious solution for the static version of the problem. They showed that there exists a static data structure which takes $(1+\epsilon)\mathsf{LB}(\mathcal{S}) + O(K)$ bits, where $\mathsf{LB}(\mathcal{S})$ is a lower bound to the storage complexity of the binary strings in $\mathcal{S}$. Searches can be supported in $O(\log_2 K + \mathsf{Scan}(P))$ I/Os or in $O(\log_B K + (1 + \frac{1}{\epsilon})(\mathsf{Scan}(P) + \mathsf{Scan}(succ(P))))$ I/Os. Even though this solution is deterministic, its query complexity still has the dependency on $\mathsf{Scan}(succ(P))$

[1] This index can be dynamized to support insertion and deletion of a string s in $O(\log_B K + (\log^2 N)\ (1 + \frac{1}{\epsilon})\mathsf{Scan}(s))$ I/Os plus the cost of identifying s's rank in $\mathcal{S}$.

which might be not bounded in terms of $|P|$, thus resulting suboptimal. For the sake of completeness, we note that the literature proposes many other compressed solutions to the prefix-search problem but their algorithms appear not to be suitable for the cache-oblivious model (see e.g., [Ferragina and Venturini 2010; Navarro and Mäkinen 2007; Hon et al. 2010]).

Recently, Belazzougui et al. [2010] introduced the *weak* variant of the problem that allows for a *one-sided error* in the answer to the prefix-search query. Namely, the answer is required to be correct only in the case that P is a prefix of some of the (binary) strings in $\mathcal{S}$; otherwise, it leaves to the algorithm the possibility to return an arbitrary answer. The *weak*-feature allowed the authors of that paper to prefix-search in the cache-oblivious model by taking $O(\log_2 |P| + \mathsf{Scan}(P))$ I/Os in the succinct space of $O(K \log \frac{N}{K})$ bits. This means that the string set $\mathcal{S}$ is not stored explicitly, because the solution uses only $O(\log \frac{N}{K})$ bits per string. This improvement is significant for very large string sets, and actually turns out to be optimal regardless of the query complexity. Subsequently, Ferragina [2013] proposed a very simple (randomized) solution for the weak-prefix search problem which matches the best known results, taking $O(\log_B N + \mathsf{Scan}(P))$ I/Os and $O(K \log \frac{N}{K})$ bits of space. The searching algorithm is randomized, and thus its answer is correct with high probability.

In this paper we attack three versions of increasing sophistication of the prefix-search problem, by asking ourselves the challenging question: *how much redundancy we have to add to the classic trie data structure in order to achieve* $O(\log_B K + \mathsf{Scan}(P))$ ***I/Os in the supported search operations.***[2]

— *Weak-prefix search.* Returns the (lexicographic) range of strings prefixed by P, or an arbitrary value whenever such strings do not exist.

— *Full-prefix search.* Returns the (lexicographic) range of strings prefixed by P, or $\perp$ if such strings do not exist.

— *Longest-prefix search.* Returns the (lexicographic) range of strings sharing the longest common prefix with P.

We get the above I/O bound of $O(\log_B K + \mathsf{Scan}(P))$ for the Weak-Prefix Search Problem, which is optimal, whereas for the other two problems we achieve $O(\log_B K + (1 + \frac{1}{\epsilon})\mathsf{Scan}(P))$ I/Os, for any constant $\epsilon > 0$. The space complexities are asymptotically optimal because they match the space lower bounds up to constant factors. This means that for the Weak-Prefix Search Problem we improve on Ferragina [2013] via a *deterministic* solution (rather than a randomized one) which also offers better space usage and better I/O-complexity; for the other two problems we improve on both Bender et al. [2006] and Ferragina et al. [2008] via a space-I/O optimal *deterministic* solution (rather than their randomized, space suboptimal, or I/O-inefficient solutions). The query complexity of our solution matches that of the string B-Tree by Ferragina and Grossi [1999] and the solution by Brodal and Fagerberg [2006] but significantly improves their space usage.

Technically speaking, our results are obtained by adopting a *new storage scheme* that extends the *locality-preserving front coding* scheme, at the base of COSB, in such a way that any prefix of the compressed strings can be decompressed with an optimal number of I/Os. Table I reports the main results for the three problems in the External Memory and the Cache-Oblivious Models.

[2]We remark that this query bound can be looked at as nearly optimal for the following first two problems because it has not been proved yet that the term $\log_B K$ is necessary in the space bounds obtained in this paper.

Table I. A summary of the main results for the three problems in the External Memory and Cache-Oblivious Models

Weak-Prefix Search Problem

I/Os	Space (bits)	Reference
$O(\log_2 \|P\| + \mathsf{Scan}(P))$	$O(K \log \frac{N}{K})$	[Belazzougui et al. 2010]
$O(\log_B N + \mathsf{Scan}(P))$ *	$O(K \log \frac{N}{K})$	[Ferragina 2013]
$O(\log_B K + \mathsf{Scan}(P))$	$\log \binom{\mathsf{Trie}(\mathcal{S})}{t-1} + O(K)$	Theorem 4.1

Full-Prefix Search Problem & Longest-Prefix Search Problem

I/Os	Space (bits)	Reference
$O(\log_B K + \mathsf{Scan}(P))$ **	$O(N + K \log N)$	[Ferragina and Grossi 1999]
$O(\log_B K + \mathsf{Scan}(P))$	$O(N + K \log N)$	[Brodal and Fagerberg 2006]
$O(\log_B K + (1+1/\epsilon)(\mathsf{Scan}(P) + \mathsf{Scan}(succ(P))))$ ***	$(1+\epsilon)\mathsf{FC}(\mathcal{S}) + O(K \log N)$	[Bender et al. 2006]
$O(\log_B K + (1+1/\epsilon)(\mathsf{Scan}(P) + \mathsf{Scan}(succ(P))))$	$(1+\epsilon)\mathsf{LB}(\mathcal{S}) + O(K)$	[Ferragina et al. 2008]
$O(\log_B K + (1+1/\epsilon)\mathsf{Scan}(P))$	$(1+\epsilon)\mathsf{LB}(\mathcal{S}) + O(K)$	Theorems 4.3 and 4.4

* The reported results are correct with high probability.
** The result holds only in the External Memory Model, with a disk with pages of size B.
*** The complexity of this data structure holds with high probability.

2. NOTATION AND BACKGROUND

In order to simplify the following presentation of our results, we assume we are dealing with *binary* strings. In the case of a larger alphabet Σ, it is enough to transform the strings over Σ into binary strings, and then apply our algorithmic solutions. Their I/O-complexity does not change because it depends only on the number of strings K in $\mathcal{S}$ and on the number of bits that fit in a disk block (hence $\Theta(B \log N)$ bits). As a further simplifying assumption we take $\mathcal{S}$ to be *prefix free*, so that no string in the set is the prefix of another string. This condition is satisfied in applications because of the `null`-character terminating each string.

Table II summarizes all our notations and terminology. Here we briefly recall a few algorithmic tools that we will deploy to design our algorithmic solutions to the three variants of the prefix-search problem stated in the previous section.

In the next solutions we will need two key tools which are nowadays the backbone of every compressed index: namely, Rank/Select data structures for binary strings. Their complexities are stated in the following theorems.

THEOREM 2.1 ([ELIAS 1974; FANO 1971]). *A binary vector $B[1 \dots m]$ with n bits set to 1 can be encoded by using $\log \binom{m}{n} + O(n)$ bits so that we can solve in $O(1)$ time the query $\mathsf{Select}_1(B, i)$, with $1 \le i \le n$, which returns the position in B of the ith occurrence of 1.*

THEOREM 2.2 ([MUNRO 1996]). *A binary vector $B[1 \dots m]$ with n bits set to 1 can be encoded by using $m + o(m)$ bits so that we can solve in $O(1)$ time the queries $\mathsf{Rank}_1(B, i)$, with $1 \le i \le m$, which returns the number of 1s in the prefix $B[1 \dots i]$, and $\mathsf{Select}_1(B, i)$, with $1 \le i \le n$, which returns the position in B of the ith occurrence of 1.*

In the next sections we will often use the following inequality.

Table II. A summary of our notation and terminology.

$\mathcal{S}$	The set of strings
N	Total length of the strings in $\mathcal{S}$
K	Number of strings in $\mathcal{S}$
$\mathcal{T}_{\mathcal{S}}$	The compact trie built on $\mathcal{S}$
t	Number of nodes in $\mathcal{T}_{\mathcal{S}}$; it satisfies $t \leq 2K-1$
$\mathsf{p}(u)$	The parent of the node u in $\mathcal{T}_{\mathcal{S}}$
$\mathsf{label}(u)$	The label of the edge $(\mathsf{p}(u), u)$
$\mathsf{string}(u)$	The string spelled out by the path in $\mathcal{T}_{\mathcal{S}}$ reaching u from the root
$\hat{\mathcal{S}}$	The set $\mathcal{S}$ augmented with string $\mathsf{string}(u)$ for all u in $\mathcal{T}_{\mathcal{S}}$
$\mathsf{Trie}(\mathcal{S})$	The sum of edge-label lengths in $\mathcal{T}_{\mathcal{S}}$
$\mathsf{LB}(\mathcal{S})$	Lower bound (in bits) to the storage complexity of the set of strings $\mathcal{S}$ which is equal to $\mathsf{Trie}(\mathcal{S}) + \log \binom{\mathsf{Trie}(\mathcal{S})}{t-1}$ bits
M	The internal memory size, unknown to the algorithm
B	The disk page size measured as the number of machine words that fit in one page, unknown to the algorithm
$\mathsf{Scan}(P)$	The optimal number of I/Os required to scan a binary string $P[1,p]$ (namely, $\mathsf{Scan}(P) = \Theta(1 + \lvert P\rvert/(B \log N))$ I/Os)

LEMMA 2.3 ([BRODNIK AND MUNRO 1999]). *Let $m_1, \ldots, m_s$ and $n_1, \ldots, n_s$ be non-negative integers, it holds*

$$\sum_{i=1}^{s} \lceil \log \binom{m_i}{n_i} \rceil < \log \binom{\sum_{i=1}^{s} m_i}{\sum_{i=0}^{s} n_i} + s.$$

Front coding is a compression scheme for strings which represents $\mathcal{S}$ as the sequence $\mathsf{FC}(\mathcal{S}) = \langle n_1, L_1, n_2, L_2, \ldots, n_K, L_K \rangle$, where n_i is the length of the longest common prefix between S_{i-1} and S_i, and L_i is the suffix of S_i remaining after the removal of its first n_i (shared) characters. Hence $|L_i| = |S_i| - n_i$. The first string S_1 is represented in its entirety, so that $L_1 = S_1$ and $n_1 = 0$. FC is a well established practical method for encoding a (lexicographically sorted) string set [Witten et al. 1999], and FC will be used to denote either the algorithmic scheme or its output size in bits. As an example, consider the set of strings $\mathcal{S} = \{\mathtt{000000000}, \mathtt{000000001}, \mathtt{000001110}, \mathtt{000001111}, \mathtt{000010100}, \mathtt{000010101}, \mathtt{00001011}, \mathtt{0001}, \mathtt{1}\}$. Its front coding is $\mathsf{FC}(\mathcal{S}) = \langle 0, \mathtt{000000000}, 8, \mathtt{1}, 5, \mathtt{1110}, 8, \mathtt{1}, 4, \mathtt{10100}, 8, \mathtt{1}, 7, \mathtt{1}, 3, \mathtt{1}, 0, \mathtt{1} \rangle$.

In order to estimate the space cost of $\mathsf{FC}(\mathcal{S})$ in bits, Ferragina et al. [2008] introduced the so-called *trie measure* of $\mathcal{S}$, defined as: $\mathsf{Trie}(\mathcal{S}) = \sum_{i=1}^{K} |L_i|$, which accounts for the number of characters outputted by $\mathsf{FC}(\mathcal{S})$. And then, they devised a lower bound $\mathsf{LB}(\mathcal{S})$ to the storage complexity of $\mathcal{S}$ which adds to the trie measure the cost, in bits, of storing the lengths $|L_i|$. We have $\mathsf{LB}(\mathcal{S}) = \mathsf{Trie}(\mathcal{S}) + \log \binom{\mathsf{Trie}(\mathcal{S})}{t-1}$ bits.

In the paper we will often obtain bounds in terms of $\log \binom{\mathsf{Trie}(\mathcal{S})}{t-1}$, so the following fact is helpful.

FACT 1. *For any dictionary of strings $\mathcal{S}$, $\log \binom{\mathsf{Trie}(\mathcal{S})}{t-1} = O(K \log \frac{N}{K})$. Nevertheless there exist dictionaries for which $K \log \frac{N}{K}$ may be up to $\log K$ times larger than $\log \binom{\mathsf{Trie}(\mathcal{S})}{t-1}$. Also, $O(\log \binom{\mathsf{Trie}(\mathcal{S})}{t-1}) = o(\mathsf{Trie}(\mathcal{S})) + O(K)$.*

PROOF. The first statement comes from the observation that $\mathsf{Trie}(\mathcal{S}) \leq N$, $t \leq 2K-1$ and $\log \binom{\mathsf{Trie}(\mathcal{S})}{t-1} \leq t \log \frac{\mathsf{Trie}(\mathcal{S})}{t} + O(t)$.

As far as the second statement is concerned, consider the dictionary of strings $\mathcal{S} = \{S_1, \ldots, S_K\}$ of total length N with $K = \frac{1}{2}(\sqrt{8N+1} - 1) = O(\sqrt{N})$. String S_i is equal to $1^{i-1}0$. We have $\mathsf{Trie}(\mathcal{S}) = 2K - 1$ because each string S_i, with $i \leq K$, shares a common

prefix of length $i-2$ with the previous string S_{i-1}. Thus $\log \binom{\mathsf{Trie}(\mathcal{S})}{t-1} = \log \binom{2K-1}{t-1} = O(K)$, because $t-1 \leq 2K-2$, which is $\log K$ times smaller than $K \log \frac{N}{K} = \Theta(K \log K)$.

For the third statement, we observe that every trie edge is labeled with at least one bit, so $t-1 \leq \mathsf{Trie}(\mathcal{S})$. We are interested in the term $\log \binom{\mathsf{Trie}(\mathcal{S})}{t-1}$ which is bounded above by $t \log \frac{\mathsf{Trie}(\mathcal{S})}{t} + O(t)$. We distinguish two cases: $t = o(\mathsf{Trie}(\mathcal{S}))$ and $t = \Theta(\mathsf{Trie}(\mathcal{S}))$. In the former case, $t \log \frac{\mathsf{Trie}(\mathcal{S})}{t} + O(t) = o(\mathsf{Trie}(\mathcal{S}))$, while in the latter case $t \log \frac{\mathsf{Trie}(\mathcal{S})}{t} + O(t) = \Theta(t) = O(K)$. Hence the term $\log \binom{\mathsf{Trie}(\mathcal{S})}{t-1}$ is bounded above by $o(\mathsf{Trie}(\mathcal{S})) + O(K)$. □

FC can be encoded with three binary arrays. One array consists of concatenating all binary sequences corresponding to the suffixes L_i of $\mathcal{S}$'s strings, which sum up to $\mathsf{Trie}(\mathcal{S})$ bits; another array marks the first bit of each of those suffixes, thus consists of $\mathsf{Trie}(\mathcal{S})$ bits; and, finally, the third array encodes in unary the length of the shared prefixes between consecutive strings (i.e., values n_i). Since $\sum_{i=1}^{K} n_i \leq N$, the last array consists of less than N bits. The last two arrays can be encoded with the solution in Theorem 2.1 and, thus, they require at most $\log \binom{\mathsf{Trie}(\mathcal{S})}{K} + \log \binom{N}{K} + O(K) = O(K \log \frac{N}{K})$ bits. Thus, the representation obtained via front coding takes

$$\mathsf{LB}(\mathcal{S}) \leq \mathsf{FC}(\mathcal{S}) \leq \mathsf{Trie}(\mathcal{S}) + O(K \log \frac{N}{K}) \leq \mathsf{LB}(\mathcal{S}) + O(K \log \frac{N}{K}) \text{ bits.} \quad (1)$$

Ferragina et al. [2008] show that pathological cases exist in which front coding requires space close to that upper bound.

A simple variant of FC, called *rear coding* (RC), achieves a more succinct storage of the string set $\mathcal{S}$ by specifying not n_i but rather the length $|S_{i-1}| - n_i$ of the suffix of S_{i-1} to be removed in order to get the longest common prefix between S_{i-1} and S_i. This simple change is crucial to avoid *repetitive* encodings of the same longest common prefixes, which is the cause of the space inefficiency of FC. Following the previous example for the front coding scheme, the rear coding of the same set of strings $\mathcal{S} = \{000000000, 000000001, 000001110, 000001111, 000010100, 000010101, 00001011, 0001, 1\}$ is $\mathsf{RC}(\mathcal{S}) = \langle 0, 000000000,\ 1, 1,\ 4, 1110,\ 1, 1,\ 5, 10100,\ 1, 1,\ 2, 1,\ 5, 1,\ 4, 1 \rangle$.

We note that smaller numbers than FC's encoding of those strings, and indeed RC is able to come very close to LB. The idea is to encode the lengths of the suffixes to be dropped via a binary array of length $\mathsf{Trie}(\mathcal{S})$ with $K-1$ bits set to 1, as indeed those suffixes partition $\mathcal{T}_{\mathcal{S}}$ into K disjoint paths from the leaves to the root. Just think to scanning the leaves of $\mathcal{T}_{\mathcal{S}}$ rightwards, then the suffix of S_i explicitly written down by RC is the prefix of the leaf-to-root path that starts at the leaf associated to S_i and stops as soon as that path meets the previous leaf-to-root path starting at S_{i-1}. Clearly, S_1 is written explicitly because its leaf-to-root path is the leftmost one in $\mathcal{T}_{\mathcal{S}}$. As an illustrative example, let us refer to Figure 2. We see that the leaf-to-root path starting at leaf S_2 stops immediately after percolating one edge (with binary label 1), at the node with label 4 (whose distance in bits is 1 from the previous leaf S_1), because it meets the leftmost path which started at leaf S_1. The next leaf-to-root path starting at leaf S_3 stops after percolating two edges (with binary labels 111 and 0) at the node with label 3 (whose distance in bits is 4 from the previous leaf S_2), because at this point it meets the leaf-to-root path which started at leaf S_2. And so on. The bits over the percolated edges and the distances in bits correspond exactly to the encoding emitted by RC.

As we have done with FC, RC can be encoded with three binary arrays. An array consists of concatenating all binary sequences corresponding to the suffixes of $\mathcal{S}$'s strings; an array marks the first bit of each of those suffixes, and, finally, the third array encodes in unary the length of the dropped suffixes. With reference to the previous example, we have that the first binary array is given by $[000000000, 1, 1110, 1, 10100, 1, 1, 1, 1]$,

the second binary array is given by $[\texttt{000000000}, \texttt{1}, \texttt{1000}, \texttt{1}, \texttt{10000}, \texttt{1}, \texttt{1}, \texttt{1}, \texttt{1}]$, whilst the third array is given by the unary encoding of $[1, 4, 1, 5, 1, 2, 5, 4]$. Since $\sum_{i=1}^{K} |S_{i-1}| - n_i \leq \mathsf{Trie}(\mathcal{S})$, the last array in RC consists of less than $\mathsf{Trie}(\mathcal{S})$ bits, and, thus, it is shorter than in FC. Thus, if we encode the two arrays with the solution in Theorem 2.1, their space usage decreases to at most $2 \log \binom{\mathsf{Trie}(\mathcal{S})}{K} + O(K)$ bits.

As a result, Ferragina et al. [2008] showed that the rear-coding representation above takes

$$\mathsf{RC}(\mathcal{S}) \leq \mathsf{Trie}(\mathcal{S}) + 2 \log \binom{\mathsf{Trie}(\mathcal{S})}{K} + O(K) = (1 + o(1))\mathsf{LB}(\mathcal{S}) + O(K) \text{ bits}, \qquad (2)$$

where the latter equality follows from the third statement in Fact 1 and the observation that $K < t$. Comparing eqn. (2) and (1), the difference between RC and FC is in the encoding of the n_i, so $\mathsf{Trie}(\mathcal{S}) \leq N$ (typically, $\mathsf{Trie}(\mathcal{S}) \ll N$).

The main drawback of front/rear codings is that decoding a string S_j might require the decompression of the entire sequence $\langle 0, L_1, \ldots, n_j, L_j \rangle$. In order to overcome this drawback, Bender et al. [2006] proposed a variant of FC, called *locality-preserving front coding* (abbreviated as LPFC), that, given a parameter ϵ, adaptively partitions $\mathcal{S}$ into blocks such that decoding any string S_j takes optimal $O((1 + \frac{1}{\epsilon})|S_j|/B)$ I/Os, and requires $(1 + \epsilon)\mathsf{FC}(\mathcal{S})$ bits of space. This adaptive scheme is agnostic in the parameter B and offers a clear space/time tradeoff in terms of the user-defined parameter ϵ.

3. A KEY TOOL: CACHE-OBLIVIOUS PREFIX RETRIEVAL

The novelty of our paper consists in a surprisingly simple representation of $\mathcal{S}$ which is compressed and still supports the cache-oblivious retrieval of any prefix of any string of $\mathcal{S}$ in an optimal number of I/Os and bits of space (up to constant factors). The striking news is that, despite its simplicity, this result will constitute the foundation for our improved algorithmic solutions.

In this section we describe our solution on tries even though it is sufficiently general to represent any (labeled) tree in compact form while guaranteeing optimal traversal of any root-to-a-node path in the cache-oblivious model. We assume that the trie nodes are numbered according to the time of their DFS visit. Any node u in $\mathcal{T}_{\mathcal{S}}$ is associated with $\mathsf{label}(u)$ which is (the variable length) string on the edge $(p(u), u)$, where $p(u)$ is the parent of u in $\mathcal{T}_{\mathcal{S}}$. Observe that any node u uniquely identifies the string $\mathsf{string}(u)$ that is a prefix of all strings of $\mathcal{S}$ descending from u. The $\mathsf{string}(u)$ can be obtained by concatenating the labels of the nodes on the path from the root to u. Our goal is to design a storage scheme whose space use is close to $\mathsf{LB}(\mathcal{S})$ bits and supports in optimal time/IO the following operation:

— $\mathsf{Retrieval}(u, \ell)$ returns the prefix of the string $\mathsf{string}(u)$ with length $\ell \in (|\mathsf{string}(\mathsf{p}(u))|, |\mathsf{string}(u)|]$. So the returned prefix ends up in the edge $(\mathsf{p}(u), u)$.

In other words, Retrieval supports the efficient access to any string prefix, and thus any root-to-a-node path in the trie built over $\mathcal{S}$. Formally, we aim to prove the following theorem.

THEOREM 3.1. *Given a set $\mathcal{S}$ of K binary strings with total length N, there exists a storage scheme for $\mathcal{S}$ that occupies $(1 + \epsilon)\mathsf{LB}(\mathcal{S}) + O(K)$ bits, where $\epsilon > 0$ is any fixed constant, and solves the query $\mathsf{Retrieval}(u, \ell)$ in optimal $O(1 + (1 + \frac{1}{\epsilon})\frac{\ell}{B \log N})$ I/Os.*

Before presenting the proof of this theorem, let us discuss the efficiency of two close relatives of our solution: *Giraffe tree decomposition* [Brodal and Fagerberg 2006] and *Locality-preserving front coding* (LPFC) [Bender et al. 2006]. The former solution has the

same time complexity as our solution but has a space usage of at least $3 \cdot \mathsf{LB}(\mathcal{S}) + O(K)$ bits. The latter approach has (almost) the same space usage as our solution but provides no guarantee on the number of I/Os required to access *prefixes* of the strings in $\mathcal{S}$.

Our novel storage scheme accurately lays out the labels of nodes of $\mathcal{T}_{\mathcal{S}}$ so that any $\mathsf{string}(u)$ can be retrieved in optimal $O((1+\frac{1}{\epsilon})\mathsf{Scan}(\mathsf{string}(u)))$ I/Os. This is sufficient to obtain the bound stated in Theorem 3.1 because, once we have reconstructed $\mathsf{string}(p(u))$, we can complete the execution of $\mathsf{Retrieval}(u,\ell)$ by accessing the prefix of $\mathsf{label}(u)$ of length $j = \ell - |\mathsf{string}(p(u))|$ which is written consecutively in memory. One key feature of our storage scheme is a proper replication of some labels in the layout, whose space is bounded by $\epsilon \cdot \mathsf{LB}(\mathcal{S})$ bits.

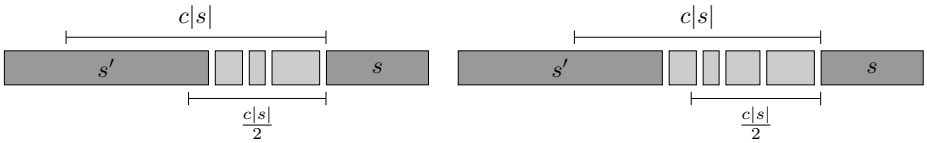
The starting point is the amortization argument in LPFC [Bender et al. 2006] which represents $\mathcal{S}$ by means of a variant of the classic front-coding in which some strings are stored explicitly rather than front-coded. More precisely, LPFC writes the string S_1 explicitly, whereas all subsequent strings are encoded in accordance with the following argument. Suppose that the scheme already compressed $i-1$ strings and has to compress string S_i. It scans back $c|S_i|$ characters (here bits, because strings are binary) in the current representation to check if it is possible to decode S_i, where $c = 2 + 2/\epsilon$. If this is the case, S_i is compressed by writing $\langle n_i, L_i\rangle$; otherwise S_i is fully written as $\langle 0, S_i\rangle$. A sophisticated amortization argument by Bender et al. [2006] proves that LPFC requires $(1+\epsilon)\mathsf{LB} + O(K\log(N/K))$ bits of storage and an optimal decompression time/IO of any string $S_i \in \mathcal{S}$, namely $O((1+\frac{1}{\epsilon})\mathsf{Scan}(S_i))$ I/Os. This space-bound can be improved by replacing front coding with rear coding, whilst still ensuring an optimal decompression time/IO. We call this approach LPRC, Locality Preserving Rear Coding, and prove straightforwardly the following result (here included for the sake of completeness).

FACT 2. *Given a set $\mathcal{S}$ of K binary strings with total length N, the* LPRC *storage scheme for $\mathcal{S}$ occupies $(1+\epsilon)\mathsf{LB}(\mathcal{S}) + O(K)$ bits, where $\epsilon > 0$ is any fixed constant, and decompresses any string $s \in \mathcal{S}$ in optimal $O(1 + (1+\frac{1}{\epsilon})\mathsf{Scan}(s))$ I/Os.*

PROOF. This proof adapts the analysis in Bender et al. [2006] to use rear coding in place of front coding as compressed storage scheme for the string set $\mathcal{S}$. As in Bender et al. [2006], we can deduce that LPRC takes the optimal I/O-bound stated above to decompress any string S_i. Indeed, either S_i is fully written or it is rear-coded, but, in this case, it is enough to look back at most $c|S_i|$ bits (by definition of LPRC). So it only remains to show that the extra-space occupied by the *fully-written* strings, namely the ones in $\mathcal{S}$ that were possibly compressed by RC but are left uncompressed by LPRC, sums up to ϵ times the rear-coding bound given in eqn. 2.

Let s' and s be two consecutive strings left uncompressed by LPRC, and denote the $c|s|$ bits belonging to the suffixes of $\mathcal{S}$'s strings that have been explored during the backward check as the *left extent* of s. There can be no other uncompressed strings beginning in the left extent of s (otherwise it would have been compressed by RC), and note that s' starts before the left extent of s but could *end* within that extent. We consider two cases for s depending on the amount of bits written for the rear-coded strings that lie between s' and s. The first case is called *crowded* and occurs when the number of bits is at most $\frac{c|s|}{2}$ (see Figure 1 left); the second case is called *uncrowded* and occurs when the number of bits is at least $\frac{c|s|}{2}$ (see Figure 1 right).

If s is crowded, then s' starts before the left extent of s (hence at least $c|s|$ bits before s) but ends within the last $c|s|/2$ bits of that extent. This means that $|s'| \geq c|s|/2$. In the other (uncrowded) case, s is preceded by at least $c|s|/2$ bits of rear-coded strings. These two properties allow us to bound the total length of the uncompressed strings.



(a) String s is crowded (b) String s is uncrowded

Fig. 1. The light gray rectangles denote the suffixes of the rear-coded strings, and the gray rectangles denote the two consecutive uncompressed strings s' and s.

We partition the fully-written strings into chains composed of one uncrowded string followed by the maximal sequence of crowded strings (recall that all of them are uncompressed in LPRC). The total number of bits in each chain is proportional to the length of its first string, namely the uncrowded one. Precisely, consider the chain $w_1 w_2 \cdots w_x$ of consecutive uncompressed strings, where w_1 is uncrowded and the following w_is are crowded. Take any crowded w_i. By the observation above, we have that $|w_{i-1}| \geq c|w_i|/2$ or, equivalently, $|w_i| \leq 2|w_{i-1}|/c \leq \cdots \leq (2/c)^{i-1}|w_1|$. Summing the length of the uncompressed strings in the chain, we have $\sum_{i=1}^{x} |w_i| = |w_1| + \sum_{i>1}^{x} |w_i| \leq |w_1| + \sum_{i>1}^{x} (2/c)^{i-1}|w_1| = |w_1| \sum_{i\geq 0}^{x} (2/c)^i < \frac{c|w_1|}{c-2}$. The latter inequality follows from the assumption that we have set $c = 2 + 2/\epsilon > 2$.

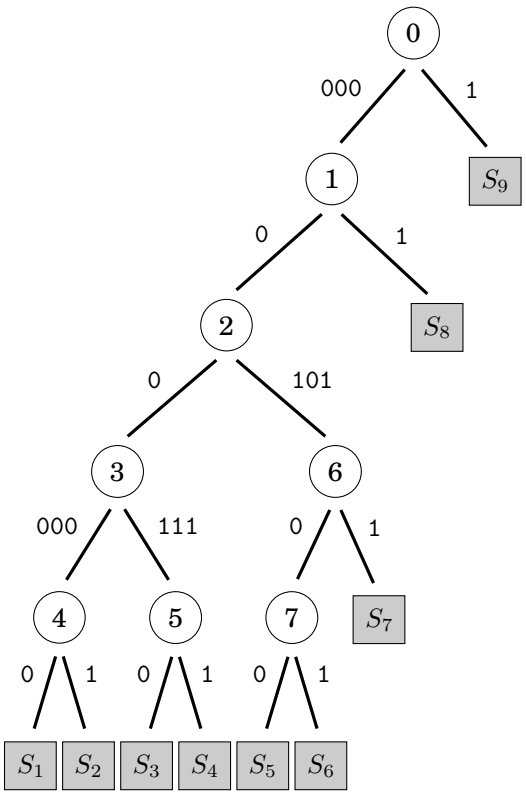
Finally, since w_1 is uncrowded, it is preceded by at least $c|w_1|/2$ bits of rear-coded string (see above); moreover we know that the total storage of uncrowded strings is bounded above by RC. Consequently we can upper bound the space induced by all uncrowded strings, starting the chains, by $\frac{2}{c}\mathsf{RC}$. By plugging this into the previous bound on the total length of the chains of uncompressed strings, we get $\frac{c}{c-2} \times \frac{2}{c}\mathsf{RC} = \frac{2}{c-2}\mathsf{RC} = \epsilon\mathsf{RC}$. Summing this space cost to the one needed to store the strings which are rear-coded also by LPRC, namely no more than RC bits (because they are a subset of $\mathcal{S}$), we get an upper bound of $(1+\epsilon)\mathsf{RC} \leq (1+\epsilon)\mathsf{LB}(\mathcal{S}) + O(K)$, because of eqn. 2. □

Although elegant, this fact does not suffice to guarantee an optimal decompression for prefixes of s because this might cost up to $\Theta((1+\frac{1}{\epsilon})\mathsf{Scan}(s))$ I/Os regardless of the length of the string prefix to be decompressed.

In order to circumvent this limitation, we modify LPRC (or, similarly, LPFC) as follows. We define the superset $\hat{\mathcal{S}}$ of $\mathcal{S}$ which contains one additional string $\mathsf{string}(u)$ for each node u in $\mathcal{T}_{\mathcal{S}}$ (possibly a leaf). The string $\mathsf{string}(u)$ is a prefix of $\mathsf{string}(v)$, for any descendant v of u in $\mathcal{T}_{\mathcal{S}}$, so $\mathsf{string}(u)$ is lexicographically smaller than $\mathsf{string}(v)$. The lexicographically ordered $\hat{\mathcal{S}}$ can thus be obtained by visiting the nodes of $\mathcal{T}_{\mathcal{S}}$ according to a DFS visit. Figure 2 shows $\mathcal{T}_{\mathcal{S}}$ and $\hat{\mathcal{S}}$ for a set of strings.

In order to design an optimal storage and decompressor for string prefixes, we introduce the following data structures.

— R is the compressed output obtained by computing $\mathsf{LPRC}(\hat{\mathcal{S}})$. Since we require that all pairs emitted by $\mathsf{LPRC}(\hat{\mathcal{S}})$ are self-delimited, we prefix each pair p with its length coded with Elias' Gamma thus taking $2\log|p|+1$ bits [Witten et al. 1999]. By Fact 2 we know that R requires no more than $(1+\epsilon)\mathsf{LB}(\hat{\mathcal{S}}) + O(K)$ bits. Thus the key observation is that the trie measure of $\hat{\mathcal{S}}$ coincides with that of $\mathcal{S}$ (i.e., $\mathsf{Trie}(\hat{\mathcal{S}}) = \mathsf{Trie}(\mathcal{S})$), so that $|R| = (1+\epsilon)\mathsf{LB}(\hat{\mathcal{S}}) + O(K) = (1+\epsilon)\mathsf{LB}(\mathcal{S}) + O(K)$. Moreover, the cost of self-delimiting the t pairs p_i emitted by LPRC with Elias' Gamma coding is at most $\sum_{i=1}^{t}(2\log|p_i|+1) \leq 2t\log(|R|/t) + O(t) = o(\mathsf{Trie}(\mathcal{S})) + O(K)$ bits by Jensen's inequality



(a) Trie $\mathcal{T}_S$

000
0000
00000
00000000
000000000
000000001
00000111
000001110
000001111
0000101
00001010
000010100
000010101
00001011
0001
1

(b) Superset $\hat{\mathcal{S}}$

Fig. 2. The figure shows the trie $\mathcal{T}_S$ of the set of strings $\mathcal{S} = \{000000000, 000000001, 000001110, 000001111, 000010100, 000010101, 00001011, 0001, 1\}$ and, to the right, the superset $\hat{\mathcal{S}}$ of $\mathcal{S}$ where the string $\mathsf{string}(u)$ is written in black if the node u is a leaf or otherwise in gray.

and Fact 1. As a result, the overall cost of storing R and self-delimiting its pairs is $(1+\epsilon)\mathsf{LB}(\mathcal{S}) + O(K)$ bits.

— The binary array $E[1 \dots |R|]$ which sets to 1 the positions in R where the encoding of some $\mathsf{string}(u)$ starts. E contains $t-1$ bits set to 1, one per trie's edges apart from the first one which surely starts at $E[1]$. Array E is enriched with the data structure in Theorem 2.1 so that Select_1 queries can be computed in constant time. The space usage of E is $\log \binom{|R|}{t-1}$ bits (Theorem 2.1). Therefore $|E| \leq t \log(|R|/t) + O(t) = o(\mathsf{Trie}(\mathcal{S})) + O(K)$ bits[3].

— The binary array $V[1 \dots t]$ that has an entry for each node in $\mathcal{T}_S$ according to their (DFS-)order. The entry $V[u]$ is set to 1 whenever $\mathsf{string}(u)$ has been fully-copied in R, 0 otherwise. We augment V with the data structure of Theorem 2.2 to support Rank and Select queries. Vector V requires just $O(K)$ bits, according to Theorem 2.2.

[3]Note that the binary array E could be also used to self-delimit the pairs p_i of the previous item instead of Gamma coding. However, there is a subtle size issue which may impact on the cache-oblivious scanning of R. Indeed, the Elias-Fano coding of E does not guarantee that the space taken by an individual pair is proportional to its space in R. Conversely, this is guaranteed by using the self-delimiting Gamma code.

In order to answer $\mathsf{Retrieval}(u, \ell)$ we first implement the retrieval of $\mathsf{string}(u)$, namely the string spelled out by the root-to-u path in $\mathcal{T}_S$. The query $\mathsf{Select}_1(E, u)$ gives in constant time the position in R where the encoding of $\mathsf{string}(u)$ starts. Now, if this string is stored uncompressed in R then we are done; otherwise we have to reconstruct it. This has some subtle issues that have to be addressed. For example, we do not know the length of $\mathsf{string}(u)$ since the array E encodes the individual edge-labels and not their lengths from the root of $\mathcal{T}_S$. Nevertheless we reconstruct $\mathsf{string}(u)$ forward by starting from the first copied string (say, $\mathsf{string}(v)$) that precedes $\mathsf{string}(u)$ in R. The node index v is obtained by computing $\mathsf{Select}_1(V, \mathsf{Rank}_1(V, u))$ which identifies the position of the first 1 in V that precedes the entry corresponding to u (i.e., the closer uncompressed string preceding u in the DFS-visit of $\mathcal{T}_S$).

Assume that the copy of $\mathsf{string}(v)$ starts at position p_v in R, which is computed by selecting the v-th 1 in the E. By the DFS-order processing of $\mathcal{T}_S$ and by the fact that $\mathsf{string}(u)$ is rear-coded, it follows that $\mathsf{string}(u)$ can be reconstructed by copying characters in R starting from position p_v up to the occurrence of $\mathsf{string}(u)$. We recall that if w and w' are two nodes consecutive in the DFS-visit of $\mathcal{T}_S$, rear-coding writes the number $|\mathsf{string}(w)| - \mathsf{lcp}(\mathsf{string}(w), \mathsf{string}(w'))$ (namely, the length of the suffix of $\mathsf{string}(w)$ that we have to remove from w in order to obtain the length of its longest common prefix with $\mathsf{string}(w')$) and the remaining suffix of w'. This information is exploited in reconstructing $\mathsf{string}(u)$. We start by copying $\mathsf{string}(v)$ into a buffer, and then scan R forward from position p_v. For every value m written by rear-coding, we overwrite the last m characters of the buffer with the characters in R of the suffix of the current string (delimited by E's bits set to 1). At the end, the buffer will contain $\mathsf{string}(u)$. By LPRC's properties, we are guaranteed that this scan takes $O((1 + \frac{1}{\epsilon})\mathsf{Scan}(\mathsf{string}(u))$ I/Os.

Let us now come back to the solution of $\mathsf{Retrieval}(u, \ell)$. First of all we reconstruct $\mathsf{string}(\mathsf{p}(u))$, then determine the edge-label $(\mathsf{p}(u), u)$ in E given the DFS-numbering of u and a Select_1 operation over E. We finally take from this string its (contiguous) prefix of length $\ell - |\mathsf{string}(\mathsf{p}(u))|$; the latter is known because we have in fact reconstructed that string.

4. SEARCHING STRINGS: THREE PROBLEMS

In this section we address the three problems presented in the introduction; they allow us to frame the wide spectrum of algorithmic difficulties and solutions related to the search for a pattern within a string set.

PROBLEM 1 (Weak-Prefix Search Problem). *Let $\mathcal{S} = \{S_1, S_2, \ldots, S_K\}$ be a set of K binary strings of total length N. We wish to preprocess $\mathcal{S}$ so that, given a pattern P, we can efficiently answer the query* $\mathsf{weakPrefix}(P)$ *which asks for the range of strings in $\mathcal{S}$ prefixed by P. An arbitrary answer could be returned whenever P is not a prefix of any string in $\mathcal{S}$.*

The lower bound in Belazzougui et al. [2010] states that $\Omega(K \log \frac{N}{K})$ bits are necessary regardless of the query time. We show the following theorem.

THEOREM 4.1. *Given a set of $\mathcal{S}$ of K binary strings of total length N, there exists a deterministic data structure requiring $\log \binom{\mathsf{Trie}(\mathcal{S})}{t-1} + O(K)$ bits of space that solves the* Weak-Prefix Search Problem *for any pattern P with $O(\log_B K + \mathsf{Scan}(P))$ I/Os.*

The space usage is optimal up to constant factor since $\log \binom{\mathsf{Trie}(\mathcal{S})}{t-1}$ is always at most $K \log \frac{N}{K}$ (see Fact 1). Moreover, our refined estimate of the space usage, which depends on the characteristics of $\mathcal{S}$, may go below the general lower bound in Belazzougui et al. [2010] which depends only on N and K. Our improvement may be up to a factor $\Theta(\log K)$, see Fact 1. The query time instead is almost optimal, because it is not clear

whether the term $\log_B K$ is necessary with this space bound. In short, our data structure is deterministic, smaller and faster than previously known solutions.

Technically speaking we follow the solution in Ferragina [2013] by using two-level indexing. We start by partitioning $\mathcal{S}$ into $s = K/\log N$ groups of (contiguous) strings defined as follows: $\mathcal{S}_i = \{S_{1+i\log N}, S_{2+i\log N}, \ldots, S_{(i+1)\log N}\}$ for $i = 0, 1, 2, \ldots, s-1$. We then construct a subset $\mathcal{S}_{\mathsf{top}}$ of $\mathcal{S}$ consisting of $2s = \Theta(\frac{n}{\log n})$ representative strings obtained by selecting the first and the last string in each of these groups. The index in the first level is responsible for searching for the pattern P within the set $\mathcal{S}_{\mathsf{top}}$, in order to identify an approximate range. This range is guaranteed to contain the range of strings prefixed by P. A search on the second level suffices to identify the correct range of strings prefixed by P. We have two crucial differences w.r.t. the solution in Ferragina [2013]: 1) our index is deterministic; 2) our space-optimal solution for the second level is the key for achieving Theorem 4.1.

First level. As in Ferragina [2013] we build the Patricia Trie $\mathsf{PT}_{\mathsf{top}}$ over the strings in $\mathcal{S}_{\mathsf{top}}$ with the change that we store in each node u of $\mathsf{PT}_{\mathsf{top}}$ a fingerprint of $O(\log N)$ bits computed for $\mathsf{string}(u)$ according to Karp-Rabin fingerprinting [Karp and Rabin 1987]. The crucial difference w.r.t. Ferragina [2013] is the use of a (deterministic) injective instance of Karp-Rabin that maps any prefix of any string in $\mathcal{S}$ into a distinct value in an interval of size $O(N^2)$.[4] Given a string $S[1 \ldots s]$, the Karp-Rabin fingerprinting $\mathsf{rk}(S)$ is equal to $\sum_{i=1}^{s} S[i] \cdot r^i \pmod M$, where M is a prime number and r is a randomly chosen integer in $[1, M-1]$. Given the set of strings $\mathcal{S}$, we can obtain an instance $\mathsf{rk}()$ of the Karp-Rabin fingerprinting that maps all the prefixes of all the strings in $\mathcal{S}$ to the first $[M]$ integers without collisions, with M chosen from the first $\Theta(N^2)$ integers. It is known that a value of t that guarantees no collisions can be found in expected $O(1)$ attempts. In the cache-oblivious setting, this implies that finding a suitable function requires $O(\mathsf{Sort(N)})$ I/Os in expectation, where $\mathsf{Sort(N)}$ is the number of I/Os required to sort N integers and thus to check whether the selected hash function has no collision.

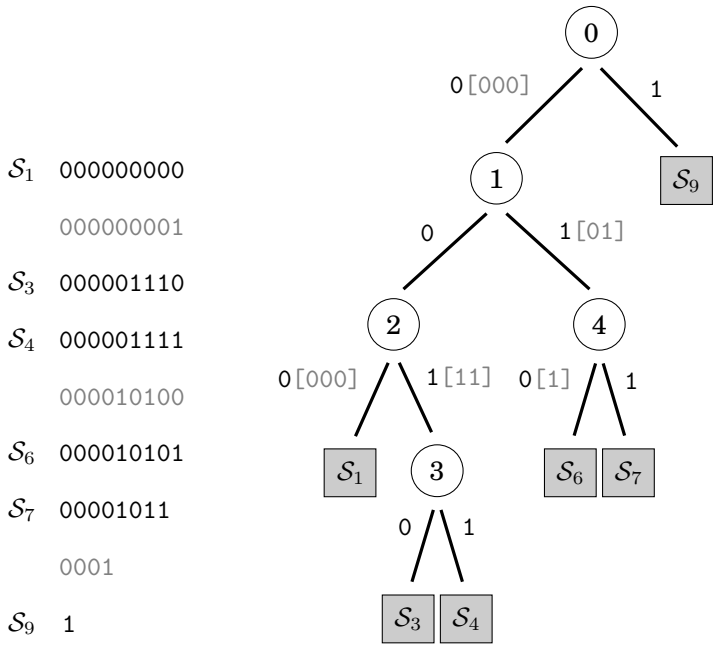
Given the binary Patricia Trie $\mathsf{PT}_{\mathsf{top}}$ and the pattern P, our goal is to find the lowest edge $e = (v, w)$ such that $\mathsf{string}(v)$ is a prefix of P and $\mathsf{string}(w)$ is not. This edge can be found with a standard trie search on $\mathsf{PT}_{\mathsf{top}}$ where fingerprints of P are compared with the ones stored in the traversed nodes. A cache-oblivious efficient solution is obtained by laying out $\mathsf{PT}_{\mathsf{top}}$ via the centroid trie decomposition [Bender et al. 2006]. This layout guarantees that the above search requires $O(\log_B K + \mathsf{Scan}(P))$ I/Os.

We note that the algorithm proposed in Ferragina [2013] identifies the edge e only with high probability. The reason is that a prefix of P and a prefix of a string in $\mathcal{S}$ may have the same fingerprint even if they are different. Our use of the injective Karp-Rabin fingerprints avoids this situation by guaranteeing that the search is always correct[5].

Figure 3 shows the strings in set $\mathcal{S}_{\mathsf{top}}$ and the corresponding patricia trie $\mathsf{PT}_{\mathsf{top}}$ for a set of binary strings.

Second level. For each edge $e = (u, v)$ of $\mathsf{PT}_{\mathsf{top}}$ we define the set of strings $\mathcal{S}_e$ as follows. Assume that each node v of $\mathsf{PT}_{\mathsf{top}}$ points to its leftmost/rightmost descending leaf, denoted by $L(v)$ and $R(v)$ respectively. We call $\mathcal{S}_{L(v)}$ and $\mathcal{S}_{R(v)}$ the two groups of strings, from the grouping above, that contain $L(v)$ and $R(v)$. Then $\mathcal{S}_e = \mathcal{S}_{L(v)} \cup \mathcal{S}_{R(v)}$. We have a total of $O(K/\log n)$ sets, each consisting of $O(\log N)$ strings. The latter is the key feature that we exploit in order to index these small sets efficiently by resorting to Lemma 4.2. It is worth noting that $\mathcal{S}_e$ will not be constructed and indexed explicitly,

[4]Note that we require the function to be injective for prefixes of strings in $\mathcal{S}$ not $\mathcal{S}_{\mathsf{top}}$.
[5]Recall that in the Weak-Prefix Search Problem we are searching under the assumption that P is a prefix of at least one string in $\mathcal{S}$.



(a) $\mathcal{S}_{\text{top}}$ (b) Patricia trie of $\mathcal{S}_{\text{top}}$

Fig. 3. The figure reports (in black) the strings in set $\mathcal{S}_{\text{top}}$ selected from $\mathcal{S}$ with groups of size 3 and the corresponding patricia trie PT_{top}.

rather we will index the sets $\mathcal{S}_{L(v)}$ and $\mathcal{S}_{R(v)}$ individually, and keep two pointers to each of them for every edge e. This avoids duplication of information and some subtle issues in the storage complexities, but poses the problem of how to weak-prefix search in $\mathcal{S}_e$ which is only virtually available. The idea is to search in $\mathcal{S}_{L(v)}$ and $\mathcal{S}_{R(v)}$ individually, and two cases may occur. Either we find that the range of $\mathcal{S}$'s strings prefixed by P is totally within one of the two sets, and in this case we return that range; or we find that the range includes the rightmost string in $\mathcal{S}_{L(v)}$ and the leftmost string in $\mathcal{S}_{R(v)}$, and in this case we merge them. The correctness comes from the properties of trie's structure and the first-level search, as one can prove by observing that the trie built over $\mathcal{S}_{L(v)} \cup \mathcal{S}_{R(v)}$ is equivalent to the two tries built over the two individual sets except for the rightmost path of $\mathcal{S}_{L(v)}$ and the leftmost path of $\mathcal{S}_{R(v)}$ which are merged in the trie for $\mathcal{S}_e$. This merge is not a problem because if the range is totally within $\mathcal{S}_{R(v)}$, then the dominating node is within the trie for this set and thus the search for P would find it by searching either $\mathcal{S}_{R(v)}$ or $\mathcal{S}_e$. Similarly this holds for a range totally within $\mathcal{S}_{L(v)}$. The other case comes by exclusion, so the following lemma makes it possible to establish the claimed I/O and space bounds.

LEMMA 4.2. *Let $\mathcal{S}_i$ be a set of $K_i = O(\log N)$ strings of total length at most N. The Patricia trie of $\mathcal{S}_i$ can be represented by requiring $\log \binom{\mathsf{Trie}(\mathcal{S}_i)}{t_i - 1} + O(K_i)$ bits of space so that the blind search of any pattern P requires $O((\log K_i)/B + \mathsf{Scan}(P))$ I/Os, where t_i is the number of nodes in the trie of the set $\mathcal{S}_i$.*

PROOF. Let PT be the patricia trie of $\mathcal{S}_i$. The patricia trie is formed by three different components: the structure of the tree of $O(K_i)$ nodes, the first symbol of the label of each edge, and the lengths of the (labels on the) edges.

The structure of the PT is represented by using balanced parenthesis in $O(t_i) = O(K_i) = O(\log N)$ bits. In correspondence with the open parenthesis of a node we also write the symbol on the edge from that node to its parent. Once this representation is in memory, accessing the structure of the tree and symbols is for free since the whole representation fits in a constant number of memory blocks for any possible value of B.[6]

Representing the lengths on the edges is more complex. The idea is to divide edge-lengths into two groups: group L contains the length of the edge (v, u) iff $|\mathsf{string}(u)| \leq K_i \log N$ and group H contains the remaining lengths. Intuitively, L contains the edge-length required to search for the first $K_i \log N$ symbols (i.e., bits) of any pattern P while lengths in H are required to complete the search for patterns that are longer than $K_i \log N$. Observe that if the lengths are presented in level-wise order and we have access to the PT structure and its edge-symbols, a pattern can be searched for by first scanning lengths in L, and then, if the pattern is longer than $K_i \log N$, by continuing with the scanning of H. The lengths in L and H are represented with the Elias-Fano solution of Theorem 2.1. Accessing the values of this representation sequentially can be done cache-obliviously with a simple scan. The space usage of these two Elias-Fano representations is in the space bound of Lemma 4.2. Indeed, we observe that we have overall $t_i - 1$ edge-lengths that sum up to $\mathsf{Trie}(\mathcal{S}_i)$. A single Elias-Fano representation for all the edge-lengths would require $\log \binom{\mathsf{Trie}(\mathcal{S}_i)}{t_i - 1} + O(K_i)$ bits according to Theorem 2.1. Inequality in Lemma 2.3 guarantees that we achieve the same upper bound even if we divide these lengths into the two (non-overlapping) groups L and H.

Regarding the query complexity, the scanning of L requires $O(1 + K_i(\log K_i + \log\log N)/(B \log N))$ I/Os, since it contains at most K_i lengths representable with $O(\log K_i + \log\log N)$ bits each. Observe that this bound is $O(1 + (\log K_i)/B)$ because if $K_i = O(\log N/\log\log N)$ then the whole L fits in one memory word and the bound is $O(1)$ I/Os; otherwise $K_i = \Omega(\log N/\log\log N)$ and the bound is $O(1 + (\log K_i)/B)$ I/Os. Similarly, scanning H requires $O(1 + K_i/B)$ I/Os, since it contains at most K_i lengths representable with at most $\log N$ bits each. Observe that H is scanned only for patterns P of length $p = \Omega(K_i \log N)$ bits and, thus, scanning the pattern costs $O(\mathsf{Scan}(P)) = \Omega(1 + K_i/B)$. This concludes the proof of Lemma 4.2. □

To conclude the proof of Theorem 4.1, we distinguish two cases based on the value of K. If $K = O(\log N)$, we do not use the first level since Lemma 4.2 with $K_i = K$ already matches the bounds in Theorem 4.1. Otherwise $K = \Omega(\log N)$, and we use Lemma 4.2 to index each set $\mathcal{S}_i$ above with the key strategy of removing the prefix common to all the strings in each set because it is already stored in the first level. We call $\mathcal{S}'_i$ the set of strings obtained by stripping their common prefix. Searching P requires $O(\log_B K + \mathsf{Scan}(P))$ I/Os on the first level and $O((\log\log N)/B + \mathsf{Scan}(P)) = O(\log_B K + \mathsf{Scan}(P))$ I/Os on the second level. For the space usage, we observe that the first level requires $O(K)$ bits, and the second level requires $\sum_i (\log \binom{\mathsf{Trie}(\mathcal{S}'_i)}{t_i - 1} + K_i)$ bits according to Lemma 4.2. The latter space bound is at most $\sum_i (\log \binom{2\mathsf{Trie}(\mathcal{S}_i)}{t_i - 1}) + O(K)$ bits by noticing that $\sum_i K_i \leq \sum_i t_i \leq t = O(K)$, because each string of $\mathcal{S}$ belongs to at most one set $\mathcal{S}_i$, and that $\sum_i \mathsf{Trie}(\mathcal{S}'_i) \leq 2\mathsf{Trie}(\mathcal{S})$, because the edge-labels shared by the first string in $\mathcal{S}'_i$ and the last string in $\mathcal{S}'_{i-1}$ are repeated at most twice.

PROBLEM 2 (Full-Prefix Search Problem). *Let $\mathcal{S} = \{S_1, S_2, \ldots, S_K\}$ be a set of K binary strings of total length N. We wish to preprocess $\mathcal{S}$ in such a way that, given a pattern P, we can efficiently answer the query* $\mathsf{Prefix}(P)$ *which asks for the range of*

[6] Indeed, the cache-oblivious model is transdichotomous and, thus, any block size B is at least $\Omega(\log N)$ bits.

strings in $\mathcal{S}$ with P as prefix, the value $\perp$ is returned whenever P is not a prefix of any string in $\mathcal{S}$.

This is the classic prefix-search which requires to establish whether P is or is not the prefix of any string in $\mathcal{S}$. By combining Theorems 3.1 and 4.1 we obtain the following theorem.

THEOREM 4.3. *Given a set of $\mathcal{S}$ of binary strings of size K of total length N, there exists a data structure requiring $(1+\epsilon)\mathsf{LB}(\mathcal{S}) + O(K)$ bits of space that solves the* Full-Prefix Search Problem *for any pattern P with $O(\log_B K + (1+\frac{1}{\epsilon})\mathsf{Scan}(P))$ I/Os, where $\epsilon > 0$ is any constant.*

We use the solution of Theorem 4.1 to identify the highest node u from which descends the largest range of strings that have P as prefix. Then, we use Theorem 3.1 to check I/O-optimally whether $\mathsf{Retrieval}(u, |P|)$ equals P. The space usage of this solution is optimal up to a constant factor; the query complexity is almost optimal as it is unclear whether it is possible to remove the $\log_B K$ term and still maintain optimal space.

PROBLEM 3 (Longest-Prefix Search Problem). *Let $\mathcal{S} = \{S_1, S_2, \ldots, S_K\}$ be a set of K binary strings of total length N. We wish to preprocess $\mathcal{S}$ in such a way that, given a pattern P, we can efficiently answer the query $\mathsf{LPrefix}(P)$ which asks for the range of strings in $\mathcal{S}$ sharing the longest common prefix with P.*

This problem waives the requirement that P is a prefix of some of the strings in $\mathcal{S}$, and thus searches for the longest common prefix between P and $\mathcal{S}$'s strings. If P is a prefix of some strings in $\mathcal{S}$, then this problem coincides with the classic prefix-search. Possibly the identified lcp is the *null* string, and thus the returned range of strings is the whole set $\mathcal{S}$. We will prove the following result.

THEOREM 4.4. *Given a set of $\mathcal{S}$ of K binary strings of total length N, there exists a data structure requiring $(1+\epsilon)\mathsf{LB}(\mathcal{S}) + O(K)$ bits of space that solves the* Longest-Prefix Search Problem *for any pattern P with $O(\log_B K + (1+\frac{1}{\epsilon})\mathsf{Scan}(P))$ I/Os, where $\epsilon > 0$ is any constant.*

First we build the data structures of Theorem 3.1 with a constant ϵ' to be fixed at a later stage, in order to efficiently access prefixes of strings in $\mathcal{S}$ but also as a basis to partition the strings. It is convenient to observe this process on $\mathcal{T}_{\mathcal{S}}$. Recall that the data structure of Theorem 3.1 processes nodes of $\mathcal{T}_{\mathcal{S}}$ in DFS-order. For each visited node u, it encodes $\mathsf{string}(u)$ either by copying $\mathsf{string}(u)$ or by writing $\mathsf{label}(u)$. In the former case we say that u is marked. Let $\mathcal{S}_{\mathsf{copied}}$ be the set formed by the $\mathsf{string}(u)$ of any marked node u. The goal of a query $\mathsf{LPrefix}(P)$ is to identify the lowest node w in $\mathcal{T}_{\mathcal{S}}$ which shares the longest common prefix with P. We identify the node w in two phases. In the first phase we solve the query $\mathsf{LPrefix}(P)$ on the set $\mathcal{S}_{\mathsf{copied}}$ in order to identify the range of all the (consecutive) marked nodes $[v_l, v_r]$ sharing the longest common prefix with P. Armed with this information, we start a second phase that scans appropriate portions of the compressed representation R of Theorem 3.1 to identify our target node w.

In the following we say that a node u is smaller than a node v iff u precedes v in the DFS-visit of $\mathcal{T}_{\mathcal{S}}$, u is larger than v otherwise.

First phase. We index the set $\mathcal{S}_{\mathsf{copied}}$ with a two-level indexing approach as in Weak-prefix Search with the difference that we replace the index on the first level with the cache-oblivious solution in Brodal and Fagerberg [2006]. By the properties of the data structure in Theorem 3.1 we know that the total length of the strings in $\mathcal{S}_{\mathsf{copied}}$ is at most $\epsilon' \cdot \mathsf{LB}(\mathcal{S})$ and their number is at most $t \leq 2K$. This implies that applying the solution in Brodal and Fagerberg [2006] requires $c \cdot \epsilon' \cdot \mathsf{LB}(\mathcal{S}) + O((|\mathcal{S}_{\mathsf{copied}}|/\log N) \cdot \log |\mathcal{S}_{\mathsf{copied}}|) = c \cdot \epsilon' \cdot \mathsf{LB}(\mathcal{S}) + O(K)$ bits of space, for a constant c. We adjust the constants to match the

one in Theorem 4.4 by fixing $\epsilon' = \epsilon/c$, so to get $\epsilon \cdot \mathsf{LB}(\mathcal{S}) + O(K)$ bits as required. It is easy to see that this two-level index identifies the range of marked nodes $[v_l, v_r]$ which share the longest common prefix with P in $O(\log_B K + \mathsf{Scan}(P))$ I/Os.

Second phase. We aim to find the target node w starting from the range of marked nodes $[v_l, v_r]$. Consider the marked node u_l (resp. u_r) that precedes (resp. follows) v_l (resp. v_r) in the DFS-visit of $\mathcal{T}_\mathcal{S}$. The crucial property is that our target node w is either between u_l and v_l, or between v_r and u_r in the DFS-visit of $\mathcal{T}_\mathcal{S}$. By this property it follows that $\mathsf{string}(w)$ is represented in R either between u_l and v_l, or between v_r and u_r, with the additional observation that no other copied $\mathsf{string}()$ is in between them. Clearly w and its position are unknown to the algorithm. Thus, our solution first identifies the unmarked node w_l in the first interval which shares the longest common prefix with P, then the homologous node w_r in the second interval. Here we show the search for w_l, since the search for w_r is the same.

We first compute the longest common prefix (say, L) between P and $\mathsf{string}(u_l)$. Since $\mathsf{string}(u_l)$ is copied (marked) this costs $O(\mathsf{Scan}(P))$. Then, we move to the end of $\mathsf{string}(u_l)$'s encoding in R and we try to extend L by matching further symbols between P and the subsequent nodes z represented in R via the DFS-visit order. The procedure stops as soon as L cannot be further increased which means, by the DFS-visit order, that $\mathtt{lcp}(\mathsf{string}(z), P)$ becomes shorter than L. It is tempting to scan and match R up to the position corresponding to the (unknown) w_l. Unfortunately, this approach would only guarantee a complexity in $O(\mathsf{Scan}(\mathsf{string}(w_l)))$ I/Os which may potentially be $\omega(\mathsf{Scan}(P))$. Consider for example the case when $\mathsf{string}(w_l)$ is much longer than P and P is one of its proper prefixes. Thus, we use a slightly different strategy. We stop as soon as we have scanned the first $(2 + 2/\epsilon)|P|$ bits in R. Our claim is that the parent of w_l (denoted $\mathsf{p}(w_l)$) must be represented in this portion and it is correctly identified by our algorithm.

The proof is as follows. We have that $\mathsf{string}(w_l)$ and P have to share a common prefix longer than $\mathsf{string}(\mathsf{p}(w_l))$; otherwise $\mathsf{p}(w_l)$ would be our answer. Thus $\mathsf{string}(\mathsf{p}(w_l))$ is a proper prefix of P implying $|\mathsf{string}(\mathsf{p}(w_l))| < |P|$. Moreover, $\mathsf{p}(w_l)$ precedes w_l in the DFS-visit and, thus, precedes w_l in R. Since $\mathsf{string}(\mathsf{p}(w_l))$ is not copied, it is represented in the first $(2+2/\epsilon)\mathsf{string}(\mathsf{p}(w_l)) < (2+2/\epsilon)|P|$ bits of R because of the properties of $\mathtt{LPRC}$. The node $\mathsf{p}(w_l)$ is easily identified by our algorithm, since $\mathsf{p}(w_l)$ is the node represented in this region with the longest $\mathsf{string}()$ which is a proper prefix of P. Given $\mathsf{p}(w_l)$ we can identify w_l by just taking its left or right child depending on the next bit in P. The left child is the string immediately to the right of $\mathsf{p}(w_l)$ in R, and the right child can be identified by adding to the data structure of Theorem 3.1 an encoding of the structure of the underlying trie using additional $O(K)$ bits (see e.g., He et al. [2012]) and just one more I/O. At this point $\mathtt{lcp}(\mathsf{string}(w_l), P)$ can be computed by accessing the label of the edge $(\mathsf{p}(w_l), w_l)$ in R via a $\mathsf{Select}_1(E, w_l)$, taking $O(1)$ more I/Os. Finally by matching P and $\mathsf{string}(w_l)$ bits-by-bits we compute their lcp. The same is done for w_r and the comparison between the two lcps determines the node w we were searching for.

5. CONCLUSION AND FUTURE WORK

In this paper we proposed cache-oblivious optimal or almost-optimal solutions for three variants of the well-known prefix-search problem over strings. We foresee two main open questions. The first one concerns the design of a dynamic solution for these problems in compressed space. The second question asks to prove or disprove the existence of a solution for the Weak-prefix search and the Full-prefix search problems whose query complexity is $o(\log_B K) + O(\mathsf{Scan}(P))$ I/Os having a space usage close to the ones stated in Theorems 4.1 and 4.3.

References

Alberto Apostolico. 1985. The Myriad Virtues of Subword Trees. *Combinatorial Algorithms on Words* (1985), 85–96.

Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. 2010. Fast Prefix Search in Little Space, with Applications. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA)*. 427–438. DOI:http://dx.doi.org/10.1007/978-3-642-15775-2_37

Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. 2006. Cache-oblivious string B-trees. In *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. 233–242. DOI:http://dx.doi.org/10.1145/1142351.1142385

Gerth Stølting Brodal and Rolf Fagerberg. 2006. Cache-oblivious string dictionaries. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 581–590. DOI:http://dx.doi.org/10.1145/1109557.1109621

Andrej Brodnik and J. Ian Munro. 1999. Membership in Constant Time and Almost-Minimum Space. *SIAM J. Comput.* 28, 5 (1999), 1627–1640. DOI:http://dx.doi.org/10.1137/S0097539795294165

Erik D. Demaine, John Iacono, and Stefan Langerman. 2004. Worst-Case Optimal Tree Layout in a Memory Hierarchy. *CoRR* cs.DS/0410048 (2004).

Peter Elias. 1974. Efficient Storage and Retrieval by Content and Address of Static Files. *J. ACM* 21, 2 (1974), 246–260. DOI:http://dx.doi.org/10.1145/321812.321820

Robert M. Fano. 1971. On the number of bits required to implement an associative memory. *Memorandum 61, Computer Structures Group, Project MAC, MIT, Cambridge, Mass.* (1971).

Paolo Ferragina. 2013. On the weak prefix-search problem. *Theorectical Computer Science* 483 (2013), 75–84. DOI:http://dx.doi.org/10.1016/j.tcs.2012.06.011

Paolo Ferragina and Roberto Grossi. 1999. The String B-tree: A New Data Structure for String Search in External Memory and Its Applications. *J. ACM* 46, 2 (1999), 236–280. DOI:http://dx.doi.org/10.1145/301970.301973

Paolo Ferragina, Roberto Grossi, Ankur Gupta, Rahul Shah, and Jeffrey Scott Vitter. 2008. On searching compressed string collections cache-obliviously. In *Proceedings of the 27th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. 181–190. DOI:http://dx.doi.org/10.1145/1376916.1376943

Paolo Ferragina and Rossano Venturini. 2010. The Compressed Permuterm Index. *ACM Trans. Algorithms* 7, 1, Article 10 (2010), 10:1–10:21 pages. DOI:http://dx.doi.org/10.1145/1868237.1868248

Paolo Ferragina and Rossano Venturini. 2013. Compressed Cache-Oblivious String B-tree. In *Proceedings of 21th Annual European Symposium on Algorithms (ESA)*. 469–480. DOI:http://dx.doi.org/10.1007/978-3-642-40450-4_40

William Frakes and Ricardo Baeza-Yates. 1992. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall.

Edward Fredkin. 1960. Trie memory. *Communication of the ACM* 3, 9 (Sept. 1960), 490–499.

Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 2012. Cache-Oblivious Algorithms. *ACM Transactions on Algorithms* 8, 1 (2012), 4. DOI:http://dx.doi.org/10.1145/2071379.2071383

Dan Gusfield. 1997. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press.

Meng He, J. Ian Munro, and Srinivasa Rao Satti. 2012. Succinct ordinal trees based on tree covering. *ACM Transactions on Algorithms* 8, 4 (2012), 42. DOI:http://dx.doi.org/10.1145/2344422.2344432

Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. 2010. Compression, Indexing, and Retrieval for Massive String Data.. In *Procedings of the 21st Annual Symposium on Combinatorial Pattern Matching CPM*. 260–274. DOI:http://dx.doi.org/10.1007/978-3-642-13509-5_24

Richard M. Karp and Michael O. Rabin. 1987. Efficient Randomized Pattern-Matching Algorithms. *IBM Journal of Research and Development* 31, 2 (1987), 249–260. DOI:http://dx.doi.org/10.1147/rd.312.0249

Donald R. Morrison. 1968. PATRICIA - practical algorithm to retrieve coded in alphanumeric. *J. ACM* 15, 4 (1968), 514–534.

J. Ian Munro. 1996. Tables. In *Proceedings of the 16th Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. 37–42. DOI:http://dx.doi.org/10.1007/3-540-62034-6_35

Gonzalo Navarro and Veli Mäkinen. 2007. Compressed Full-text Indexes. *ACM Comput. Surv.* 39, 1, Article 2 (2007). DOI:http://dx.doi.org/10.1145/1216370.1216372

Ian H. Witten, Alistair Moffat, and Timothy C. Bell. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images* (second ed.). Morgan Kaufmann Publishers, Los Altos, CA 94022, USA. xxxi + 519 pages.