On Searching Compressed String Collections Cache-Obliviously

Paolo Ferragina*
Università di Pisa

Roberto Grossi†
Università di Pisa

Ankur Gupta
Butler University

Rahul Shah‡
Louisiana State University

Jeffrey Scott Vitter§
Purdue University

ABSTRACT

Current data structures for searching large string collections either fail to achieve minimum space or cause too many cache misses. In this paper we discuss some edge linearizations of the classic trie data structure that are simultaneously cache-friendly and compressed. We provide new insights on front coding [24], introduce other novel linearizations, and study how close their space occupancy is to the information-theoretic minimum. The moral is that they are *not just* heuristics. Our second contribution is a novel dictionary encoding scheme that builds upon such linearizations and achieves nearly optimal space, offers competitive I/O-search time, and is also conscious of the query distribution. Finally, we combine those data structures with cache-oblivious tries [2, 5] and obtain a succinct variant whose space is close to the information-theoretic minimum.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Arrays, Tables; E.4 [**Coding and Information Theory**]: Data compaction and compression.

General Terms

Algorithms, Design, Theory.

Keywords

B-tree, string searching, data compression, front coding, cache efficiency.

*Supported in part by Italian MIUR grants PRIN MainStream and Italy-Israel FIRB and by a Yahoo! Research grant.

†Supported in part by Italian MIUR grants PRIN MainStream and Italy-Israel FIRB.

‡Supported in part by National Science Foundation grant CCF–0621457.

§Supported in part by National Science Foundation grants IIS–0415097 and CCF–0621457.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PODS'08, June 9–12, 2008, Vancouver, BC, Canada.

Copyright 2008 ACM 978-1-60558-108-8/08/06 ...$5.00.

1. INTRODUCTION

Many modern applications of web search, information retrieval, and data mining boil down to a core set of search primitives on huge collections of data that contain (long) strings of variable length. (See [24, 18] for examples.) In this setting, compression and locality of access are key design features for developing efficient and scalable software tools. As the dataset grows in size, string queries must use more levels of the memory hierarchy. Each level of memory has its own performance features, but memory tends to get faster and smaller as it gets closer to the CPU. Therefore, the main goal in modern data structure design is to address *simultaneously* space succinctness (which fits more data in faster memory) *and* cache-friendly queries (which exploits faster memory). For large string collections, achieving these goals can provide huge performance gains.

A typical design choice to store a set of strings is the *trie* data structure [16] and its derivatives, which unfortunately cannot simultaneously guarantee the above two goals. B-tries and their variants (e.g., [1, 7, 5, 2] and references therein) achieve I/O-efficiency but cannot guarantee compressed space occupancy. On the other hand, recent compressed text indexes [21] achieve space succinctness but require many I/Os. As a result, the problem of basic string searching in hierarchical memory is open when we need to optimize both compression and locality of reference.

In this paper we investigate the string searching problem: Let $\mathcal{S}$ be a *sorted set* of K variable-length strings s_1, s_2, ... , s_K containing a total of N characters, drawn from an arbitrary alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$. The problem consists of storing $\mathcal{S}$ in a *compressed format* while supporting *fast access and search primitives* over its strings without scanning their compressed encodings entirely at each query. Given a pattern string $P[1, p]$, we identify the following fundamental queries:

- MEMBER(P) determines whether P occurs in $\mathcal{S}$.
- RANK(P) counts the number of strings in $\mathcal{S}$ that are lexicographically smaller than or equal to P.
- PREFIX_RANGE(P) returns all strings in $\mathcal{S}$ that are prefixed by P.
- SELECT(i) returns s_i, the ith lexicographically-ranked string in $\mathcal{S}$.

Other important operations on strings—such as successor, predecessor, and count—can be expressed as a combination of these queries and will not explicitly be dealt with here.

This is also the case for MEMBER, a simplification of PREFIX_RANGE. Notice that we do *not* need storage for string pointers or identifiers in $\mathcal{S}$, since string s_i can be implicitly and uniquely identified by the integer i, its rank in $\mathcal{S}$. Hereafter the term *string set encoding* refers to storing $\mathcal{S}$ in compressed format, and *string dictionary encoding* refers to a compressed and indexed storage of $\mathcal{S}$ that supports the above query operations. The string dictionary encoding can be seen as generalization to a set of strings of the fully-indexable dictionary (FID) introduced in [22].

In this paper we address both string set encoding and string dictionary encoding by revisiting the classic trie data structure and considering some *tree linearizations* that are cache-friendly and storable in compressed space. Informally, a *linearization* of a compacted trie is a sequence of its edge labels in some predetermined order. A widely known example of linearization is *front coding* (FC), in which the edge labels of the compacted trie are recorded in preorder. FC is widely used in practice to solve the string set encoding problem within the nodes of prefix B-trees [1] and other disk-conscious indexes for string collections [24]. Although the connection between FC and tries is intuitive [16], we introduce new insights on FC and other trie linearizations.

Our first contribution (see Section 3) is to quantify the difference between the space occupancy of FC and the information-theoretic *minimum number of bits*, denoted LT, needed to store $\mathcal{S}$ (see Theorem 2). Our analysis leads us to propose a related linearization, called *rear coding* (RC), which comes much closer to LT than does FC (see Theorem 3). FC and RC are not merely heuristics; they achieve provable space bounds..

We then move to the string dictionary encoding problem. FC is not searchable as is; it needs a bucketing strategy and some extra pointers that increase its space occupancy and query performance [24]. To overcome these limitations, the *locality-preserving front coding* (LPFC) scheme adaptively partitions the set of strings into blocks such that decoding any string of $\mathcal{S}$ takes optimal I/Os [2], but it incurs a (constant) increase in the space occupancy of FC. String searching still needs extra data structures and thus an additional overhead in space and query time. Compressed linearizations of labeled trees (and tries) [8, 9] or compressed dictionary data structures [10] do exist, but unfortunately they do not guarantee locality of access in supporting string queries over $\mathcal{S}$.

In this paper we go one step further in the design of a compressed string dictionary encoding by proposing a linearization of the compacted trie that, augmented by a few extra bits, can support fundamental string queries. To analyze the performance of our solutions, we use the *cache-oblivious model* [11], in which the computer is abstracted to consist of a processor and two memory levels: the internal memory of size M and the (unbounded) disk memory, which operates by reading and writing data by blocks of size B. These two parameters are *unknown* to algorithms, and thus algorithms cannot exploit knowledge of the values of M and B in their design; however, these parameters come into play during algorithm analysis. The advantage of cache-oblivious algorithms, originally designed for a two-level memory hierarchy, is that they *automatically and optimally tune* to hierarchies with arbitrarily many memory levels [11].

Our second main contribution is the design of a new trie linearization based on the centroid path decomposition of the compacted trie for $\mathcal{S}$ that supports I/O-efficient string queries and approaches LT plus $O(1)$ bits per string in $\mathcal{S}$. It achieves our twofold goal of space compression and cache-oblivious query access. (See Theorem 7 in Section 4.2.) Such a compressed and searchable dictionary encoding can be used to squeeze more strings into each node of a prefix B-tree, thus achieving a double advantage: It decreases both its height (because of the increase in the nodes' fan-out) and routing-time of string searches (because of its searchability).

Our linearization can easily be adapted to be sensitive to query distributions [19]. The search cost turns out to be proportional to $\log(1/q(x))$, where $q(x)$ is the probability that string $x \in S$ is accessed. Previous distribution-sensitive data structures collections are neither cache-oblivious nor succinct.

Finally, we propose a novel use of LPFC for sampling a suitable subset of $\mathcal{S}$, which can be *succinctly* stored in the *cache-oblivious* index of [5]. Combining this result with our string dictionary encoding, we obtain the first *succinct* data structure that simultaneously achieves space close to the information-theoretic minimum and cache-oblivious search performance (like COSB [2]). The space savings can be a multiplicative factor $\Omega(\log N)$. (See Theorem 9 in Section 5.) We plan to develop our ideas further for practical implementation with realistic string collections.

Overview of Paper. In the next section we compare our work with results in the literature. In Section 3 we analyze some edge linearizations of the classic trie data structure and compare them against a novel lower bound we derive for the storage cost of a (sorted) string set. In Section 4 we design two novel dictionary encoding schemes that build upon such linearizations and are efficient in the RAM model as well as in the cache-oblivious model. Finally, in Section 5 we show how to combine our scheme with LPFC and the cache-oblivious trie of [5] to obtain a succinct dictionary encoding whose space occupancy is closer to the information-theoretic minimum, and overall it is better than the cache-oblivious String B-tree of [2].

2. KNOWN RESULTS

The string dictionary problem is one of the fundamental problems in computer science, dating back to the 1960s when the Patricia Trie data structure [16] was introduced. It goes without saying that hashing is not suitable for the string dictionary problem, because it cannot support the PREFIX_RANGE(P) query. Tries and their variants [16, 4] are efficient for managing small dictionaries that fit into internal memory, but fail to provide efficient performance once the strings spread over multiple memory levels [24]. That shortcoming is the reason why String B-trees were introduced as an I/O-efficient data structure to manage large dictionaries of variable-length strings [7]. The String B-tree requires optimal $O(p/B + \log_B N)$ I/Os to search for a pattern P of length p in $\mathcal{S}$, but needs $\Theta(N \log \sigma + K \log N)$ bits of storage and the knowledge of B.

Recently cache-oblivious tries have been devised [5, 2] that still achieve the optimal $O(p/B + \log_B N)$ I/Os. Unfortunately, [5] is static and space inefficient, requiring $\Omega(N \log \sigma + K \log N)$ bits. Conversely, the *cache-oblivious string B-tree* (COSB) of [2] is dynamic and achieves the improved space of $(1 + \epsilon)|\texttt{FC}(\mathcal{S})| + K \log N$ bits. The novelty of this solution relies on the design of *locality-preserving front cod-*

ing (LPFC). LPFC is an encoding scheme for a dictionary of strings that, given a parameter ϵ, decodes any string $s \in \mathcal{S}$ in $O((1/B\epsilon)|s|)$ I/Os and requires $(1+\epsilon)|\mathtt{FC}(\mathcal{S})|$ bits of space. This adaptive scheme offers a clear space/time tradeoff in terms of the user-defined parameter ϵ, and its space occupancy depends ultimately on the effectiveness of the FCscheme.

The known results we discussed are mainly theoretical in flavor. In practice, software developers use (prefix) B-trees to achieve I/O-efficiency in the query operations, and they use the FCscheme to succinctly store the strings in the B-tree nodes [24]. This approach is efficient in practice, but it is not cache-oblivious (because it requires empirical tuning of the blocking parameter B), and it is not space optimal (because of the limitations of the FCscheme).

3. STRING SET ENCODINGS

In this section we first devise a lower bound LT on the bit complexity of encoding the *ordered* string set $\mathcal{S}$ by drawing inspiration from the structure of its compacted trie $\mathcal{T_S}$. Then we investigate the FClinearizations of $\mathcal{T_S}$ and compare its bit-space complexity with LT. The lesson we learn is that FC is *not just a heuristic* with good practical performance, but it is also a suitable string encoding with space occupancy intimately related to LT. Surprisingly enough, we take inspiration from FC and LTand propose a seemingly novel linearization—called *rear coding* (RC)—that comes closer to LT. In the following, we assume that $\mathcal{S}$ is prefix-free, namely, no two strings in $\mathcal{S}$ such that one is the prefix of another.

Let us start with the lower bound LT. Since the K strings have a total of N characters over $\Sigma = \{1, 2, \ldots, \sigma\}$, it may seem that we can obtain an information-theoretic lower bound by taking the logarithm of $\sigma^N\binom{N+K-1}{K-1}$. The term σ^N counts the number of possible sequences of N characters, and ther term $\binom{N+K-1}{K-1}$ counts all possible ways to split that sequence into K strings. However, this approach does not take into account the fact that $\mathcal{S}$ is *sorted*, nor does it take into account the similarity of adjacent strings, and hence it is not a lower bound.

Suppose instead we build a (uncompacted) trie to represent the strings in $\mathcal{S}$, such that the ith leaf (equivalently, the ith root-to-leaf path) in preorder identifies the string s_i. This trie can be viewed as a cardinal tree $\mathcal{C_S}$ whose edges are labeled with characters drawn from alphabet Σ. As noted in [3], the information-theoretic minimum to represent $\mathcal{C_S}$ is the logarithm of the number of cardinal trees, namely $\log\left(\binom{E\sigma+1}{E}/(E\sigma+1)\right)$ bits, where E is the number of edges in $\mathcal{C_S}$ and log denotes the base 2 logarithm. This can be bounded by

$$\log\left(\binom{E\sigma+1}{E}/(E\sigma+1)\right) \geq E\log\sigma + E - \Theta(\log(E\sigma)), \tag{1}$$

and approaches $E\log\sigma + E\log e - \Theta(\log(E\sigma))$ for increasing values of $\sigma \geq 2$.

Unfortunately, eqn. (1) is *not* a lower bound for storing $\mathcal{S}$ because of the unary nodes that arise when $\mathcal{C_S}$ is used to represent variable-length strings. The correct way to proceed is to observe that the trie $\mathcal{C_S}$ can be *compacted* so that no unary nodes exist (except possibly the root), thus turning edge labels in strings of variable length. The resulting tree is denoted by $\mathcal{T_S}$ and consists of K leaves, $k \leq K$ internal nodes, $t = k + K$ nodes overall, and $t - 1 \leq 2K$ edges. Note that E is equal to the total length of all edge labels in $\mathcal{T_S}$.

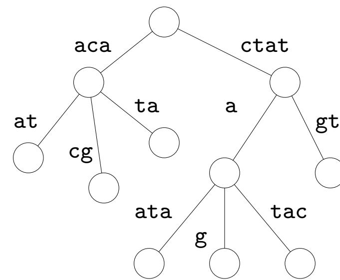

Figure 1: String set $\mathcal{S}$ = {acaat, acacg, acata, ctataata, ctatag, ctatatac, ctatgt}, and its compacted trie $\mathcal{T_S}$. Its encodings FC($\mathcal{S}$) = ⟨0, acaat, 3, cg, 3, ta, 0, ctaaata, 5, g, 5, tac, 4, gt⟩ and RC($\mathcal{S}$) = ⟨0, acaat, 2, cg, 2, ta, 5, ctaaata, 3, g, 1, tac, 4, gt⟩, obtained by traversing $\mathcal{T_S}$ in preorder.

Symbol	Definition
$\mathcal{S}$, $\mathcal{T_S}$	sorted set of strings and its representing trie
K	number of strings in $\mathcal{S}$
N	total number of characters in $\mathcal{S}$
Σ, σ	alphabet set and its size
FC($\mathcal{S}$)	Front coding of $\mathcal{S}$ (bitstream)
|FC($\mathcal{S}$)|	length in bits of FC($\mathcal{S}$)
LT	lower bound in bits on trie-based encoding of $\mathcal{S}$
E	number of characters (sum of edge labels) in $\mathcal{T_S}$
t	total number of nodess in $\mathcal{T_S}$
k	number of internal nodes in $\mathcal{T_S}$
P, p	Pattern and its length in number of characters

Table 1: Table of notations

An example of $\mathcal{T_S}$ is shown in Fig. 1.

To provide a lower bound for the storage complexity of $\mathcal{S}$, denoted LT($\mathcal{S}$) (or simply LT), we exploit the one-to-one correspondence between prefix-free ordered sets $\mathcal{S}$ and compacted tries $\mathcal{T_S}$, as follows.

1. Remove all $E - t + 1$ non-branching characters (if any) from the $t - 1$ edges of $\mathcal{T_S}$. Those characters can be arbitrarily chosen from alphabet Σ and freely distributed among the edges, without changing $\mathcal{T_S}$'s structure. Storing these characters needs $(E - t + 1)\log\sigma + \log\binom{E}{t-1}$ bits,[1] because the first term reflects the cost of storing those $(E - t + 1)$ characters, whereas the latter term accounts for encoding the possible ways of partitioning those non-branching characters among the $t - 1$ edges of $\mathcal{T_S}$ (possibly with empty partitions if some edge labels contain only branching characters).

2. The trie resulting from dropping the non-branching characters is actually a cardinal tree with no unary nodes. Since each of the k internal nodes has *at least* two children, the number of such cardinal trees is certainly at least the number of *binary* cardinal trees with $t = k + K = 2k + 1$ nodes. The number of bits required to represent a binary cardinal tree is at least $t + k\log\binom{\sigma}{2}$, since t bits are needed for encod-

[1]Strictly speaking, the latter term should be $\log\binom{E-1}{t-2}$. However, the difference between the two is at most a negligible additive factor of $\log N$ and is consumed by the second order terms.

ing the tree structure, and each pair of outgoing edges from an internal node may be labeled in $\binom{\sigma}{2}$ possible ways. Thus, $t + k \log \binom{\sigma}{2} = 2k \log \sigma + (2k+1) - k\big(1 + \log(\sigma/(\sigma-1))\big) \geq (t-1)\log\sigma$ bits are needed.

We have therefore proved the following.

THEOREM 1. *Encoding a prefix-free sorted string set $\mathcal{S}$ needs at least*

$$\mathtt{LT}(\mathcal{S}) = E \log \sigma + \log \binom{E}{t-1} \quad \textit{bits.} \tag{2}$$

This result can be used in two ways: One way is *negatively* to provide a lower bound to the storage complexity of any sorted string set $\mathcal{S}$; this lower bound is smaller than eqn. (1) because $\log \binom{E}{t-1} \leq E$, for any $t \geq 1$. Another way is *positively* to infer that the encoding of $\mathcal{T}_\mathcal{S}$ outperforms the encoding of $\mathcal{C}_\mathcal{S}$ because we strip off the unary nodes.

We now turn our attention to some interesting linearizations of $\mathcal{T}_\mathcal{S}$ and relate their storage complexity to LT. We start from the well known front coding scheme FC. It represents the string set $\mathcal{S} = \{s_1, s_2, \ldots, s_K\}$ as the sequence $\mathtt{FC}(\mathcal{S}) = \langle 0, s_1, n_2, s'_2, \ldots, n_K, s'_K \rangle$, where n_i is the length of longest common prefix (lcp) between s_{i-1} and s_i, and s'_i is the suffix of s_i remaining after the removal of its first n_i (shared) characters. The first string s_1 is represented in its entirety. FC is a well established practical method for encoding a string set [24]. Its main drawback is that decoding a string s_j might require the decompression of the entire prefix sequence $\langle 0, s_1, \ldots, n_j, s'_j \rangle$. Recently, [2] proposed a variant of FC, called *locality-preserving front coding* (LPFC), that, given a parameter ϵ, adaptively partitions $\mathcal{S}$ into blocks such that decoding any string s_j takes $O((1/\epsilon)|s_j|)$ optimal time and requires $(1+\epsilon)|\mathtt{FC}|$ bits of space. This adaptive scheme offers a clear space/time tradeoff in terms of the user-defined parameter ϵ. It is not at all clear just how much good FC and LPFC are with respect to the information-theoretic lower bound LT. We investigate this question below.

We start with a long-standing observation [16] about a relationship between $\mathcal{T}_\mathcal{S}$ and $\mathtt{FC}(\mathcal{S})$. (See Fig. 1.) The characters stored by FC in the suffixes s'_i can be obtained by one of two ways: (1) scanning the edge labels of $\mathcal{T}_\mathcal{S}$ in preorder; or (2) building $\mathcal{T}_\mathcal{S}$ incrementally via the insertion of each string s_i (in lexicographic order) and then taking suffix s'_i as the label of the new edge added to the current (compacted) trie. Thus, there is a bijective correspondence between trie edges and FC's suffixes s'_i, whose total length sums up to E. The net result is that $\mathtt{FC}(\mathcal{S})$ consists of three parts: (1) the edge labels, taking $E \log \sigma$ bits; (2) the label separators #, taking at least $\log \binom{E}{K-1}$ bits; and (3) the lcp lengths, taking at least $\sum_i \log(n_i + 1)$ bits. As a result, we get

$$|\mathtt{FC}| \geq E \log \sigma + \log \binom{E}{K-1} + \sum_{i=1}^{K} \log(n_i + 1). \tag{3}$$

If we consider a *binary array* of m bits with n bits set to 1 and $m - n$ bits set to 0 (or vice versa), there are known ways to compress it using the information-theoretic minimum of $\log \binom{m}{n} + O(\log m)$ bits of space [23]. Therefore, since $\sum_{i=1}^{K} n_i \leq N$, we can use these solutions to encode the values n_i in $\log \binom{N}{K-1} + O(\log N)$ bits, thus obtaining

$$|\mathtt{FC}(\mathcal{S})| \leq E \log \sigma + \log \binom{E}{K-1} + \log \binom{N}{K-1} + O(\log N). \tag{4}$$

The three-fold decomposition of FC's storage cost suggests some pathological situations when FC may be far apart from LT. To get them, it suffices to enlarge the cost of storing the lcp lengths in FC, a cost that is absent in LT. (See Theorem 1.) An illustrative example (for the case when K is a power of 2) consists of K binary strings eaching sharing the same K-long binary prefix, followed by a distinct $\log K$-bit suffix. This case is a *hard* case for FC, because it *repeatedly* represents the value $n_i = K$, thus incurring a significant penalty with respect to LT. To evaluate this penalty, we consider the relationship between $\mathcal{S}$ and $\mathcal{T}_\mathcal{S}$. Here $\mathcal{T}_\mathcal{S}$ consists of $t = 2K$ nodes ($k = K$ internal nodes plus K leaves) and $E = 3K - 2$ edge-labeling characters (bits). Using eqn. (3), we see that $|\mathtt{FC}|$ is at least $\approx K(3 + \log 3 + \log K)$ bits. On the other hand, $\mathtt{LT} \approx K(3 + \log 1.5)$ bits by Theorem 1, and therefore FC is a $\Theta(\log K)$ factor larger than the lower bound.

THEOREM 2. *For an arbitrary string set $\mathcal{S}$ of K strings of total length N, $\mathtt{LT} \leq |\mathtt{FC}(\mathcal{S})| \leq \mathtt{LT} + O(K \log N/K)$. There exist string dictionaries for which this upper bound is tight, and the storage gap between* LT *and* FC *is $\omega(\log K)$.*

The above example drives us to consider a different linearization, called *rear-coding* (RC), which is a simple variation of FC. RC *implicitly* encodes the lcp length n_i by specifying the length of the suffix of s_{i-1} to be removed from it to get the longest common prefix between s_{i-1} and s_i. (See Fig. 1 for an example.)

$$\mathtt{RC}(\mathcal{S}) = \langle 0, s_1, |s_1| - n_2, s'_2, \ldots, |s_{K-1}| - n_K, s'_K \rangle.$$

This change is *apparently small* but is nonetheless crucial to avoid *repetitive* encodings of the same longest common prefixes. RC mimics a preorder traversal of $\mathcal{T}_\mathcal{S}$): characters in the edge labels are counted just once, guaranteeing that $\sum_{i=1}^{K-1}(|s_i| - n_{i+1}) \leq E$. In other words, we are decomposing the (compacted) trie $\mathcal{T}_\mathcal{S}$ into K upward paths (one per leaf/string) and encoding their individual lengths. If we encode these lengths using a compressed binary array of E bits with $K - 1$ bits set to 1, we need $\log \binom{E}{K-1} + O(\log E)$ bits, and thus get overall

$$|\mathtt{RC}(\mathcal{S})| \leq E \log \sigma + 2 \log \binom{E}{K-1} + O(\log E), \tag{5}$$

where one of the $\log \binom{E}{K-1}$ terms comes from encoding the above binary array and the other comes from encoding the $K - 1$ delimiters for the $K - 1$ suffixes of total length E.

Comparing eqn. (4) and (5), RC overcomes the inefficiencies of FC since $E \leq N$. (In practice, $E \ll N$.) RC matches LT when $t \approx 2K$ (e.g., when $\sigma = 2$). Using our earlier hard case for FC, we see that $|\mathtt{RC}| \approx 2K(1 + \log 3)$ bits, which is within a factor of 1.5 from LT, and thus RC is asymptotically better than FC in this case. Since $K < t \leq 2K$, RC is never more than a constant factor worse than LT.

THEOREM 3. *For any string set $\mathcal{S}$,*

$$\mathtt{LT} \leq |\mathtt{RC}(\mathcal{S})| \leq \left(1 + \frac{4(1 + \log e)}{2 \log \sigma + 1} + o(1)\right) \mathtt{LT}. \tag{6}$$

The ultimate moral is that front and rear coding are *not merely heuristic* approaches to the (lossless) compression of string collections. Their performance is provably related to the information-theoretic minimum LT of the storage complexity of the string set $\mathcal{S}$. Our results also suggest that the novel RC may be asymptotically better than FC and never worse than a constant factor from LT. Another interesting bound is that of $E \log \sigma + \log \binom{E}{t-1} + \log \binom{2t}{t} = \mathtt{LT}(\mathcal{S}) + \Theta(K)$, descending from the result in [8]. It can be lower than eqn. (5) in some cases, and is comparable to the one we achieve in Theorem 7 where we achieve cache-obliviousness too.

4. STRING DICTIONARY ENCODINGS

Using the terminology for other data types [14, 20], we call a data structure for the string dictionary problem *succinct* if it takes space close to the information-theoretic lower bound LT and supports query operations (in this case, MEMBER, RANK, PREFIX_RANGE, and SELECT) with no *slowdown* with respect to classic tries.

In the rest of the paper, we will make use of the following well known tools for succinctly encoding and efficiently accessing (labeled) trees and binary arrays. We recall that an ordinal tree is a rooted tree with nodes of arbitrary degree, while a σ-ary cardinal tree is the extension of a binary tree to the case in which each node has σ children. (Empty children are explicitly stored as null pointers.)

THEOREM 4. [22] *We can encode a binary array $B[1, m]$ of n 1s and $m - n$ 0s (or vice versa) in $\log \binom{m}{n} + o(m)$ bits and perform Rank/Select operations in $O(1)$ time.*

THEOREM 5. [15, 13] *We can encode an (unlabeled) ordinal tree of t nodes in $2t + o(t)$ bits, and support sophisticated navigational operations in constant time. Supported operations include finding the parent, the ordinal position among its siblings, the ith child, the* DFS *rank, and the jth ancestor.*

Theorem 5 makes use of the DFUDS (depth first unary degree sequence) encoding, which we will employ later. Briefly, it traverses a tree in preorder, and for each visited node, it outputs the node degree in *unary* using the symbols (and). For example, ((() is for a node with three children, and) is for a leaf. An extra (is added to the beginning to obtain a sequence of $2t$ balanced parantheses. For example, the tree in Fig. 1 gives ((() ((()))) (() ((())))).

From this example, we observe that each node v (when traversed in preorder) is mapped to a distinct) in the sequence; if v is not the root, v is also mapped to a distinct (in the sequence when its parent was traversed. We therefore define the DFS rank (preorder number) of a node as the number of) symbols in the sequence that are to the left of its) symbol (inclusive). Its DFUDS rank is the number of (symbols in the sequence that are to the left of its (symbol (inclusive). The DFS rank and DFUDS rank of a node are not necessarily equal. Theorem 5 allows us to map DFS rank to DFUDS rank, and vice versa, in $O(1)$ time.

THEOREM 6. [22, 3] *We can encode a σ-ary cardinal tree of t nodes in $\log (\binom{t\sigma+1}{t}/(t\sigma + 1)) + o(t + \log \sigma)$ bits and support navigational operations in constant time. Supported operations include finding the parent, the degree, the ordinal position among its siblings, the child with label c, the ith child, and the* DFS *rank.*

Recent results [12] have achieved better little-oh terms in the space occupancy bounds of the above solutions.

4.1 A Simple Approach Using Cardinal Trees

Our first succinct dictionary data structure for an ordered set $\mathcal{S}$ of strings mixes the DFUDS encoding with a linearization of the edge labels in $\mathcal{T}_\mathcal{S}$. We call this simple mixed encoding $\mathcal{Z}$. We show that its space occupancy is close to $\mathtt{LT}(\mathcal{S})$, and its time efficiency in supporting dictionary queries in the RAM model is close to that of compacted tries. Unfortunately, this encoding does not *efficiently generalize* to a hierarchy of memory levels, which is a primary goal of this paper. Such a key issue will be fully addressed in the next sections.

Let w be a node of $\mathcal{T}_\mathcal{S}$ and let s_w be the string labeling the edge leading to w from its parent. We assume $s_w = \sigma_w \alpha_w$, where σ_w is the branching character of that edge and α_w is the (possibly empty) string of non-branching characters of that edge. We visit $\mathcal{T}_\mathcal{S}$ in pre-order, and for each visited node v (other than the root), we append the substring $\#\alpha_v$ to $\mathcal{Z}$ and drop α_v from the corresponding edge. At the end of the traversal, each edge of $\mathcal{T}_\mathcal{S}$ is labeled only with its branching character, and the string $\mathcal{Z}$ consists of E symbols, of which $t-1$ are #. Additionally, we can map $\mathcal{Z}$'s characters to $\mathcal{T}_\mathcal{S}$'s edge labels in constant time, once we are given their DFS rank since it corresponds to the rank of # in $\mathcal{Z}$.

We have enough information to (a) store $\mathcal{T}_\mathcal{S}$ with just its branching characters as a cardinal tree (Theorem 6), and (b) build compressed data structures for supporting Rank and Select operations over $\mathcal{Z}$ (Theorem 4), where # symbols in $\mathcal{Z}$ are interpreted as 1s and the remaining symbols as 0s. This scheme takes $2t + t \log \sigma$ bits for the cardinal tree, whereas the indexing and storage of $\mathcal{Z}$ takes $(E - t + 1) \log \sigma + \log \binom{E}{t-1} + o(E)$ bits, where $t \leq 2K$. Summing up, the total space occupancy is $4K + E \log \sigma + \log \binom{E}{t-1} + o(E) = \mathtt{LT} + 4K + o(E)$ bits.

Whenever we need to access an edge label, we compute the DFS rank of the destination node and then access the corresponding #-delimited substring in $\mathcal{Z}$. Since every subtree of $\mathcal{T}_\mathcal{S}$ is stored contiguously in $\mathcal{Z}$, its traversal for retrieving the pattern occurrences of PREFIX_RANGE(P) takes optimal time (even in the cache-oblivious setting). The total number of characters retrieved for the occurrences of pattern $P[1, p]$ is at most $p + E_{occ}$ symbols, since p is the length of the upward path from the root of this subtree to the root of $\mathcal{T}_\mathcal{S}$, and E_{occ} denotes the number of characters in the subtrie for the subset $occ \subseteq \mathcal{S}$ of strings having prefix P. (Thus $E_{occ} \leq \sum_{s \in occ} |s|$, but it is much less in practice). Furthermore, these strings are contiguous, since $\mathcal{S}$ is sorted. We therefore have (easily) proved the following result.

LEMMA 1. *Given trie $\mathcal{T}_\mathcal{S}$, there exists a data structure that requires $\mathtt{LT}(S) + 4K + o(E) = (1 + o(1))\mathtt{LT} + O(K)$ bits and supports* RANK(P), SELECT(i), *and* PREFIX_RANGE(P) *operations, respectively, in $O(p + \log \sigma)$, $O(|s_i|)$, and $O(p + E_{occ})$ time.*

It was just an exercise to derive this solution using existing succinct data structures. Surprisingly enough, this is the first *searchable* and *succinct* implementation of an encoding scheme for string dictionaries that is better than any known FC-based scheme both in space and time. (See Theorem 2 and eqn. (4).) Compared with the RCscheme, the space in

Lemma 1 is larger by the additive term $O(K)$. (See Theorem 3 and eqn. (5).) The extra space is neglibigle: theoretically, since $\texttt{LT} \geq E \log \sigma$ and $E \geq K$, we have $K = o(\texttt{LT})$ when σ goes to infinity; practically, we are just using four extra bits per string.

Our approach compares favorably with the static version of COSB [2] when it is used in the RAM model, because of many positive features: (1) it removes the dependance on the parameter ϵ (with respect to space occupancy and throughput cost of PREFIX_RANGE), (2) it achieves space close to optimal (up to the additive term $4K + o(E)$), and (3) it supports faster query operations (since this cost does not depend on the length of the predecessor and successor strings). However, unlike COSB, it does not *generalize* to a hierarchy of memory-levels because there is little locality of reference when traversing the trie structure and comparing the edge labels. This point is a major one that we address in the rest of the paper, where we introduce our main result.

4.2 A Cache-Oblivious Approach

In this section we propose a string dictionary encoding scheme, called PFC, which is succinct in space and cache-oblivious in supporting the queries of our string dictionary problem. PFC linearizes the trie $\mathcal{T}_\mathcal{S}$ by means of the *centroid path decomposition* technique that suitably *reshuffles* the strings in a way that they are succinctly encodable and cache-consciously searchable.

Centroid path decomposition of a tree. We shall assume without loss of generality that the root of $\mathcal{T}_\mathcal{S}$ has degree 1. (If not, we can add an artificial root.) For any node u in $\mathcal{T}_\mathcal{S}$, the child of u with the most leaves in its subtree is called u's *heavy* child. (Ties are broken arbitrarily.) For any node u, the *heavy path* (or *centroid path*) from u to a leaf is the downward path that traverses only heavy children. It is well known that $\mathcal{T}_\mathcal{S}$ can be partitioned into a set of K centroid paths, one per leaf (string), by a recursive procedure called *centroid path decomposition*. The procedure finds a centroid path of the root and then recursively invokes itself in each subtree hanging off of this centroid path.

FACT 1. *Any root-to-leaf path π in $\mathcal{T}_\mathcal{S}$ (i.e., a string $s \in \mathcal{S}$) shares edges with at most $\lceil \log K \rceil$ centroid paths.*

The centroid path tree of $\mathcal{T}_\mathcal{S}$. From the cardinal tree $\mathcal{T}_\mathcal{S}$ we construct the new ordinal tree $\mathcal{T}_S^c$, called the *centroid path tree*, in which each node corresponds to a distinct centroid path in $\mathcal{T}_\mathcal{S}$. Precisely, $\mathcal{T}_S^c$ has K nodes as there are so many centroid paths, one per leaf in $\mathcal{T}_\mathcal{S}$. We let π_u be the centroid path starting from a node u in $\mathcal{T}_\mathcal{S}$; we also use u to denote the corresponding node in $\mathcal{T}_S^c$, which will store information about the centroid path π_u it represents. Therefore, nodes in $\mathcal{T}_S^c$ are a *subset* of the nodes in $\mathcal{T}_\mathcal{S}$, namely, those originating a centroid path.

Let π_r denote the centroid path of $\mathcal{T}_\mathcal{S}$ beginning at its root r, and let $\mathcal{T}_1, \mathcal{T}_2, \ldots\ \mathcal{T}_{n(r)}$ represent the subtries hanging off of π_r, numbered in *lexicographical order of their leaves*. (See Fig. 2 for an example.) Let $u_1, u_2, \ldots, u_{n(r)}$ be the nodes at the root of these subtries, respectively. Every trie $\mathcal{T}_i$ is then recursively decomposed according to the centroid path beginning at its root u_i. As a result, $\mathcal{T}_S^c$ is recursively defined as the tree whose root is r, annotated with π_r, having children $u_1, u_2, \ldots, u_{n(r)}$, annotated with the paths resulting from the centroid path decomposition of $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{n(r)}$, respectively. When it is clear from the context, we will use the subtrie $\mathcal{T}_i^c$ and its root u_i interchangeably in $\mathcal{T}_S^c$, since both are conceptually represented by the same node in $\mathcal{T}_S^c$.

Consider a node v on a centroid path π_u of $\mathcal{T}_\mathcal{S}$. Its children in $\mathcal{T}_\mathcal{S}$ are partitioned into three groups (some possibly empty): G_L contains the children of v lying on the left side of π_u; G_M contains the child of v along π_u; G_R contains the children of v lying on the right side of π_u. This observation is crucial to understand the encoding that we are presenting next. We would like the nodes in $G_L \cup G_M \cup G_R$ to be stored contiguously in the encoding of $\mathcal{T}_S^c$ to permit efficient branching from v to its children. Fortunately, this is (partially) true in $\mathcal{T}_S^c$, because the nodes in G_L are contiguous in $\mathcal{T}_S^c$, and the same holds for those in G_R, while G_M is stored somehow as additional information in u (within the label π_u). Using this fact, we will show how to efficiently branch from a node to its children using $\mathcal{T}_S^c$.

The structure of $\mathcal{T}_S^c$ is suitable for cache-efficient access. As previously mentioned, the nodes of the ordinal tree $\mathcal{T}_S^c$ are in one-to-one correspondence with a subset of the nodes in the cardinal tree $\mathcal{T}_\mathcal{S}$. While $\mathcal{T}_\mathcal{S}$ can have height $\Theta(K)$, by Fact 1, the height of $\mathcal{T}_S^c$ is at most $\lceil \log K \rceil$. On the average $\mathcal{T}_\mathcal{S}$ has height $O(\log_\sigma K)$ [16, p.496] and so the average height of $\mathcal{T}_S^c$ is $O(\log_\sigma K)$ since it cannot exceed that of $\mathcal{T}_\mathcal{S}$. Another interesting property is that each root-node path π in $\mathcal{T}_\mathcal{S}$ is represented by a suitable root-node path $z_1, z_2, \ldots, z_h$ in $\mathcal{T}_S^c$ (where z_1 is its root and $h \leq \lceil \log K \rceil$). In other words, π can be obtained by concatenating suitable prefixes of the centroid paths π_{z_i} annotated in the nodes z_i in $\mathcal{T}_S^c$, for $1 \leq i \leq h$. A portion of the same path in both $\mathcal{T}_\mathcal{S}$ and $\mathcal{T}_S^c$ is described in the caption of Fig. 2. We therefore want to mimic a path traversal in $\mathcal{T}_\mathcal{S}$ using the above property on a suitable path of $\mathcal{T}_S^c$, by minimizing the random accesses to its encoding.

Given the interesting features above, we want to succinctly store $\mathcal{T}_S^c$ and its annotated centroid paths; this is the prelude to our linearization PFC of $\mathcal{T}_\mathcal{S}$.

Succinct representation of T_S^c. We represent T_S^c with a few succinctly indexed strings that encode its structure and content, so that the subsequent string dictionary queries can be implemented quickly. All those strings are generated by *visiting in pre-order* the tree T_S^c, and by *rearranging* the children and the labels of every visited node in a suitable way. As previously mentioned, the key difficulty here is that, for any node u in T_S^c, its children representing the roots of $\mathcal{T}_1^c, \mathcal{T}_2^c, \mathcal{T}_3^c ... \mathcal{T}_{n(u)}^c$ are numbered according to the lexicographical order of their leaves (i.e., a sort of DFS visit driven by π_u), whereas efficient branching out of π_u's nodes requires *shuffling* these subtrees first level-wise and then in left-to-right order (because the pattern search proceeds top-down in $\mathcal{T}_\mathcal{S}$, see Fig. 2). The following discussion is aimed at showing how to succinctly encode $\mathcal{T}_S^c$ while supporting cache-oblivious string-dictionaries queries.

Encoding the structure. We define the string $\mathcal{Y}_{\texttt{struct}}$ of balanced parentheses that encodes the structure of $\mathcal{T}_S^c$ as an ordinal tree, using DFUDS representation. Since $\mathcal{T}_S^c$ consists of K nodes, we can use Theorem 5 to store $\mathcal{Y}_{\texttt{struct}}$ in $2K + o(K)$ bits and take constant time to implement many sophisticated navigational operations over $\mathcal{T}_S^c$. We recall that we can map DFS rank to DFUDS rank, and vice versa, in constant time.

Encoding the labels. We need to suitably arrange the annotation π_u of every node u in $\mathcal{T}_S^c$ in a way that the resulting

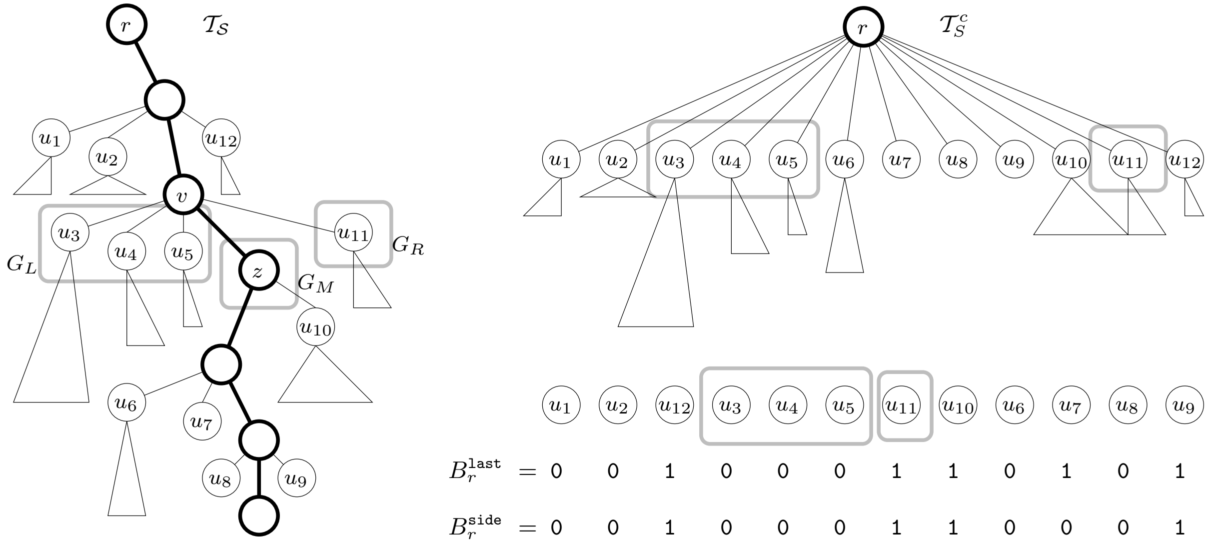

Figure 2: Left: A compacted trie $\mathcal{T}_\mathcal{S}$ in which the centroid path π_r from the root r is drawn in bold. Each node u_i is the root of the subtrie $\mathcal{T}_i$, where $1 \le i \le 12$. For node v, its groups are $G_L = \{u_3, u_4, u_5\}$, $G_M = \{z\}$, $G_R = \{u_{11}\}$. Top right: The recursive structure of the centroid path tree $\mathcal{T}_S^c$, where u_i recursively stores the centroid path decomposition of $\mathcal{T}_i^c$. Bottom right: The π_r-based level-wise ordering of r's children in $\mathcal{T}_i^c$, along with their binary arrays $B_r^{\mathtt{last}}$ and $B_r^{\mathtt{side}}$. Note that the path that starts from the root $r \in \mathcal{T}_\mathcal{S}$ and goes through the nodes v and z to u_{10}'s subtrie in $\mathcal{T}_\mathcal{S}$, is equivalently represented in $\mathcal{T}_S^c$ by the path that starts from $r \in \mathcal{T}_S^c$ and goes to its child u_{10}, and so on, plus the prefix of π_r (up to node z) annotated at r, the prefix of $\pi_{u_{10}}$ annotated at u_{10}, and so on.

strings can be encoded succinctly and can be orchestrated with $\mathcal{Y}_{\mathtt{struct}}$ to support fast string-dictionary queries over $\mathcal{T}_\mathcal{S}$. Hence, the basic question is how to branch from a centroid path π_u to its suitable subtrie.

Given a path π_u we use the symbol # to demarcate its edge labels in top-to-bottom order. Notice that every symbol # corresponds to a node in π_u, with potentially many subtries of $\mathcal{T}_\mathcal{S}$ hanging off it and lying to the left or to the right side of π_u. In $\mathcal{T}_S^c$, we *reshuffle* these children first level-wise and then in left-to-right order. This means that we traverse π_u downward, and for every encountered node (i.e., # symbol), we write the subtries that hang-off that node according to the left-to-right order of their leaves. We hereafter use the term π_u*-based level-wise ordering* to refer to this ordering of u's children in $\mathcal{T}_S^c$, and we drop π_u-based when the context is clear (see Fig. 2).

We now wish to encode the level-wise ordering in such a way that the encoding is succinct and can be related efficiently with $\mathcal{Y}_{\mathtt{struct}}$. We introduce two binary arrays $B_u^{\mathtt{last}}$ and $B_u^{\mathtt{side}}$ that keep track, for every subtrie hanging off of π_u, which # symbol it originates and which side (right or left) it lies on. We also introduce another string $B_u^{\mathtt{head}}$ that keeps track of the branching characters of the edges connecting π_u to the above subtrees. This fully encodes the structure and annotation of $\mathcal{T}_S^c$. More precisely, given the ith subtree hanging-off π_u (according to the level-wise ordering above), we set the following binary arrays.

- $B_u^{\mathtt{last}}[i] = 1$ iff that subtree is the *rightmost* one that hangs off some node in π_u.
- $B_u^{\mathtt{side}}[i] = 1$ iff that subtree lies to the right of π_u in $\mathcal{T}_\mathcal{S}$.
- $B_u^{\mathtt{head}}[i] = c$ iff c is the first character labeling the edge which connects that subtree to π_u.

We construct compressed data structures (Theorem 4) to support Rank and Select queries over $B_u^{\mathtt{last}}$ and $B_u^{\mathtt{side}}$, each containing $n(u)$ entries (bits), where $n(u)$ is the degree of u in $\mathcal{T}_S^c$. Building all these data structures at the global level, over all nodes $u \in \mathcal{T}_S^c$, they require $2K + o(K)$ space because $\mathcal{T}_S^c$ has K nodes. Furthermore, we notice that $B_u^{\mathtt{head}}$ can be partitioned in $n(u)$ subgroups, one per node along the centroid path π_u in $\mathcal{T}_\mathcal{S}$. (Hence, one per symbol # in the annotation of u.) Each subgroup refers to a node z on π_u (hence, symbol #), and it is formed by the first (hence, branching) symbols of the centroid paths hanging-off z and lying on the left or the right side of π_u in $\mathcal{T}_\mathcal{S}$. It is important to notice that each subgroup consists of *distinct* characters, and thus can be indexed by means of the dictionary data structure of [3, Thm 4.2] which takes $\log\sigma$ bits per indexed character. All these (subgroup) indexes can be concatenated to form the indexed $B_u^{\mathtt{head}}$, thus taking space which is $\log\sigma$-times the number of indexed characters. Over all nodes $u \in \mathcal{T}_S^c$, the space required by all these strings is $(t-1)\log\sigma + o(t)$ bits because we have one indexed character per edge in $\mathcal{T}_\mathcal{S}$, and there are $t-1$ edges in total.

We finally concatenate those data structures by visiting the nodes u of $\mathcal{T}_S^c$ in *pre-order*, so building the (indexed) strings:

- $\mathcal{Y}_{\mathtt{head}}$ which contains the $B_u^{\mathtt{head}}$-substrings;
- $\mathcal{Y}_{\mathtt{last}}$ which contains the $B_u^{\mathtt{last}}$ binary vector;
- $\mathcal{Y}_{\mathtt{side}}$ which contains the $B_u^{\mathtt{side}}$ binary vector;
- $\mathcal{Y}_{\mathtt{tail}}$ which is the string containing the tails of the centroid paths π_u (i.e., without their first character, which is already stored in $\mathcal{Y}_{\mathtt{head}}$).

Overall, $\mathcal{Y}_{\mathtt{side}}$ and $\mathcal{Y}_{\mathtt{last}}$ occupy $2K + o(K)$ bits (because $\mathcal{T}_S^c$ has K nodes), $\mathcal{Y}_{\mathtt{tail}}$ takes $(E-t+1)\log\sigma + \log\binom{E}{t-1} + o(E)$

bits (because of $t-1$ edges and thus branching characters), and $\mathcal{Y}_{\texttt{head}}$ takes $(t-1)\log\sigma + o(t)$ bits (because we have $t-1$ branching characters in $\mathcal{T}_{\mathcal{S}}$). We point out that the little-oh terms are due to the Rank and Select data structures that we build to support constant-time access to the individual data structures given the DFS ranks of their corresponding nodes in $\mathcal{T}^c_S$ (by Theorem 4). In summary, the total space (in bits) taken by all those strings is

$$4K + E\log\sigma + \log\binom{E}{t-1} + o(E) = \texttt{LT} \; + 4K + o(E).$$

It goes without saying that all those strings can be fused in one string, maintaining their individuality, by paying an extra additive $o(E)$ term, and giving rise to our PFC.

The reader may notice that both $\mathcal{Y}_{\texttt{struct}}$ and the other $\mathcal{Y}$ strings are built according to the DFS visit of $\mathcal{T}^c_S$. However, for every visited node u, the substructures of the (indexed) $\mathcal{Y}$ strings referring to u are arranged according to the level-wise order of the corresponding subtrees, but the ones in the string $\mathcal{Y}_{\texttt{struct}}$ are arranged according to the DFS order of those subtrees. The rest of this section shows how to orchestrate the path-navigation of $\mathcal{T}_{\mathcal{S}}$ with the (succinct) arrangements of the centroid path information in $\mathcal{T}^c_S$.

Orchestrating $\mathcal{Y}$ strings. String $\mathcal{Y}_{\texttt{struct}}$ is useful to navigate $\mathcal{T}^c_S$ via structure-based queries, and the other $\mathcal{Y}$ strings are useful to navigate $\mathcal{T}^c_S$ via pattern-based queries. To support all our string-dictionary queries, we need to go back and forth between these two navigational approaches.

Let us consider a node u in $\mathcal{T}^c_S$. By Theorem 5, we know that we can map the DFUDS rank of u in $\mathcal{T}^c_S$ to its DFS rank, and vice versa, in constant time. This is useful because we can jump in constant time from u's position in $\mathcal{Y}_{\texttt{struct}}$—its corresponding (symbol—to its data structures $B^{\texttt{side}}_u$, $B^{\texttt{last}}_u$, $B^{\texttt{head}}_u$, and $B^{\texttt{tail}}_u$ in the strings $\mathcal{Y}_{\texttt{side}}$, $\mathcal{Y}_{\texttt{last}}$, $\mathcal{Y}_{\texttt{head}}$, and $\mathcal{Y}_{\texttt{tail}}$.

Assume that v is the yth node on the centroid path π_u. The subtrees hanging-off v in $\mathcal{T}_{\mathcal{S}}$ may be divided into three groups according to π_u, as shown in Fig. 2. Given y, we can access the yth data structure in $B^{\texttt{head}}_u$ to find all characters heading the edges hanging off of v. Additionally, given y, $B^{\texttt{last}}_u$, $B^{\texttt{side}}_u$, we can determine the ordinal positions of G_L and G_R among the children of u in $\mathcal{T}^c_S$ (and thus get their DFUDS-ordering). It is enough to compute $a = Select(y-1)$ and $b = Select(y)$ in $B^{\texttt{last}}_u$, and then count the number q_0 of 0s in $B^{\texttt{side}}_u[1,a]$ (subtrees on the left of π_u) using Rank of 0s; and count the number q_1 of 1s in $B^{\texttt{side}}_u[1,b]$ (subtrees on the right of π_u) using Rank of 1s. Then, the value q_0 (resp. $n(u)-q_1$) indicates the starting *ordinal* position of G_L (resp. G_R) among the children of u in $\mathcal{T}^c_S$. Therefore, if $B^{\texttt{head}}_u$ provides us with the ordinal position of the children of the v where we need to jump, and we know the starting *ordinal* (DFS) position of G_L and G_R; then we also know the ordinal (DFS) position of that child in $\mathcal{T}^c_S$, and thus its DFUDS-position (by Theorem 5). This tool is useful to percolate $\mathcal{T}^c_S$ for pattern matching.

Dictionary queries in $\mathcal{T}_{\mathcal{S}}$. All the following operations are implemented by using string $\mathcal{Y}_{\texttt{struct}}$ to navigate $\mathcal{T}^c_S$ via structure-based queries, and the other $\mathcal{Y}$ strings to navigate $\mathcal{T}^c_S$ via pattern-based queries.

Select(i). We take a top-down approach starting at the root of $\mathcal{T}^c_S$ and navigating to its required child subtree recursively, till we find the ith leaf in $\mathcal{T}_{\mathcal{S}}$. The nodes in $\mathcal{T}^c_S$ correspond to leaves in $\mathcal{T}_{\mathcal{S}}$, however the pre-ordering of $\mathcal{T}^c_S$ does not give the pre-ordering of $\mathcal{T}_{\mathcal{S}}$'s leaves. Nonetheless, we are able to determine which child of the root r of $\mathcal{T}^c_S$ contains the ith leaf of $\mathcal{T}_{\mathcal{S}}$; then, we can recursively navigate from that child node until we find the ith leaf. Hence, we first find out whether the required leaf is on the left or the right of π_r. We do this by counting the number of children that correspond to centroid paths hanging off of the left of π_r; then we count the number of nodes descending from those (left) children in $\mathcal{T}^c_S$ via $\mathcal{Y}_{\texttt{struct}}$. We can easily obtain this information by the DFS rank in $\mathcal{T}^c_S$ of the rightmost among those children. If this number is larger than i, then the leaf we are interested in is to the left of π_r, otherwise it is to the right. At this point, the following observation is crucial: although the ith) does not correspond to the centroid path of s_i, because of the shuffling induced by the centroid path decomposition, the corresponding centroid path and the one of s_i belong to the same subtree of r. Therefore, if the required node goes on the left of π_r, then we find in which subtree of r the $(i+1)$th ')' symbol falls into; otherwise we check for ith) symbol. This child (subtree) can be found by using a level-ancestor query (with level =2) [15] issued from the ')' symbol we found.

This process continues recursively (by increasing the level for the LA query) until we find that the path we are at is exactly what we want (i.e., when the left or right query fails). Now, we reconstruct the string by traversing upwards in $\mathcal{T}^c_S$. Since the height of $\mathcal{T}^c_S$ is at most $\log K$, the time taken for this operation is $O(|s_i| + \log K)$.

Prefix-Range(P). We first check the path π_r corresponding to the root r of $\mathcal{T}^c_S$ and find the longest common prefix (lcp) between P and π_r. Notice that π_r is stored contiguously (in $\mathcal{Y}_{tail}$). If P is fully consumed, then we have a match. Otherwise, we look at the mismatching character $P[\texttt{lcp}+1]$. If it does not correspond to a symbol # on π_r, then we stop there and no match exists; otherwise, we have reached a node z in π_r and thus compare $P[\texttt{lcp}+1]$ with the character in π_r after the #. (It is a branching char of z.) If it is smaller (lexicographically) then we should follow a branching edge of z on the left of π_r; if it is larger, we should follow a branching edge of z on the right of π_r. The test on the existence of the branching char $P[\texttt{lcp}+1]$ at z is done by accessing one of the two indexing data structures in $B^{\texttt{head}}_z$ corresponding to the centroid paths hanging-off z and lying to the left/right of π_r. These data structures can be determined in constant time given the position of z in π_r (found during the scanning); additionally, it takes constant time to check the existence of a branching for $P[\texttt{lcp}+1]$ ([3, Thm 4.2]).[2] Once we know that this branching does exist, we also know its ordinal position among the children of z and thus we can derive the ordinal position of this child among the children of r according to the DFS order in constant time (as illustrated above). Finally, we jump to the representation of this node in $\mathcal{Y}_{\texttt{struct}}$ via ordinal tree operations in constant time (Theorem 5).

This process continues until we find a mismatch or until the entire pattern is consumed. In the latter case, we identify the group of subtrees (children) of the current node in $\mathcal{T}^c_S$ that provide the answer for the current prefix-range query. All these subtrees occur contiguously in the strings $\mathcal{Y}_{tail}$ and $\mathcal{Y}_{\texttt{head}}$, so that we can reconstruct the resulting

[2] If the branching character does not match, [3, Thm 4.2] does not tell the rank of the predecessor char of $P[\texttt{lcp}+1]$. But we do not need this!

strings in $O(p+E_{occ})$ time, where E_{occ} denotes the number of characters in the subtree storing the subset $occ \subseteq \mathcal{S}$ of strings prefixed by P.

RANK(P). We first navigate pattern P in $\mathcal{T}_S^c$ and reach a node v. It is easy to find the preorder rank of v in $\mathcal{T}_S^c$ by [15]. However, this pre-order rank is not the same as the rank of the leaf in $\mathcal{T}_S$ corresponding to P, which is the value we are interested in. Nonetheless, we can derive this value by subtracting from the preorder rank of v the number of all ancestors w of v in $\mathcal{T}_S^c$ such that π_v is on left of π_w in $\mathcal{T}_S$. It is easy to keep track of this number during the navigation of $\mathcal{T}_S^c$. In the case that the pattern is not fully consumed at v, we find out (in $\log \sigma$ time) which two children of v the pattern falls between. We then go to the end of the encoding of the subtree corresponding to the left-most child among the two, and report the rank as above.

Analysis in the cache-oblivious model. The layout of the $\mathcal{Y}$ strings involves few random accesses, corresponding to the operations in Theorems 4– 6. In all the above queries, the number of random accesses is a constant per centroid path visited and thus it is bounded by $O(\log K)$ overall (Fact 1). Furthermore, the edge labels that we retrieve (of total length $p+E_{occ}$) occur contiguously.

THEOREM 7. *Given a set $\mathcal{S}$ of K strings drawn from an alphabet of size σ, there is an encoding of $\mathcal{S}$ that takes* $\texttt{LT}(\mathcal{S})+4K+o(E) = (1+o(1))\texttt{LT} + O(K)$ *bits and supports* RANK(s_i), PREFIX_RANGE(P), *and* SELECT(i) *queries, respectively, in* $O(p/B+\log K+E_{occ}/B)$, $O(p/B+\log K)$, *and* $O(|s_i|/B+\log K)$ *I/Os.*

This result improves Theorem 1 because it guarantees cache-obliviousness in the cost of scanning the searched pattern P. We also notice that the space bound is asymptotically better than any FC-based scheme (and thus also LPFC), because it is larger than the minimum LT by just an additive term of $4K+o(E)$ bits. Furthermore, we restate here that term $\log K$ in Theorem 7 is actually $\log_\sigma K$ on the average (the height of $\mathcal{T}_S^c$). This term is actually a very small value for large sets of strings, and in practice, we expect this term to be small. For example, it is approximately 6 for a billion strings over an English alphabet ($\sigma \approx 60$).

A query-distribution-aware approach. Let us now assume that the leaves of $\mathcal{T}_S$ are weighted according to the *probability* $q(s)$ of occurrence of the corresponding string s in a user query. We devise a succinct and cache-oblivious solution whose worst-case term $O(\log K)$ is replaced by the *potentially smaller* term $O(\log 1/q(s))$, which reflects the information content of the queried string s. We start with leaves and give them weights $q(s)$ in $\mathcal{T}_S$; then we weight an internal node by the sum of weights of its children. Finally, we slightly modify our centroid path decomposition scheme to select *the child with the largest weight.* This scheme leads to the following result.

LEMMA 2. *Given a stream of queries for which we know its distribution in advance, the term $\log K$ in Theorem 7 can be transformed into $\log_2(1/q(s))$, where $q(s)$ is the probability of querying string $s \in \mathcal{S}$.*

In the next section we show how to combine Theorem 7 with LPFC of [2] to achieve improved worst-case bounds.

5. SUCCINCT CACHE-OBLIVIOUS STRING B-TREE

This structure consists of the two-level blocking scheme often used by software developers in practice [24], which has been improved in this section to make use of our cache-oblivious scheme plus some other specialties related to the LPFC-approach. We prove several results below.

First of all, we use the terminology of the LPFC-encoding scheme [2]. The LPFC scheme uses front compression, but periodically writes the whole string without relying on any previous string. Thus, when decoding a particular string, one does not have to go too far back. We call a string of $\mathcal{S}$ *copied* if it is either entirely copied by FC or by LPFC. The former means that its lcp is zero; the latter means that a string s has been copied because the cost of reconstructing s given $\texttt{FC}(\mathcal{S})$ was $\Omega(|s|/\epsilon)$, where ϵ is a suitable constant chosen at construction time. The key feature of LPFC was to show that the total length of the strings (copied or not) was upper bounded by $(1+\epsilon)\texttt{FC}(\mathcal{S})$, and the cost of decoding any string s was the optimal $O(|s|/\epsilon)$. This is an elegant trade-off between space occupancy and time complexity to decode any FC-compressed string, driven by the parameter ϵ. In this paper, we propose a novel use of LPFC as a *sampling* tool to derive a more succinct and cache-obliviously searchable dictionary encoding scheme.

We first show how to create a sampled set of strings. We set $\epsilon = 1$, construct $\texttt{LPFC}(\mathcal{S})$, and mark all the copied strings. Next, partition $\mathcal{S}$ into contiguous blocks of $\log^2 K$ strings each. We call a block *valid* if it contains at least one copied string. From any valid block, we select the minimum-length string and insert it in the set $\mathcal{S}'$. Since each selected string is no longer than any copied string of its block, we can prove that the total length of the strings in $\mathcal{S}'$ is $O(\texttt{FC}/\log^2 K) = o(\texttt{LT})$. Now, we use the cache-oblivious trie of [5] on the strings of $\mathcal{S}'$, as a top-level index structure to route the string queries to their proper block. The space taken by this structure is $|\mathcal{S}'|\log|\mathcal{S}'| + S\log\sigma$ bits, where S is the total length of the strings in $\mathcal{S}'$. We noted above that $|\mathcal{S}'| = K/\log^2 K$ and $S = \texttt{FC}(\mathcal{S})/\log^2 K$, so the total space required by the top-level structure is $o(\texttt{LT})$ bits.

For the bottom-level data structure, we use Theorem 7 to store valid blocks; the other blocks are RC-encoded. This takes $E\log\sigma + 2\log\binom{E}{K-1} + O(\log E)$ bits (eqn. (5)). By simple algebraic arguments, one can show that $\log\binom{E}{K-1} = O(K)+o(E)$, and thus the bottom level takes $(1+o(1))\texttt{LT} + O(K)$ bits of space and, any string s in a non-valid block can be decoded in optimal $O(|s|/B)$ I/Os, since it is a non-copied string of LPFC. The use of RC instead of FC in the non-valid blocks is needed to ensure the above space bound, without slowing down the decoding performance.

Searching proceeds in the following way. We query the top-level index in $O(p/B+\log_B K)$ I/Os [5] and find a valid block b. Then, we search b using Theorem 7, thus either finding the result or determining that the searched string is larger than any other string in block b. This takes $O(p/B+\log\log K)$ I/Os, because b contains $\log^2 K$ strings. If the searched string is larger than any other string in block b, we perform a scan of the (non-valid) blocks following b (if any). These blocks are actually RC-encoded, because they do not contain any copied string, and since they are non-valid we have the guarantee that every string in those blocks can be decoded in optimal time. Therefore the search for P

takes $O((p+|succ(P)|)/B)$ I/Os, where $succ(P)$ refers to the smallest lexicographically ordered string in $\mathcal{S}$ larger than P.

THEOREM 8. *Given a set $\mathcal{S}$ of K strings, we can design a data structure that takes $(1+o(1))\mathtt{LT}(\mathcal{S})+O(K)$ bits of space and supports* PREFIX_RANGE(P)*,* RANK(s_i) *and* SELECT(i) *queries, respectively, in $O((p+|succ(P)|)/B+\log_B K+\log\log K+E_{occ}/B)$, $O((p+|succ(P)|)/B+\log_B K+\log\log K)$, and $O(|s_i|/B+\log_B K+\log\log K)$ I/Os.*

As we observed before, term $\log\log K$ is actually $\log_\sigma \log K$ on the average [16, p.496], which is negligible for large sets of strings and in practice. When $\log B = O(\log K/\log\log K)$ (which is true in practice for large data sets), we get the `COSB`-bound within optimal space.

In order to bound in the worst case the $\log\log K$ term by $\log_B K$, we modify the way the K_b strings in a (valid) block b of the bottom-level structure are stored. We compute the storage space $S(b)$ of our scheme for the block b (Theorem 7), and distinguish two cases:

1. If $S(b) \leq \log^4 K$, we use our scheme because we can obtain the $O(\log_B K)$ search bound for searching this structure. In fact, if $B > S(b)$, then everything fits in one page; otherwise, $B \leq S(b) = \text{polylog}(K)$, and so $\log\log K = O(\log_B K)$.
2. If $S(b) > \log^4 K$, the length of the edge labels E_b is $\Omega(K_b^2)$. Thus, we can afford to store the cache-oblivious blind trie of [5] to search within b in an optimal number of I/Os. This takes $O(K_b \log K_b)$ bits for the trie, plus the cost of storing the strings. The leaves of the trie point to the starting locations of the strings in the linear storage structure. If we use `LPFC` (with parameter ϵ) to store the strings in b, we need $(1+\epsilon)\mathtt{FC}(b)+O(K_b\log K_b) = (1+\epsilon+o(1))\mathtt{LT}(b)$ bits of space, because we have to consider the cost of the `lcps` and the trie (but both are bounded since $K_b \leq \sqrt{E_b}$). Any string s of block b from the search in the trie can be decoded in optimal $O(|s|/B)$ I/Os.

THEOREM 9. *Given a set $\mathcal{S}$ of K strings, we can design a data structure that takes $(1+\epsilon)\mathtt{LT}+O(K)$ bits of space and supports* PREFIX_RANGE(P)*,* RANK(s_i) *and* SELECT(i) *queries, respectively, in $O((p+|succ(P)|)/(B\epsilon)+\log_B K+E_{occ}/B)$, $O((p+|succ(P)|)/B+\log_B K)$, and $O(|s_i|/B+\log_B K)$ I/Os, where ϵ is a user defined parameter.*

This is better than the storage space of the cache-oblivious String B-tree [2] because it takes $(1+\epsilon)\mathtt{FC}+O(K\log N)$ bits and `LT` may be asymptotically smaller than `FC`, as we pointed out in Section 3.

6. REFERENCES

[1] R. Bayer and K. Unterauer. Prefix B-trees. *ACM Transactions on Database Systems*, 2(1):11–26, 1977.

[2] M. Bender, M. Farach-Colton, and B. Kuszmaul. Cache-oblivious string b-trees. In *Proc. ACM PODS*, 233–242, 2006.

[3] D. Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, and S. Rao. Representing trees of higher degree. *Algorithmica*, 43:275–292, 2005.

[4] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. ACM-SIAM SODA*, 360–369, 1996.

[5] G. S. Brodal and R. Fagerberg. Cache-oblivious string dictionaries. In *Proc. ACM-SIAM SODA*, 581–590, 2006.

[6] V. Ciriani, P. Ferragina, F. Luccio, and S. Muthukrishnan. A data structure for a sequence of string accesses in external memory. *ACM Transactions on Algorithms*, 3(1), 2007.

[7] P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[8] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. IEEE FOCS*, 184–193, 2005.

[9] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and searching xml data via two zips. In *Proc. WWW*, 751–760, 2006.

[10] P. Ferragina and R. Venturini. Compressed permuterm index. In *Proc. ACM SIGIR*, 535–542, 2007.

[11] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. IEEE FOCS*, 285–298, 1999.

[12] A. Golynski, R. Grossi, A. Gupta, R. Raman, and S. S. Rao. On the size of succinct indices. In *Proc. ESA*, LNCS 4698, 371–382, 2007.

[13] M. He, J. I. Munro, and S. S. Rao. Succinct ordinal trees based on tree covering. In *Proc. ICALP*, LNCS 4596, 509–520, 2007.

[14] G. Jacobson. Space-efficient static trees and graphs. In *Proc. IEEE FOCS*, 549–554, 1989.

[15] J. Jansson, K. Sadakane, and W. Sung. Ultra-succinct representation of ordered trees. In *Proc. ACM-SIAM SODA*, 575–584, 2007.

[16] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, second edition, 1998.

[17] P. Ko and S. Aluru. Optimal self-adjusting trees for dynamic string data in secondary storage. In *Proc. SPIRE*, LNCS 4726, 184–194, 2007.

[18] G. Manku, A. Jain, and A.-D. Sarma. Detecting near-duplicates for web crawling. In *Proc. WWW*, 141–150, 2007.

[19] K. Mehlhorn and A. K. Tsakalidis. Data structures. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, 301–342, 1990.

[20] J. I. Munro. Succinct data structures. *Electr. Notes Theor. Comput. Sci.*, 91(3), 2004.

[21] G. Navarro and V. Mäkinen. Compressed full text indexes. *ACM Computing Surveys*, 39(1), 2007.

[22] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. ACM-SIAM SODA*, 233–242, 2002.

[23] F. Ruskey. *Combinatorial Generation*, 2007. In preparation.

[24] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, second edition, 1999.