On Searching Compressed String Collections Cache-Obliviously

Paolo Ferragina* Roberto Grossi† Ankur Gupta‡ Rahul Shah§ Jeffrey Scott Vitter¶

ABSTRACT

Current data structures for searching large string collections are limited in that they either fail to achieve minimum space or they 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 storable in compressed space. The widely-known front-coding scheme [26] is one example of linearization; it is at the core of Prefix B-trees and many other disk-conscious compressed indexes for string collections. However, it is largely thought of as a space-effective heuristic without efficient search support.

In this paper, we introduce new insights on front-coding and 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. The second contribution of this paper engineers these linearizations to design a novel dictionary encoding scheme that 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 [6, 3] and obtain a succinct variant, whose space is close to the information-theoretic minimum.

1. INTRODUCTION

Many tasks at the core of modern web search, information retrieval, and data mining applications boil down to a sequence of search primitives on huge collections of data, containing (long) strings of variable length. (See [23, 26, 19] for some 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 require more and 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, these goals can represent huge performance improvements.

*Dipartimento di Informatica, Università di Pisa.
†Dipartimento di Informatica, Università di Pisa.
‡Department of Computer Science, Butler University.
§Department of Computer Science, Louisiana State U.
¶Department of Computer Science, Purdue University.

The typical choice to store a set of strings is the *trie* data structure [17] and its derivatives, which unfortunately cannot simultaneously guarantee the two issues above. B-tries and their variations (see e.g. [2, 9, 6, 3] and references therein) achieve I/O-efficiency but cannot guarantee compressed space occupancy. On the other hand, recent compressed text indexes [22] 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, formalized as follows. Let $\mathcal{S}$ be a *sorted set* of K strings $s_1, s_2, \ldots, s_K$ having variable length for 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 constant combination of these queries and will not be dealt with explicitly 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 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 [24].

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 and many other disk-conscious indexes for string collections [26]. Although the connection between FC and tries is largely intuitive [17], we introduce new insights on FC and other linearizations of compacted tries. 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 $\mathcal{S}$. Our analysis leads naturally to another linearization, called *rear coding* (RC), which is a simple variant of FC. Still RC comes much closer to LT than FC does—in other words, FC and RC are not merely heuristics.

We then move to the string dictionary encoding problem. It is known [26] that FC is not searchable as is, but it needs some bucketing strategy and some extra pointers that increase its space occupancy and query performance. To overcome these limitations, [3] proposed the *locality-preserving front coding* (LPFC), which adaptively partitions the set of strings into blocks such that decoding any string of $\mathcal{S}$ takes optimal I/Os, but it incurs in 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) that support powerful string queries [10, 11] do exist, but unfortunately they do not guarantee locality of access in querying $\mathcal{S}$.

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 may be annotated with a small amount of extra bits to support fundamental string queries. To analyze the performance of our solutions, we will use the *cache-oblivious model* [12], 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, which only know about the existence of the memory hierarchy: they cannot exploit the value of M and B in their design parameters, yet these parameters come into play only during algorithm analysis. As a consequence, cache-oblivious algorithms originally designed for a two-level memory hierarchy, *automatically tune* to hierarchies with many memory levels [12]. Our second main contribution is

• to design 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 $\mathcal{S}$. This achieves our twofold target on space compression and cache-oblivious query accesses. (See Theorem 6 in Section 4.2.) Such a compressed and searchable dictionary encoding could be used to squeeze more strings into each node of a Prefix B-tree, thus achieving a double advantage: i.e. decrease both its height (because of the increase in the nodes' fan-out) and routing-time of string searches (because of its searchability).

• We show that such a linearization has the further positive feature of efficiently managing a *flow of string queries* issued according to a *given* probability distribution $p(x)$, where $x \in \mathcal{S}$. This flexibility allows us, for instance, to exploit the access statistics in a server [1]. The search cost turns out to be proportional to $\log(1/p(x))$, for any $x \in \mathcal{S}$, which is the information content of x. This scenario is very well known in data structure design [20], and has led many authors to propose *weighted* search data structures whose time complexity depends on the probability distribution of their queries. All those results known for string dictionaries [7, 18] are neither cache-oblivious nor succinct. Our approach can be easily adapted to work in this setting, thus achieving compressed space, cache-oblivious behavior and distribution-awareness for string queries. (See Lemma 3 in Section 5.)

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 [6]. Combining this result with our string dictionary encoding, we present

• the first *succinct* data structure that simultaneously achieves space close to the information-theoretic minimum and cache-oblivious search performance (like COSB [3]). The space saving can be up to a multiplicative factor of $\Omega(\log N)$. (See Theorem 8 in Section 6.)

2. KNOWN RESULTS

The string dictionary problem is one of the fundamental problems in computer science, dating its results back to the 60s when the Patricia Trie data structure [17] 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. Trie and their variations [17, 5] 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 [26]. This failure is the reason why [9] introduced the String B-tree as a I/O-efficient data structure to manage large dictionaries of variable-length strings. The String B-tree requires optimal $O(P/B + \log_B N)$ I/Os to search for a pattern P in $\mathcal{S}$, but needs $\Omega(N/B)$ disk pages to store the strings in $\mathcal{S}$, and $\Theta(N \log \sigma + K \log N)$ overall bits of storage. Therefore, the String B-tree is space *inefficient* and its design depends on the a priori knowledge of the page size B.

Recently, cache-oblivious tries have been devised [6, 3] that overcome the problem above by achieving the optimal $O(P/B + \log_B N)$ I/Os, without requiring the a priori knowledge of the page size B. The advantage of these solutions is that they *automatically and optimally tune* to hierarchies with arbitrarily many memory levels [12]. Unfortunately [6] is static and space inefficient, requiring $\Omega(N \log \sigma + K \log N)$ bits. Conversely, the *cache-oblivious string B-tree* (COSB) of [3] is dynamic and achieves the improved space of $(1 + \epsilon)|\mathtt{FC}(\mathcal{S})| + K \log N$ bits. The novelty of this solution relies on the design of *locality-preserving front coding* (LPFC). LPFCis an encoding scheme for a dictionary of strings that, given a parameter ϵ, ensures the decoding of any string $s \in \mathcal{S}$ takes $O((1/B\epsilon)|s|)$ 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 pa-

rameter ϵ, and its space occupancy depends ultimately on the effectiveness of the FC-scheme, which we address in Section 3.

All those solutions are mainly theoretical in their flavor. In practice, software developers use (Prefix) B-trees to achieve I/O-efficiency in the query operations, and the FC-scheme to succinctly store the strings in the B-tree nodes [26]. The final result 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 FC-scheme).

3. STRING SET ENCODINGS

In this section, we first devise a lower bound LT on the bit complexity of encoding the string set $\mathcal{S}$ by drawing inspiration from the structure of its compacted trie $\mathcal{T}_\mathcal{S}$. Then, we investigate two linearizations of $\mathcal{T}_\mathcal{S}$ (FCand RC) and compare their bit-space complexity to 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 LT, and propose another linearization of $\mathcal{T}_\mathcal{S}$ (called RC) that comes closer to LT and seems novel. In the following, we assume that no two strings in $\mathcal{S}$ are one prefix of the other.

Let us start with the lower bound LT. Suppose 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 s_i. This trie can be viewed as a cardinal tree $\mathcal{C}_\mathcal{S}$ whose edges are labeled with characters drawn from alphabet Σ. (Recall that $\sigma = |\Sigma|$.) As noted in [4], the information-theoretic minimum to represent $\mathcal{C}_\mathcal{S}$ is the logarithm of the number of cardinal trees, namely $\log\left(\binom{\mathtt{E}\sigma+1}{\mathtt{E}}/(\mathtt{E}\sigma+1)\right)$ bits, where E is the number of edges in $\mathcal{C}_\mathcal{S}$ and log denotes the base 2 logarithm. This can be bounded by

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

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

Interestingly, eqn.(1) is *not* a lower bound for storing $\mathcal{S}$ because of the presence of the many unary nodes that arise when $\mathcal{C}_\mathcal{S}$ is used to represent variable-length strings. In fact, the trie $\mathcal{C}_\mathcal{S}$ can be *compacted* so that no unary nodes exist (except for the root), thus turning edge labels in strings of variable length. The resulting tree is denoted by $\mathcal{T}_\mathcal{S}$ 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 $\mathcal{T}_\mathcal{S}$. An example of a compacted trie is shown in Fig. 1.

To provide a lower bound for the storage complexity of $\mathcal{S}$, denoted $\mathtt{LT}(\mathcal{S})$ (or simply LT), we proceed as follows.

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

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 the binary cardinal trees is at least $t + k\log\binom{\sigma}{2}$, since t bits are needed for encoding the tree structure, and each pair of outgoing edges from an internal node may be labeled in $\binom{\sigma}{2}$ possible ways. Thus, $t + k\log\binom{\sigma}{2} = 2k\log\sigma + (2k+1) - k(1 + \log(\frac{\sigma}{\sigma-1})) \geq (t-1)\log\sigma$ bits to represent.

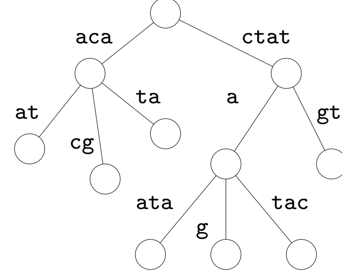

Figure 1: String set $\mathcal{S}$ = {acaat, acacg, acata, ctataata, ctatag, ctatatac, ctatgt}, and its compacted trie $\mathcal{T}_\mathcal{S}$. Its encodings $\mathtt{FC}(\mathcal{S}) = \langle$0, acaat, 3, cg, 3, ta, 0, ctaaata, 5, g, 5, tac, 4, gt$\rangle$ and $\mathtt{RC}(\mathcal{S}) = \langle$0,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

Given the bijection that exists between $\mathcal{S}$ and its compacted trie $\mathcal{T}_\mathcal{S}$, we arrive at the following theorem.

THEOREM 1. *Encoding string collection $\mathcal{S}$ needs at least*

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

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

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

PSfrag replacements

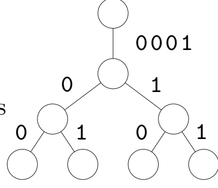

Figure 2: A "hard" example for FC.

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 [26]; 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, [3] 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 trade-off in terms of the user-defined parameter ϵ. In any case, it is not at all clear how much good is FC or LPFC with respect to the information-theoretic lower bound LT. We investigate this below.

We resort a long-standing observation [17] about a relationship between $\mathcal{T_S}$ and $\mathtt{FC}(\mathcal{S})$. (See Fig. 1). The characters stored by FC in the suffixes s'_i can be obtained by either (1) scanning the edge labels of $\mathcal{T_S}$ in preorder; or, (2) building $\mathcal{T_S}$ incrementally via the insertion of individual strings 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 $\mathtt{E}\log\sigma$ bits; (2) the label separators #, taking at least $\log\binom{\mathtt{E}}{K-1}$ bits; and (3) the lcp lengths, taking at least $\sum_i \log(n_i+1)$ bits via any prefix-free encoding of integers [8]. As a result, we see that

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

On the other hand, there exist solutions to compress a *binary array* of m bits with n bits set to 1 and $m-n$ bits set to 0 (or vice versa), using the information-theoretic minimum of $\log\binom{m}{n} + O(\log m)$ bits of space [25].

Therefore, since $\sum_{i=1}^{K} n_i \leq \sum_{i=1}^{K} |s_i| = N$, we can use these solutions to encode the values n_i in $\log\binom{N}{K-1} + O(\log N)$ bits, thus obtaining

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

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 is given in Fig. 2, where the set $\mathcal{S}$ consists of K binary strings (i.e. $\sigma = 2$) that share the same K-long binary prefix, followed by a distinct $\log K$-bit suffix. This is a *hard* case for FC, because it *repeatedly* represents the value $n_i = K$, thus incurring in a significant penalty with respect to LT. To evaluate this penalty, we consider the relationship between $\mathcal{S}$ and $\mathcal{T_S}$. Here, $\mathcal{T_S}$ consists of $t = 2K$ nodes ($k = K$ internal nodes plus K leaves) and $\mathtt{E} = 3K-2$ edge-labeling characters. 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 an $\Omega(\log K)$ factor larger than the lower bound.

Lemma 1. *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/or the storage gap between LT and FC is more than a factor $\Omega(\log K)$.*

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 lcps: characters in the edge labels are counted just once, guaranteeing that $\sum_{i=1}^{K-1}(|s_i| - n_{i+1}) \leq \mathtt{E}$. In other words, we are decomposing the (compacted) trie $\mathcal{T_S}$ 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 $\log\binom{\mathtt{E}}{K-1} + O(\log\mathtt{E})$ bits, and thus get overall

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

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

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

Theorem 2. *For any string set $\mathcal{S}$,*

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

Proof. Let's examine the ratio $|\mathtt{RC}|/\mathtt{LT}$ using Theorem 1 and eqn. (5), and upper-bound it by

$$\frac{\mathtt{E}\log\sigma + 2\log\binom{\mathtt{E}}{K-1} + O(\log\mathtt{E})}{\mathtt{E}\log\sigma + \log\binom{\mathtt{E}}{t-1}} \leq 1 + \frac{2\log\binom{\mathtt{E}}{K-1}}{\mathtt{E}\log\sigma + 1} + o(1).$$

Since $\binom{x}{y} \leq (xe/y)^y$ and $\mathtt{E} > K(1+1/\sigma)$, we can write:

$$|\mathtt{RC}|/\mathtt{LT} \leq 1 + o(1) + \frac{2(1+\frac{\log e}{1+1/\sigma})}{\log\sigma + 1/\mathtt{E}}.$$

□

The ultimate moral is that front- and rear-codings are *not just mere heuristic* approaches to the (lossless) compression of string collections. They are intimately related to the information-theoretic minimum LT on 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, which is realizable for large alphabet size σ. Another interesting bound is that of $\mathtt{E}\log\sigma + \log\binom{E}{t-1} + \log\binom{2t}{t} = \mathtt{LT}(\mathcal{S}) + \Theta(K)$, descending from the result in [10]. It can be lower than eqn. (5) in some cases and is comparable to that of the dictionary encoding in Theorem 6, but does not achieve cache-obliviousness in decoding.

4. STRING DICTIONARY ENCODINGS

Similar to the terminology for other data types [15, 21], 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—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 3. [24] *We can encode a binary array $B[1,m]$ with n 1s and $m-n$ 0s (or vice versa) in $\log\binom{m}{n} + o(m)$ bits, and support Rank/Select operations in $O(1)$ time.*

Theorem 4. [4, 16, 14] *We can encode an (unlabeled) ordinal tree of t nodes in $2t + o(t)$ bits, and support sophisticated navigational operations—find the parent, the ordinal position among its siblings, the ith child, the* DFS*-rank, the jth level-ancestor—in constant time.*

Theorem 4 makes use of the DFUDS (depth first unary degree sequence) encoding, which we will employ later on. Briefly, it traverses a tree in preorder and, for each visited node, 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 can observe the following. Each node v (when traversed in preorder) is mapped to a distinct) in the sequence; as a child (except 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). Note that, for a node, its DFS-rank and DFUDS-rank are not necessarily equal: to this end, Theorem 4 allows us to map DFS-rank to DFUDS-rank, and vice versa, in $O(1)$ time.

Theorem 5. [24, 4] *We can encode a σ-ary cardinal tree of t nodes in $\log\left(\binom{t\sigma+1}{t}/(t\sigma+1)\right) + o(t + \log\sigma)$ bits, and support navigational operations—find the parent, the degree, the ordinal position among its siblings, the child with label c, the ith child, the* DFS*-rank—in constant time.*

Recent results [13] have refined the space occupancy of the above solutions for binary arrays and trees by achieving better o-terms, since they are non-negligible in many cases.

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 $\mathcal{Z}$, and 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 compacted tries. Unfortunately, this encoding does not *efficiently generalize* to a hierarchy of memory levels, which is the main 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 #. Note that 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 5), and (b) build compressed data structures for supporting Rank and Select operations over $\mathcal{Z}$ (Theorem 3), 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 $(\mathtt{E}-t+1)\log\sigma + \log\binom{\mathtt{E}}{t-1} + o(\mathtt{E})$ bits, where $t \le 2K$. Summing up, the total space occupancy (in bits) is $4K + \mathtt{E}\log\sigma + \log\binom{\mathtt{E}}{t-1} + o(\mathtt{E}) = \mathtt{LT} + 4K + o(\mathtt{E})$.

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). Notice that the total number of characters retrieved for the occurrences of pattern $P[1,p]$ is at most $p + \mathtt{E}_{occ}$ symbols: because p is the length of the upward path from the root of this subtree to the root of $\mathcal{T}_{\mathcal{S}}$, and $\mathtt{E}_{occ}$ denotes the number of characters in the subtrie for the set $occ \subseteq \mathcal{S}$ of strings having prefix P (so $\mathtt{E}_{occ} \le \sum_{s\in occ}|s|$, but it is much less in practice). Note that these strings are contiguous since $\mathcal{S}$ is sorted. We therefore have (easily) proved the following result.

Lemma 2. *Given trie $\mathcal{T}_{\mathcal{S}}$, there exists a data structure that requires $\mathtt{LT}(S) + 4K + o(\mathtt{E}) = (1+o(1))\mathtt{LT}(\mathcal{S}) + O(K)$ bits and supports* Rank(P), Select(i), *and* Prefix_Range(P)

operations in $O(p+\log\sigma)$, $O(|s_i|)$, and $O(p+\mathtt{E}_{occ})$ time respectively.

It was just an exercise to derive this solution by 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 efficiency (see Lemma 1 and eqn. (4)). Compared to the `RC`-scheme, the space in Lemma 2 is larger by the additive term $O(K)$ (see Theorem 2 and eqn. (5)). Theorically, since $\mathtt{LT} \geq \mathtt{E}\log\sigma$ and $\mathtt{E} \geq K$, we have that $K = o(\mathtt{LT})$ when σ goes to infinity; practically, we are just using 4 extra bits per string.

Additionally, this approach compares favorably to the static version of the cache-oblivious string B-tree [3] 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 $4K + o(\mathtt{E})$), 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 due to lack of locality of references when traversing the trie structure and comparing the edge labels. This is the major point 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 procedure called *centroid path decomposition.* This is a recursive procedure that first finds a centroid path of the root, and then repeats the procedure in each subtree hanging off of this centroid path.

FACT 1. *Any root-to-leaf path π in $\mathcal{T}_\mathcal{S}$ (that is, 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 $\mathcal{T}^c_S$, called the *centroid-path tree*, in which each node corresponds to a distinct centroid path in $\mathcal{T}_\mathcal{S}$. Precisely, $\mathcal{T}^c_S$ has K nodes as there are so many centroid paths, one per leaf in $\mathcal{T}_\mathcal{S}$. We will denote by π_u the centroid path starting from a node u in $\mathcal{T}_\mathcal{S}$; and wlog, we will also denote by u the corresponding node in $\mathcal{T}^c_S$, storing in it information about the centroid path π_u it represents. Given this, nodes in $\mathcal{T}^c_S$ 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. We denote the subtries hanging off π_r by $\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_{n(r)}$, numbering them in accordance to the *lexicographical order of their leaves.* 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, $\mathcal{T}^c_S$ 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 $\mathcal{T}^c_i$ and its root u_i interchangeably in $\mathcal{T}^c_S$, since both are conceptually represented by the same node in $\mathcal{T}^c_S$.

Fig. 3 illustrates the recursive definition of $\mathcal{T}^c_S$. Pick a node v on a centroid path π_u of $\mathcal{T}_\mathcal{S}$. We observe that its children in $\mathcal{T}_\mathcal{S}$ are partitioned into three groups (some can be 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 $\mathcal{T}^c_S$ to permit efficient branching from v to its children. Fortunately this is (partially) true in $\mathcal{T}^c_S$, because the nodes in G_L are contiguous in $\mathcal{T}^c_S$, 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 $\mathcal{T}^c_S$.

The structure of $\mathcal{T}^c_S$ is suitable for cache-efficient access. As previously mentioned, the nodes of the ordinal tree $\mathcal{T}^c_S$ 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 $\Theta(K)$, the height of $\mathcal{T}^c_S$ is at most $\lceil \log K \rceil$ by Fact 1. (The maximum height in $\mathcal{T}^c_S$ is achieved when $\mathcal{T}_\mathcal{S}$ is a binary full balanced tree.) On the average $\mathcal{T}_\mathcal{S}$ has height $O(\log_\sigma K)$ [17, p.496] and so the average height of $\mathcal{T}^c_S$ 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 $\mathcal{T}^c_S$ (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 $\mathcal{T}^c_S$, for $1 \leq i \leq h$. A portion of the same path in both $\mathcal{T}_\mathcal{S}$ and $\mathcal{T}^c_S$ is commented in the caption of Fig. 3. We therefore want to mimic a path traversal in $\mathcal{T}_\mathcal{S}$ using the above property on a suitable path in $\mathcal{T}^c_S$, by minimizing the random accesses to the trie encoding.

Given the interesting features above, we want to succinctly store $\mathcal{T}^c_S$ and its annotated centroid paths as an equivalent representation of $\mathcal{T}_\mathcal{S}$. As will be clear next, this is the prelude to our linearization `PFC` of $\mathcal{T}_\mathcal{S}$.

Succinct representation of T^c_S. We represent T^c_S with a few succinctly indexed strings that encode trie structure and content, so that the subsequent string dictionary queries can be implemented fast. All those strings are generated by *visiting in pre-order* the tree T^c_S, and by *rearranging* the children and the labels of every visited node in a suitable way. As previously mentioned, the key difficulty

Sfrag replacements

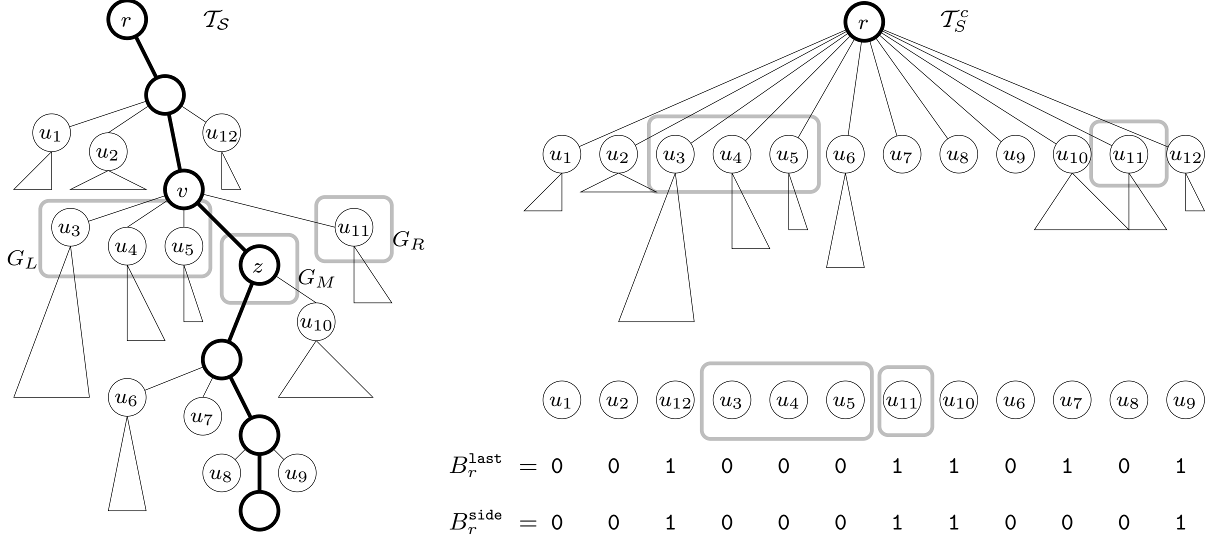

Figure 3: Left: A compacted trie $\mathcal{T}_{\mathcal{S}}$ in which the centroid path π_r from the root r is drawn in bold. Each node u_i is the root of the subtrie $\mathcal{T}_i$, for $1 \leq i \leq 12$. For node v, its groups are $G_L = \{u_3, u_4, u_5\}$, $G_M = \{z\}$, and $G_R = \{u_{11}\}$. 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 $B_r^{\texttt{last}}$ and $B_r^{\texttt{side}}$. 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.

here is that, for any node u in $\mathcal{T}_S^c$, 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 sub-trees first level-wise and then in left-to-right order (because the pattern search proceeds top-down in $\mathcal{T}_{\mathcal{S}}$, see Fig. 3). 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 first introduce 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 4 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 such a way that the resulting strings can be encoded succinctly and can be combined efficiently on-the-fly with $\mathcal{Y}_{\texttt{struct}}$ 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 will hereafter call this ordering of u's children in $\mathcal{T}_S^c$, π_u-based *level-wise* ordering, and drop π_u-based when this is clear from the context (see Fig. 3).

We wish now to encode the level-wise ordering in such a way that the encoding is succinct and can be related efficiently with $\mathcal{Y}_{\texttt{struct}}$. We introduce two binary arrays $B_u^{\texttt{last}}$ and $B_u^{\texttt{side}}$ which keep track, for every subtrie hanging-off π_u, from which of its # symbols it originates and from which side (right or left) it lies. We also introduce another string $B_u^{\texttt{head}}$ 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 $\mathcal{T}_S^c$. Precisely, given the ith subtree hanging-off π_u (according to the level-wise ordering above), we set the following binary arrays.

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

We construct compressed data structures (Theorem 3) to support Rank and Select queries over $B_u^{\texttt{last}}$ and $B_u^{\texttt{side}}$, 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 $B_u^{\texttt{head}}$ can be partitioned in $n(u)$ sub-groups, 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 [4, Thm 4.2] which takes $\log \sigma$ bits per indexed character. All these (subgroup) indexes can be concatenated to form the indexed $B_u^{\texttt{head}}$, thus taking space which is $\log \sigma$-times the number of indexed characters. We notice that 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:

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

Overall, $\mathcal{Y}_{\texttt{side}}$ and $\mathcal{Y}_{\texttt{last}}$ occupy $2K + o(K)$ bits (because $\mathcal{T}_S^c$ has K nodes), $\mathcal{Y}_{\texttt{tail}}$ takes $(\mathtt{E} - t + 1)\log \sigma + \log \binom{\mathtt{E}}{t-1} + o(\mathtt{E})$ 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 $\mathcal{T}_S^c$ (by Theorem 3). In summary, the total space (in bits) taken by all those strings is

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

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

The reader may notice at this point that both $\mathcal{Y}_{\texttt{struct}}$ and the other $\mathcal{Y}$s are built according to the `DFS`-visit of $\mathcal{T}_S^c$. However, for every visited node u, the sub-structures of the (indexed) strings $\mathcal{Y}$s referring to u are arranged according to the level-wise order of the corresponding subtrees, whereas the ones present in the string $\mathcal{Y}_{\texttt{struct}}$ are arranged according to the `DFS`-order of those subtrees. The rest of this section is therefore dedicated to show how to orchestrate the path-navigation of $\mathcal{T}_\mathcal{S}$ with the (succinct) arrangements of the centroid-path information in $\mathcal{T}_S^c$.

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

Let us consider a node u in $\mathcal{T}_S^c$. By Theorem 4, we know that we can map the `DFUDS`-rank of u in $\mathcal{T}_S^c$ to its `DFS`-rank, and vice versa, in constant time. This is useful because we can jump in constant time between u's position in $\mathcal{Y}_{\texttt{struct}}$—its corresponding `(` symbol—to its data structures $B_u^{\texttt{side}}, B_u^{\texttt{last}}, B_u^{\texttt{head}}$, and $B_u^{\texttt{tail}}$ in the strings $\mathcal{Y}$s.

Now, 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. 3. Indeed, given y, we can access the yth data structure in $B_u^{\texttt{head}}$ to find all characters heading the edges hanging-off v. Additionally, given y and $B_u^{\texttt{last}}$ and $B_u^{\texttt{side}}$, we can determine the ordinal positions of G_L and G_R among the children of u in $\mathcal{T}_S^c$ (and thus get their `DFUDS`-ordering). It is enough to compute $a = \mathit{Select}(y-1)$ and $b = \mathit{Select}(y)$ in $B_u^{\texttt{last}}$, and then count the number q_0 of `0`s in $B_u^{\texttt{side}}[1,a]$ (subtrees on the left of π_u) using Rank of 0s; and count the number q_1 of `1`s in $B_u^{\texttt{side}}[1,b]$ (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 $\mathcal{T}_S^c$. Therefore, if $B_u^{\texttt{head}}$ 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 children in $\mathcal{T}_S^c$, and thus its `DFUDS`-position (by Theorem 4). This tool is useful to percolate $\mathcal{T}_S^c$ 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 $\mathcal{T}_S^c$ via structure-based queries, and the other strings $\mathcal{Y}$s to navigate $\mathcal{T}_S^c$ via pattern-based queries.

Select(i). We take a top-down approach starting at the root of $\mathcal{T}_S^c$ and navigating to its required child subtree recursively, till we find the correct path. Note that we are looking for the ith leaf in $\mathcal{T}_\mathcal{S}$. The nodes in $\mathcal{T}_S^c$ correspond to leaves in $\mathcal{T}_\mathcal{S}$, however the pre-ordering of $\mathcal{T}_S^c$ 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 $\mathcal{T}_S^c$ 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. This can be done by counting the number of children which correspond to centroid paths hanging-off the left of π_r; and then counting the number of nodes descending from those (left) children via $\mathcal{Y}_{\texttt{struct}}$. This can be easily obtained by the `DFS`-rank 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 i-th `)` 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) [16] 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 $\mathcal{T}_S^c$. Since the height of $\mathcal{T}_S^c$ 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 $\mathcal{T}_S^c$, and find the longest common prefix ($\texttt{lcp}$) 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 lower (lexicographically) then we should follow a branching edge of z on the left of π_r; if it is higher, 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 $B_z^{\texttt{head}}$ 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 $P[\texttt{lcp}+1]$ ([4, 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 4).

This process continues recursively 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 $\mathcal{T}_S^c$ 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 strings in $O(p + \texttt{E}_{occ})$ time, where $\texttt{E}_{occ}$ denotes the number of characters in the subtree storing the set $occ \subseteq \mathcal{S}$ of strings prefixed by P.

RANK(P). For 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 [16]. 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 by 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 the node v, we find between which two children of v the pattern search should have gone in $\log\sigma$ time and we go to the end of the encoding of the subtree to the left (of where the pattern should have gone) and then report the rank as above.

Analysis in the Cache-Oblivious Model. We note that the layout of the string $\mathcal{Y}$ involves few random accesses, corresponding to the operations in Theorems 3– 5. 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 (of total length $p + \texttt{E}_{occ}$) to be retrieved occur contiguously. Hence, we get

THEOREM 6. *Given a set $\mathcal{S}$ of K strings drawn from an alphabet of size σ, there is an encoding of $\mathcal{S}$ that takes* $\texttt{LT}(\mathcal{S}) + 4K + o(\texttt{E}) = (1+o(1))\texttt{LT} + O(K)$ *bits and supports* PREFIX_RANGE(P), RANK(s_i), *and* SELECT(i) *in* $O(P/B + \log K + \texttt{E}_{occ}/B)$, $O(P/B + \log K)$, *and* $O(|s_i|/B + \log K)$ *I/Os, respectively.*

This result improves Theorem 2 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 $4K + o(\texttt{E})$ bits.

As we observed before, we restate here the important fact that term $\log K$ in Theorem 6 is actually $\log_\sigma K$ on the average (the height of $\mathcal{T}_S^c$): this is actually a very small value for large sets of strings. As a consequence, we expect in practice this term to be small—e.g. it is approximately 6 for a billion strings over an English alphabet ($\sigma \approx 60$).

In the following sections we aim at achieving an improvement in the *worst case*, and thus proceed in two distinct directions: either we deploy the knowledge of the distribution of the dictionary queries (Section 5), or we resort to the cache-oblivious trie of [6] and devise a novel use for the LPFC-scheme of [3] (Section 6).

5. A DISTRIBUTION-AWARE APPROACH

Another key property of our scheme is that it is flexible enough to be distribution-aware in supporting the string queries. In fact, let us assume that the leaves of $\mathcal{T}_\mathcal{S}$ are weighted according to the *probability* $p(s)$ of occurrence of the corresponding string s in a user query. We aim at devising a succinct and cache-oblivious solution whose worst-case term $O(\log K)$ is replaced by the *(potentially) smaller* term $O(\log 1/p(s))$, which reflects the information content of the queried string s. This scenario is very well known in data structural design [20], and has lead many authors to propose *weighted* search data structures whose time complexity depends on the probability distribution of their queries. All those results are neither cache-oblivious nor succinct. The elegance of our approach is that it can be easily adapted to work in this setting too.

The idea is that any leaf of $\mathcal{T}_\mathcal{S}$ is weighted according to the probability of occurrence of its corresponding string in the flow of dictionary queries. Then all internal nodes of $\mathcal{T}_\mathcal{S}$ are weighted by summing the weights of their descending leaves. Our centroid-path decomposition approach, described in the previous section, can then be applied by selecting now *the child with the largest weight*. As a result, the path corresponding to string s traverses $O(\log_2 1/p(s))$, instead of $O(\log_2 K)$ (see Fact 1), weighted-centroid paths of $\mathcal{T}_\mathcal{S}$. Given that this value influences the number of random memory accesses, we have proven the following.

LEMMA 3. *Given a stream of queries for which we know its distribution in advance, the term* $\log K$ *in Theorem 6 can be transformed into* $\log_2(1/p(s))$ *where* $p(s)$ *is the probability of querying string* $s \in \mathcal{S}$.

It is interesting to note that in the case of a prefix search for the pattern P is $\mathcal{S}$, the extra cost is $\log_2(1/p')$, where p' is

[2]If the branching character does not match, [4, Thm 4.2] does not tell the rank of the predecessor char of $P[\texttt{lcp}+1]$. But we do not need this!

the cumulative sum of the probability of occurrence of the dictionary strings having prefix P. So our encoding has the positive feature that the most frequently P occurs in the query sequence, the lower is the extra I/Os our search process induces with respect to the minimum $\Omega(P/B+\mathtt{E}_{occ}/B)$. Our string dictionary is the first that is succinct in space, cache-oblivious and conscious of the query distribution. In various practical settings this distribution can be estimated in advance, e.g. using query-logs in search engines (see e.g. [1] and refs therein).

6. SUCCINCT CACHE-OBLIVIOUS STRING B-TREE

This structure consists of the two-level blocking scheme often used by software developers in practice [26], 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 [3]. The LPFC scheme uses front compression but periodically writes the whole string without relying on any previous strings. Thus, when decoding a particular string, one does not have to go too far back. We call a string of $\mathcal{S}$ *copied* that is either entirely copied by FC or 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 $\mathtt{FC}(\mathcal{S})$ was $\Omega(|s|/\epsilon)$, where ϵ is a suitable constant. The key feature of LPFC was to show that the total length of the strings (copied or not) was upper bounded by $(1+\epsilon)\mathtt{FC}(\mathcal{S})$, 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 $\mathtt{LPFC}(\mathcal{S})$, 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 $O(\mathtt{FC}(\mathcal{S})/\log^2 K) = o(\mathtt{LT}(\mathcal{S}))$. Now, we use the cache-oblivious trie of [6] 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 $S = \mathtt{FC}(\mathcal{S})/\log^2 K$, so that the total space required by the top-level structure is $o(\mathtt{LT}(\mathcal{S}))$ bits.

As far as the bottom-level data structure is concerned, we use our structure of Theorem 6 for the valid blocks, the other blocks are RC-encoded. This takes $\mathtt{E}\log\sigma + 2\log\binom{\mathtt{E}}{K-1} + O(\log\mathtt{E})$ bits (eqn. (5)). By simple algebraic arguments, one can show that $\log\binom{\mathtt{E}}{K-1} = O(K) + o(\mathtt{E})$, and thus the bottom level takes $(1+o(1))\mathtt{LT}(\mathcal{S}) + O(K)$ 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 [6] and find a valid block b. Then, we search b using Theorem 6, 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 FC-encoded, because they do not contain any copied string, and since they are non-valid we are ensured 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.

THEOREM 7. *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 in $O((P + succ(P))/B + \log_B K + \log\log K + E_{occ}/B)$, $O((P+succ(P))/B + \log_B K + \log\log K)$, $O(|s_i|/B + \log_B K + \log\log K)$ I/Os, respectively.*

As we observed before, term $\log\log K$ is actually $\log_\sigma \log K$ on the average [17, p.496], which is actually a tiny value for large sets of strings and in practice. When $logB = O(\log K/\log\log K)$ (which means "always" in practice for large data sets), we get the COSB-bound with the optimal space occupancy. 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 6), and distinguish two cases:

1. If $S(b) \le \log^4 K$, we simply use our scheme because we can obtain $O(\log_B K)$ search bound for searching this structure. In fact if $B > S(b)$ then everything fits in one page, otherwise $B \le S(b) = \mathrm{polylog}(K)$ and so $\log\log K = O(\log_B K)$.
2. If $S(b) > \log^4 K$, we have that the length of the edge labels $\mathtt{E}_b$ is $\Omega(K_b^2)$. So we can afford to store the powerful cache-oblivious blind trie of [6] to search within b in an optimal number of I/Os. This takes $O(K_b \log K_b)$ bits for the blind trie plus the cost of storing the strings. The leaves of the blind-trie point to the starting locations of the strings in the linear storage structure. If we use LPFC to store the strings in b (with parameter ϵ), we have a space occupancy of $(1+\epsilon)\mathtt{FC}(b) + O(K_b\log K_b) = (1+\epsilon+o(1))\mathtt{LT}(b)$ because we have to consider the cost of the lcps and the blind trie (but both are bounded because $K_b \le \sqrt{\mathtt{E}_b}$). Any string s of block b, identified by a search in the blind trie can be decoded in optimal $O(|s|/B)$ I/Os.

Thus, we proved the following:

THEOREM 8. *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 in $O((P+succ(P))/(B\epsilon) + \log_B K + E_{occ}/B)$, $O((P+succ(P))/B + \log_B K)$, $O(|s_i|/B + \log_B K)$ I/Os, respectively, where ϵ is a user defined parameter.*

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

7. REFERENCES

[1] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *ACM SIGIR*, pages 183–190, 2007.

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

[3] M. Bender, M. Farach-Colton, and B. Kuszmaul. Cache-oblivious string b-trees. In *ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 233–242, 2006.

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

[5] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369, 1996.

[6] G. S. Brodal and R. Fagerberg. Cache-oblivious string dictionaries. In *SODA*, pages 581–590, 2006.

[7] 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.

[8] P. Fenwick. Universal codes. In *Lossless Compression Handbook*, pages 55–78. Boston, Academic Press, 2003.

[9] 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.

[10] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. 46th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 184–193, 2005.

[11] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and searching xml data via two zips. In *Proc. 15th International World Wide Web Conference (WWW)*, pages 751–760, 2006.

[12] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *IEEE Symposium on Foundations of Computer Science*, pages 285–298, 1999.

[13] A. Golynski, R. Grossi, A. Gupta, R. Raman, and S. S. Rao. On the size of succinct indices. In L. Arge, M. Hoffmann, and E. Welzl, editors, *ESA*, volume 4698 of *Lecture Notes in Computer Science*, pages 371–382. Springer, 2007.

[14] M. He, J. I. Munro, and S. S. Rao. Succinct ordinal trees based on tree covering. In *ICALP*, volume 4596 of *Lecture Notes in Computer Science*, pages 509–520, 2007.

[15] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[16] J. Jansson, K. Sadakane, and W. Sung. Ultra-succinct representation of ordered trees. In *Proc. 18th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2007.

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

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

[19] G. Manku, A. Jain, and A.-D. Sarma. Detecting near-duplicates for web crawling. In *WWW*, 2007.

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

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

[22] G. Navarro and V. Mäkinen. Compressed full text indexes. *ACM Computing Surveys*. To Appear.

[23] P. Raghavan. Information retrieval algorithms: A survey. *Proc. of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 11–18, 1997.

[24] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[25] F. Ruskey. *Combinatorial Generation*. 2007. In preparation.

[26] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1999.