Engineering a Textbook Approach to Index Massive String Dictionaries[⋆]

Paolo Ferragina, Mariagiovanna Rotundo, and Giorgio Vinciguerra

Department of Computer Science, University of Pisa, Pisa, Italy
{paolo.ferragina, giorgio.vinciguerra}@unipi.it
m.rotundo1@studenti.unipi.it

Abstract. We study the problem of engineering space-time efficient indexes that support membership and lexicographic (rank) queries on *very* large static dictionaries of strings.

Our solution is based on a very simple approach that consists of decoupling string storage and string indexing by means of a blockwise compression of the sorted dictionary strings (to be stored in external memory) and a succinct implementation of a Patricia trie (to be stored in internal memory) built on the first string of each block.

Our experimental evaluation on two new datasets, which are at least one order of magnitude larger than the ones used in the literature, shows that (i) the state-of-the-art compressed string dictionaries (such as FST, PDT, CoCo-trie) do not provide significant benefits if used in an indexing setting compared to Patricia tries, and (ii) our two-level approach enables the indexing of 3.5 billion strings taking 273 GB in less than 200 MB of internal memory, which is available on any commodity machine, while still guaranteeing comparable or faster query performance than those offered by array-based solutions used in modern storage systems, such as RocksDB, thus possibly influencing their future designs.

Keywords: String dictionary problem · Trie data structure · String compression · Algorithm engineering · Key-value store

1 Introduction

The string dictionary problem is a classic one in the string-matching field. It is defined on a set S of n strings of variable length, drawn from an alphabet Σ. The goal is to build an indexing data structure on S that efficiently answers a *membership query* on any query string $q \in \Sigma^+$, namely: "does $q \in S$?" Sometimes, the data structure is required to answer a more powerful query, which finds the *lexicographic position* of q within the sorted set S (aka the *rank* of q in S). The attention to this operation is motivated by the fact that the implementation of several other operations on S — such as the *prefix search*, which finds all the

[⋆] This paper has been accepted for publication in the Proceedings of the 30th International Symposium on String Processing and Information Retrieval (SPIRE 2023).

strings in S prefixed by q, and the *range search*, which finds all the strings in S that fall in a given query range — boil down to solving it.

In this paper, we assume that S is static, and thus it cannot be updated, but its total length N and number n of strings is so large that it has to be stored in slow storage, such as HDDs or SSDs. In fact, the recent explosion in the availability of massive string dictionaries in several applications — such as databases [43,33], bioinformatic tools [11], search engines [28], code repositories [13], and string embeddings (see e.g. [27,44]), just to name a few — has revitalised the interest in solving the problem in efficient time and space by taking into account the hierarchy of memory levels that are involved in their processing.

To solve the string dictionary problem, different approaches were proposed over the years in the literature. A trivial one consists of using an array of string pointers and deploying a binary search to answer queries, which causes random memory accesses and possibly I/Os. The classic one is the trie [22], a multiway tree that stores each string in S as a root-to-leaf path, and whose edges are labelled with either one character from Σ (the so-called uncompacted trie) or a substring from the strings in S (the so-called compacted trie). This historical solution has undergone over the years many significant developments that improved its query or space efficiency (see also [6] and refs therein) such as compacting subtries [6,40], using adaptive representations for its nodes [2,4,31], succinct representations of its topology [25,43], cache-aware or disk-based layouts [18,21], and even replacing it with learned models [16].

Among the most recent and performing variants of tries, which are pertinent to our discussion, we mention: ART [31], CART [42], Path Decomposed Trie (PDT) [24], Fast Succinct Trie (FST) [43], ctrie++ [40], and CoCo-trie [6]. According to the experimental results published in [6], we know that ART, CART, and ctrie++ are space inefficient and offer query times on par with the other data structures, which is a strong limitation in the massive-dictionary context we consider in this paper. The other three proposals — namely, FST, PDT, and CoCo-trie — stand out as the most interesting ones because they offer the best space-time trade-offs. Nevertheless, they incur three main "limitations": they are very complex to be implemented; their code is highly engineered, and thus difficult to be maintained or adapted to different scenarios (e.g., rank operations, adding satellite information); and, finally, they are designed to compress and index the string dictionary entirely in internal memory. In this paper, we ask ourselves whether this "sophistication" is really needed in practice to achieve efficient time and space performance on massive string dictionaries.

Inspired by the theoretical proposals of [12,17,19,21], our solution consists of decoupling string indexing and string storage, via a two-level approach [15]. The on-disk storage level compresses the sorted strings in S via rear coding [18] and partitions them into blocks of fixed size. The indexing level exploits a succinctly-encoded Patricia trie built on the first string of each block, so that it plays the role of a *router* for determining the block that possibly contains the query string q. Then, that block is fetched from the storage level and eventually scanned to search for the (lexicographic position of the) string q. Now, as long as the indexing level is small enough to fit in internal memory, we can solve the query in at most two disk I/Os without resorting to more complicated solutions [17,21]. Additionally, as for LSM-trees [38,33], decoupling indexing from storage allows us to support some dictionary updates, thus making our proposed solution suitable to manage datasets with high insertion rates too.

To perform our massive-scale experiments, we first notice that datasets from previous evaluations [6,24,43] are inadequate because their size is at most about 7 GB and the number of strings is at most 114 million. We, therefore, increase these sizes by at least an order of magnitude via two new datasets, one consisting of URLs from various Web crawls (272 GB, 3.5 billion strings) [8], the other consisting of filenames of source code files from the Software Heritage initiative (69 GB, 2 billion strings) [32].

Our first experimental finding is that sophisticated compressed string dictionaries (i.e., FST, PDT, CoCo-trie) are too complex for the indexing level, and they do not provide substantial space-time performance advantage compared to our well-engineered succinct Patricia trie, which is also much faster to construct.

Then, we show that our overall two-level approach based on succinct Patricia tries enables the indexing of the largest dataset with only at most 195 MB of internal memory (at least $\approx$1400× smaller than the dataset size). This small memory footprint allows dedicating much more memory to caching disk pages and this, in turn, determines a query efficiency that is comparable to or faster than the one offered by array-based solutions (which however take 5.2× more internal memory).

For these reasons, our two-level approach is a robust candidate for indexing massive string dictionaries, and it paves the way for further investigations and engineering, as we elaborate upon in the conclusions.

2 Background

A Patricia trie (PT) [36] for a string set S is derived from the trie of S by compacting each unary path into a single edge labelled with its first character, and by storing at each node the length of the (uncompacted) root-to-node path. Figure 1 shows an example of a PT built on a set of 8 strings.

Even if the PT strips out some information from the compacted trie, it is still able to support the search for the lexicographic position of a pattern $P[1, p]$ among a sorted sequence of strings, with the significant advantage (discussed below) that this search needs to access only one single string, and hence execute typically 1 I/O instead of the p I/Os potentially incurred by the traversal of the compacted trie due to accessing its (possibly long) edge labels. This algorithm is called *blind search* in the literature [17,15]. It is a little bit more complicated than prefix searching in classic tries, because of the presence of only one character per edge label. Technically speaking, blind search consists of three stages.

Stage 1: Downward traversal. Trace a downward path in the PT to locate a leaf l which points to one of the indexed strings sharing the longest common prefix (LCP) with P (see [17] for the proof). The traversal compares the characters of P with the single characters which label the traversed edges until either a leaf is reached or no further branching is possible. In this last case, we can choose l as any descendant leaf from the last traversed node; in our implementation, we will take the leftmost one.

Stage 2: LCP computation. Compare P against the string s pointed to by leaf l, in order to determine their LCP $\ell \geq 0$.

Stage 3: Upward traversal. Traverse upward the PT from l to determine the edge $e = (u, v)$ where the mismatched character $s[\ell + 1]$ lies. If $s[\ell + 1]$ is a branching character (and recall that $s[\ell + 1] \neq P[\ell + 1]$), then we determine the lexicographic position of $P[\ell + 1]$ among the branching characters of u. Say this is the ith child of u, the lexicographic position of P is therefore to the immediate left of the subtree descending from this ith child. Otherwise, the character $s[\ell + 1]$ lies within the edge e and after its first character, so the lexicographic position of P is to the immediate right of the subtree descending from edge e, if $P[\ell+1] > s[\ell+1]$, otherwise it is to the immediate left of that subtree.

The topology of the PT can be represented in several different ways, like, for example, using pointers or succinct encodings. Since we aim for space savings, we will use the latter and, in particular, the Level-Order Unary Degree Sequence (LOUDS) [25] and the Depth-First Unary Degree Sequence (DFUDS) [5]. Both encode the trie topology with a bitvector in which a node of degree d is represented by the binary string 1^d0. The difference is the order in which the nodes are visited and the corresponding binary strings are written in the bitvector: in level-wise left-to-right for LOUDS, and in preorder for DFUDS. For our implementation of DFUDS, we follow [37] and prepend 110 to the representation. For our implementation of LOUDS, we follow [43] and prepend no bits. See Figure 1 for an example of LOUDS and DFUDS representation.

Regarding compressing a lexicographically-sorted set of strings, two simple techniques are front coding [15,18] and rear coding [18]. Front coding represents each string with two values: an integer denoting the length of the LCP between the considered string and the previous one, and the remaining suffix of the considered string obtained by removing that LCP. If the string has not a predecessor, the LCP length is set to 0. In rear coding, the suffix is obtained in the same way as in front coding, but the integer represents the number of characters to remove from the previous string to obtain the longest common prefix. Rear coding may be more efficient than front coding since it does not encode the length of repeated prefixes [18,21].

3 Our Two-Level Approach

As anticipated in the Introduction, our string dictionary consists of two levels: a storage level (residing on disk), which consists of a sequence of fixed-size blocks where strings are stored in lexicographic order and compressed; and an indexing level (residing in internal memory), which consists of a succinctly-encoded Patricia trie (PT) that indexes the first strings of every block.

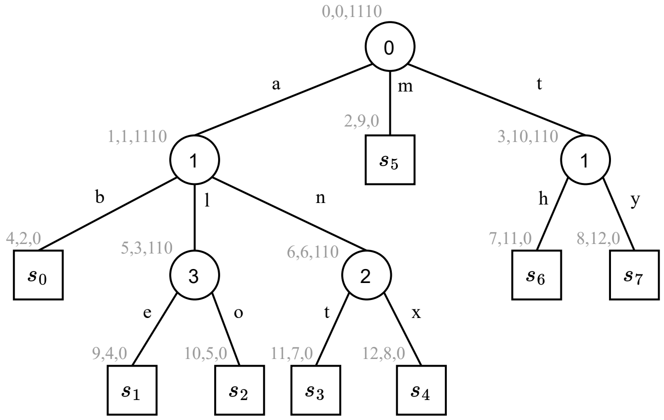
3.1 Storage Level

For the on-disk storage level, let us consider the sequence of lexicographically-sorted strings, and disk blocks of size 4, 8, 16, and 32 KiB. The first string of each block is stored explicitly (i.e., not compressed), whereas the subsequent strings are compressed with rear coding until the block is (almost) full, that is, it cannot host the subsequent rear-coded string s. In this case, the current block is padded with zeroes, and a new block is started by setting its first string to s. The lengths in rear coding are stored with a variable-byte encoder to keep byte alignment, and thus speed up string decompression.

Since the blocks are of fixed size, the indexing level just needs to return the rank of the block containing the query string, which is then multiplied by the block size to get the byte offset of that block on disk.

To efficiently compute the rank of the query string q in S, we store for each block b an integer indicating how many dictionary strings appear before it in the lexicographic order, denoted with $c(b)$. This way, let $\hat{b}$ be the disk block containing the lexicographic position of the query string q: the rank of q is then computed by summing $c(\hat{b})$ with the *relative* rank of q among the strings in $\hat{b}$. The latter value is obtained via a linear scan and decompression of the block $\hat{b}$, which takes advantage of rear coding and LCP length information to possibly skip some characters, as detailed in [34, §6]. For simplicity, we store the integers $c(b)$ in an in-memory packed array that allocates a number of bits per element sufficient to contain the largest one. It goes without saying that, since these integers are increasing, one could save some further space by using a randomly-accessible compressed integer dictionary (see e.g. [7,20] and references therein), but this is deferred to subsequent studies.

Clearly, one can apply other compression techniques on top of or in place of rear coding, such as entropy coding, grammar compression, and dictionary compression. These techniques have been shown to be useful to reduce the space of in-memory string dictionaries [3,9,10,30,34], but since we are dealing with strings kept in (the much cheaper, but slower) secondary storage, we opt for the simplicity of rear coding, which is shown next to be already very effective in our context. In fact, even for datasets of billions of strings, the number of created blocks (and thus “first strings” to be indexed in memory) is sufficiently small that the indexing level (i.e., the succinct PT) fits in a few MBs (e.g., up to 195 MB for a dictionary of 273 GB, see Section 4). We finally mention that, compared to the approach of creating variable-sized blocks with a fixed number of (front- or rear-coded) strings [30,34], our use of fixed-size blocks allows for better compression because it may take more advantage of runs of consecutive strings sharing long common prefixes, which thus result highly compressible in one single block.



	LOUDS	**DFUDS**
Topology	`1110 1110 0 110 0 110 110 0 0 0 0 0 0`	`110 1110 1110 0 110 0 0 110 0 0 0 110 0 0`
Labels	`amt bln hy eo tx`	`amt bln eo tx hy`
Lengths	`0 1 1 2 1`	`0 1 2 1 1`
Leaves	`5 0 6 7 1 2 3 4`	

Fig. 1. At the top, the Patricia Trie on the strings {*abduct, algebra, algorithm, ant, anxiety, machine, three, typo*} corresponding to the leaves $s_0, \ldots, s_7$. Outside each node, we denote its position in the LOUDS order, in the DFUDS order, and its degree in unary, respectively. At the bottom, the corresponding succinct representations.

The storage level is accessed by memory-mapping the corresponding file (via the `mmap` system call), which compared to explicit `read`s of disk blocks allows a simpler implementation and often faster performance [39].

3.2 Indexing Level

We succinctly encode the Patricia Trie (PT), forming the indexing level, by considering one of two succinct representations of its topology, i.e. LOUDS or DFUDS, and using two additional sequences: one for the single characters labelling the edges of the PT, and the other for the root-to-node path lengths. Both sequences are stored as packed arrays whose elements are ordered according to the topology representation, thus in level-wise order for LOUDS and in preorder for DFUDS. To reduce the number of bits needed to store the lengths, we consider the length of the edge that leads to a node and not the one of the whole root-to-node path, which can be easily recovered by summing the lengths of the visited nodes during the downward traversal (see Section 2).

If LOUDS is used, we need one more sequence that maps each leaf in the level-wise ordering to the lexicographic rank of the corresponding string, which we need to jump to the corresponding block in the storage level. If DFUDS is used, such a sequence is not needed since the leaves are ordered according to the lexicographic rank of the corresponding strings. Figure 1 shows an example of the sequences created for the encoding of a PT.

Downward traversal with LOUDS. To downward traverse the PT encoded with LOUDS, rank and select primitives are used: $\mathit{rank}_b(i)$ counts the number of bits equal to b up to position i, while $\mathit{select}_b(i)$ finds the position of the ith bit equal to b. Assuming that the nodes, their children, and the bits of the binary sequences are counted starting from 0, it is well known [25,37,43] that we can traverse the trie downwards by computing the position of the kth child of the node that starts at position p with the formula $\mathit{select}_0(\mathit{rank}_1(p+k))+1$. We can prove that we do not need rank_1, because its result can be computed with proper arithmetic operations during the traversal. This fact (whose proof is in Appendix A) allows in practice to save space, because we discard the auxiliary data structure needed for constant-time rank_1 operations, and to save time, because several CPU cycles and possibly cache misses are needed for rank_1.

Fact 1 *The downward traversal of a Patricia trie encoded with LOUDS can be executed with just* select_0 *operations.*

When a leaf is reached, we compute its rank in the leaf sequence by counting how many leaves appear before its position x in the LOUDS representation of the PT. This rank is given by $\mathit{rank}_0(x) - \mathit{rank}_{10}(x)$, where the first value denotes the number of nodes (internal and leaves) that appear in LOUDS before the considered one, and the second value denotes the number of internal nodes (not leaves) that appear before position x. Now we notice that the value $\mathit{rank}_0(x) = x - \mathit{rank}_1(x) + 1$ can be computed by substituting $\mathit{rank}_1(x)$ with the value returned by the arithmetic operations executed during the downward traversal, as mentioned above and detailed in Appendix A.

Thus, we build overall just the select_0 and rank_{10} data structures on the LOUDS sequence (due to their time efficiency [29], we use the `sux` library [41] for the former, and the `sdsl` library [23] for the latter).

Downward traversal with DFUDS. To downward traverse the PT encoded with DFUDS, we compute the position of the kth child of the node whose encoding starts at position p with the formula $\mathit{close}(\mathit{succ}_0(p)-(k+1))+1$ [37]. Here, $\mathit{succ}_0(p)$ returns the position of the first 0 that follows p in the DFUDS sequence, and it is implemented by using a linear scan starting from the position p until a 0 is found. Since DFUDS can be seen as a sequence of balanced parenthesis, we have that if i is the position of an open parenthesis, $\mathit{close}(i)$ returns the position of the corresponding close one. For *close* we adopt the `sdsl::bp_support_sada` implementation of balanced parenthesis.

When a leaf is reached, we compute its rank among the leaves with a rank_1 and rank_{10} operation. By knowing the position where the leaf starts, the rank_1 allows us to derive the number of nodes that appear in the sequence before it, while the rank_{10}, as for LOUDS above, allows us to compute how many of these nodes are internal nodes, thus by exploiting the results of these operations we get the rank of the leaf. Therefore, in our implementation of DFUDS, we exploit data structures that allow us to execute in constant time operations of rank_{10}, *close*, and rank_1 (these last two ones are included in `sdsl::bp_support_sada`).

Upward traversal in LOUDS and DFUDS. For the upward traversal of a PT (either encoded with LOUDS or DFUDS), we need to scan back the nodes accessed during the downward traversal. But, instead of executing any of the bit-operations above (as typically done for the upward traversal of trees [25]), we adopt a much simpler and time-efficient approach that pushes in a stack the LOUDS/DFUDS positions of the nodes visited during the downward traversal, and then it pops them from the stack during the upward traversal.

4 Experiments

Experimental Setting. We use a machine with a KIOXIA KPM61RUG960G SSD and two NUMA nodes, each with a 1.80 GHz Intel Xeon E5-2650L v3 CPU and 30 GB local DDR4 RAM. The machine runs Ubuntu 20.04.4 LTS with Linux 5.4.0, and the compiler is GCC 9.4.0. We schedule experiments on a single node via `numactl`. For the `mmap` in the storage level, we tested both the `MAP_SHARED` and `MAP_PRIVATE` flags and noticed no significant performance difference (indeed, the storage level is read-only), so we choose the former. The `MAP_POPULATE` flag too did not impact the query performance, so we do not set it. We alternate datasets given to `mmap` to try to prevent caching by the operating system. Our source code is available at https://github.com/MariagiovannaRotundo/Two-level-indexing.

Datasets. Table 1 in Appendix B shows the sizes of the datasets used in previous experimental evaluations of state-of-the-art solutions (i.e., FST [43], PDT [24], and CoCo-trie [6]). The known datasets are quite small, their size is indeed no more than 0.5 GB and 25M strings for FST, 2.7 GB and 40.5M strings for the CoCo-trie, and 7.1 GB and 114.3M strings for PDT.

Since we want to evaluate our solution on big datasets, we introduce two new ones. The first, *URLs*, combines web page addresses from various crawls [8], has a size of 272.7 GB, and contains 3.7 billion strings. The second, *Filenames*, consists of the name of source code files collected by Software Heritage [1,14,13,32], has a size of around 68.9 GB, and contains 2.3 billion strings. So our datasets are larger than the ones used in previous evaluations by up to 32.0× in number of strings and up to 38.4× in size. Also, we point out that our datasets are up to one order of magnitude larger than the internal memory of our machine, described above.

About the features of the new datasets, we briefly report that URLs contains long strings (avg. 73.6, max. 2083) with long LCPs among them (avg. 53.7), on a medium-size alphabet (88 characters); whereas Filenames offers the opposite features, namely shorter string (avg. 29.1, max. 16051) with even shorter LCPs among them (15.4), on a large alphabet (241 characters).

Competitors. For the indexing level, we consider the set S' of the first strings of each block truncated at their minimum distinguishing prefix, use S' to construct an in-memory index, and then we discard S'. As the index, other than our PT-LOUDS and PT-DFUDS implementations, we consider FST [43], PDT [24],

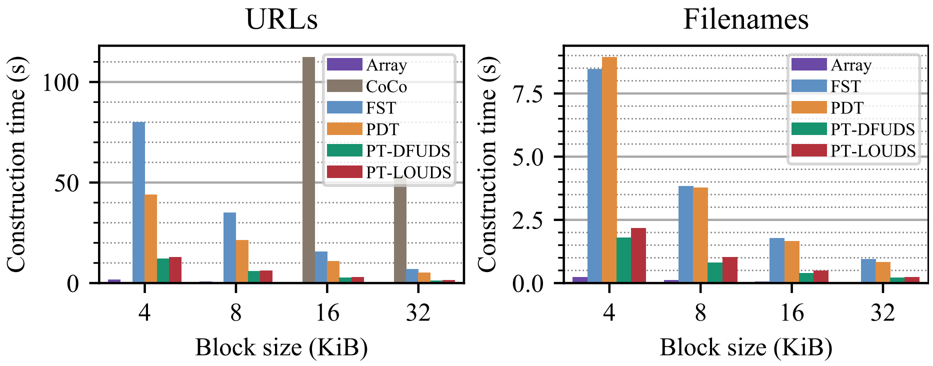


Fig. 2. Times needed to construct each data structure in the indexing level.

CoCo-trie [6], and a simple and commonly-used solution [35,34] — that we name Array — which stores S' contiguously in an array and binary searches on it via an auxiliary packed array of offsets to the beginning of the strings. Notice that, for all solutions, the truncation of strings in S' saves space in the resulting index and still allows identifying the correct block in the storage level (actually, upon accessing the first string of a block we might find that the sought string is in the preceding block, which nonetheless is likely to be loaded quickly thanks to disk prefetching). On the other hand, PT does not store the distinguishing prefixes but only $\Theta(|S'|)$ characters/edges/nodes, thus occupying a space that is independent of the string lengths. We also anticipate that all these implementations of the indexing level allow us to fit it in the internal memory of our machine and thus solve a query with at most two random I/Os to the storage level.

In what follows, we first evaluate in Section 4.1 the different data structures for the indexing level in isolation, i.e. without considering the access to the storage level that concludes the query. Then, in Section 4.2 we evaluate the performance of the overall two-level approach.

4.1 Indexing Level Evaluation

Construction Time. Figure 2 shows the time to construct the various data structures from the set S' loaded in memory. CoCo-trie is constructed only on URLs because the current implementation [6] supports only ASCII alphabets. Moreover, we point out that its construction time for blocks of 4 and 8 KiB is not shown due to its high-memory consumption that required a machine with a much larger internal memory and thus different performance (still, we constructed these CoCo-tries because we test their search time in Figure 3).

Unsurprisingly, Array has the fastest construction because it involves just strings and offsets storage. Our PT-LOUDS and PT-DFUDS implementations have the second-fastest construction, which is based on scanning prefixes at increasing lengths of (ranges) of strings, determining sub-ranges corresponding

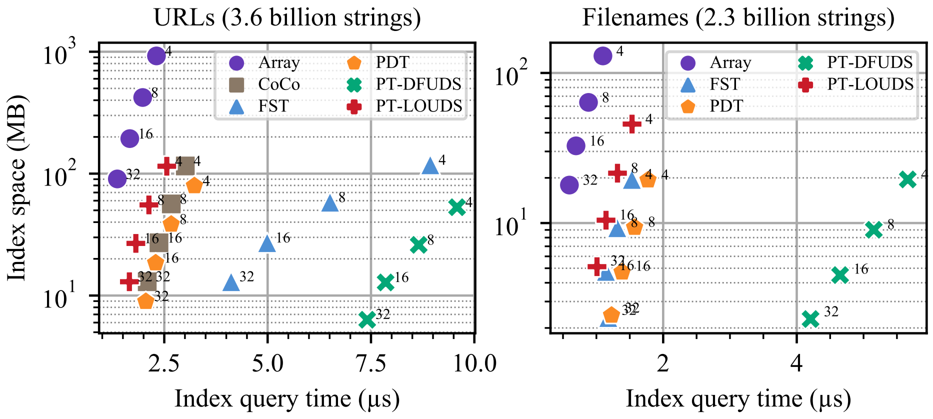


Fig. 3. Space and average query time of different data structures for the indexing level.

to deeper levels of the PT, and handling these sub-ranges recursively in LOUDS order or DFUDS order. Finally, we notice that FST, PDT, and CoCo-trie are significantly slower to construct than our PT, up to 7×, 5×, 42×, respectively.

Space-Time Performance. Figure 3 shows the performance of data structures for the indexing level. The query time refers to the average time needed to perform a membership query on a sample of 10% strings drawn from the set of distinguishing prefixes S', without any access to the storage level. In particular, for PT, since such access is needed for Stage 2 of the blind search (cf. Section 2), the time is evaluated by executing a downward and an upward traversal.

The results show that Array is the fastest but also the most space-hungry solution. FST is competitive only for the Filenames dataset due to its shorter strings. Our PT approaches, despite their simplicity, are very competitive and on the Pareto space-time frontier of both experimented datasets. In particular, PT-LOUDS is the second-fastest data structure with a space occupancy that is competitive with that of the most sophisticated solutions such as CoCo and PDT. We notice in fact that the difference in space with those data structures is no more than 35 MB, which is not much significant given the size of the indexed dictionaries. On the other hand, our PT-DFUDS is the most space efficient but also it is the slowest solution due to the more complex bit-operations needed to traverse the PT structure (hence, we leave as an open issue their engineering).

4.2 Two-Level Approach Evaluation

Given the results of the previous section, we restrict our evaluation of the overall solution (involving the indexing level in memory and the storage level on disk) just to Array and PT-LOUDS, since the other data structures are either not competitive or too much complex for this indexing setting (as detailed above),

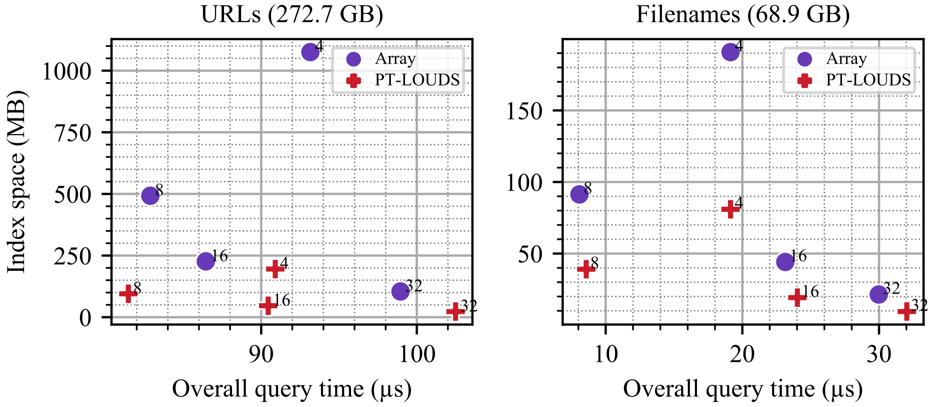


Fig. 4. Space and average query time of our two-level approach.

or their current implementations do not return the rank of the query string among the indexed ones, being this a crucial information to jump to the correct disk block. We mention here that returning the rank of the query string in the LOUDS-based FST requires adding an integer for each leaf (as we did with our PT-LOUDS, cf. Figure 1), thus increasing the space of FST, or it requires switching to the much slower DFUDS representation, thus increasing the query time. On the other hand, returning the rank of a query string in PDT requires more complex trie traversals thus increasing the query time. So Figure 3 underestimates the space-time performance of FST or PDT when they are used in the two-level setting, which justifies our choice of experimenting below just with Array and PT-LOUDS (henceforth referred to simply as PT).

The following paragraphs discuss the experimental results reported in Figure 4. Note that, as stated in Section 3.1, we need to keep in memory the array of integers $c(b)$ to answer rank queries on the indexed strings (which is why the index space in Figure 4 is larger than the one reported in Figure 3).

Storage Level Size. We begin by reporting that our storage level with blocks of size 4–32 KiB compresses the URLs dataset to 80.5–82.2 GB, and the Filenames dataset to 35.9–36.1 GB (details in Table 2 in Appendix C). Therefore, our approach to the blocked-compressed storage of dictionary strings achieves a compression factor of up to 3.4× on URLs, and up to 1.9× on Filenames, which is an interesting achievement given the simplicity of rear coding.

Space-Time Performance. Figure 4 shows that the PT and Array configurations with 8 KiB blocks are the fastest solutions overall. In particular, PT is faster on URLs and Array on Filenames (although PT is very close), but PT takes 5.2× less memory than Array on URLs, and 2.3× less on Filenames.

For increasing block sizes from 8 to 32 KiB, both solutions with PT and Array get from 1.3× to 3.7× slower, because of the larger block to scan and decompress, but more space efficient. Notably, as the block size halves, PT scales better in memory consumption compared to Array, because its space does not depend on the length of the strings but just on their number (as already observed above).

Interestingly enough, the PT and Array configurations with 4 KiB blocks are dominated by the corresponding ones with 8 KiB blocks. This occurs because the indexing level takes more space and thus there is less memory available for caching disk pages, hence making page faults more frequent, as we have verified with the `mincore` system call. The more space available for caching explains also why PT is not slowed down by the execution of one more random I/O compared to Array because of Stage 2 of the blind search (c.f. Section 2).

5 Conclusions and Future Work

Our two-level approach based on a succinct Patricia trie is a robust candidate for indexing massive string dictionaries. As we showed above, it enables indexing up to 272.7 GB with less than 195 MB of internal memory (a space at least 1396.3× smaller than the dictionary's size). This small memory footprint allows dedicating much more memory to caching disk pages and this, in turn, determines a query efficiency that is comparable to or faster than the one offered by Array-based solutions (which take 5.2× more memory). We believe these findings are significant not only for static dictionaries but also for dynamic ones that occur in the design of modern storage systems. As an example, RocksDB [35] is based on (static) runs of strings with in-memory Array-based indexes.

As future work, other than investigating the impact of our findings on these storage systems we suggest: for the indexing level, combining Patricia tries with dynamic succinct tree representations [26] or proper compressors for node fan-outs (similarly to FST and CoCo-trie); and, for the storage level, designing solutions that take into account the query distribution to reduce the average time for block decompression/scan, or that use more sophisticated techniques on top of rear coding (such as dictionary and grammar compression) to improve block compression thus further reducing the internal-memory footprint of Patricia tries.

Acknowledgements. We thank Antonio Boffa for executing some tests on the CoCo-trie, and the Green Data Centre at the University of Pisa for machines and technical support. We also thank Roberto Di Cosmo, Valentin Lorentz, Stefano Zacchiroli, and the Software Heritage team for providing us with the Filenames dataset. This work was made possible by Software Heritage, the great library of source code: https://www.softwareheritage.org.

This work has been supported by the European Union – Horizon 2020 Program under the scheme "INFRAIA-01-2018-2019 – Integrating Activities for Advanced Communities", Grant Agreement n. 871042, "SoBigData++: European Integrated Infrastructure for Social Mining and Big Data Analytics" http://www.sobigdata.eu, by the NextGenerationEU – National Recovery and Resilience Plan (Piano Nazionale di Ripresa e Resilienza, PNRR) – Project: "SoBigData.it - Strengthening the Italian RI for Social Mining and Big Data Analytics"

– Prot. IR0000013 – Avviso n. 3264 del 28/12/2021, by the spoke "FutureHPC & BigData" of the ICSC – Centro Nazionale di Ricerca in High-Performance Computing, Big Data and Quantum Computing funded by European Union – NextGenerationEU – PNRR, by the Italian Ministry of University and Research "Progetti di Rilevante Interesse Nazionale" project: "Multicriteria data structures and algorithms" (grant n. 2017WR7SHH).

References

1. Abramatic, J., Di Cosmo, R., Zacchiroli, S.: Building the universal archive of source code. Commun. ACM **61**(10), 29–31 (2018). https://doi.org/10.1145/3183558
2. Acharya, A., Zhu, H., Shen, K.: Adaptive algorithms for cache-efficient trie search. In: Proc. International Workshop on Algorithm Engineering and Experimentation (ALENEX). pp. 300–315 (1999). https://doi.org/10.1007/3-540-48518-X_18
3. Arz, J., Fischer, J.: LZ-compressed string dictionaries. In: Proc. 24th Data Compression Conference (DCC). pp. 322–331 (2014). https://doi.org/10.1109/DCC.2014.36
4. Baskins, D.: A 10-minute description of how Judy arrays work and why they are so fast (2002), http://judy.sourceforge.net/doc/10minutes.htm
5. Benoit, D., Demaine, E.D., Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Representing trees of higher degree. Algorithmica **43**(4), 275–292 (2005). https://doi.org/10.1007/s00453-004-1146-6
6. Boffa, A., Ferragina, P., Tosoni, F., Vinciguerra, G.: Compressed string dictionaries via data-aware subtrie compaction. In: Proc. 29th International Symposium on String Processing and Information Retrieval (SPIRE). pp. 233–249 (2022). https://doi.org/10.1007/978-3-031-20643-6_17, implementation available at https://github.com/aboffa/CoCo-trie
7. Boffa, A., Ferragina, P., Vinciguerra, G.: A learned approach to design compressed rank/select data structures. ACM Trans. Algorithms **18**(3) (oct 2022). https://doi.org/10.1145/3524060
8. Boldi, P., Marino, A., Santini, M., Vigna, S.: BUbiNG: massive crawling for the masses. ACM Trans. Web **12**(2), 12:1–12:26 (2018). https://doi.org/10.1145/3160017, datasets of URLs available at https://law.di.unimi.it/datasets.php
9. Boncz, P., Neumann, T., Leis, V.: FSST: fast random access string compression. PVLDB **13**(12), 2649–2661 (jul 2020). https://doi.org/10.14778/3407790.3407851
10. Brisaboa, N.R., Cerdeira-Pena, A., de Bernardo, G., Navarro, G.: Improved compressed string dictionaries. In: Proc. 28th ACM International Conference on Information and Knowledge Management (CIKM). pp. 29–38 (2019). https://doi.org/10.1145/3357384.3357972
11. Chikhi, R., Holub, J., Medvedev, P.: Data structures to represent a set of k-long DNA sequences. ACM Comput. Surv. **54**(1) (mar 2021). https://doi.org/10.1145/3445967
12. Clark, J.L.: PATRICIA-II. two-level overlaid indexes for large libraries. Int. J. Parallel Program. **2**(4), 269–292 (1973). https://doi.org/10.1007/BF00985662
13. Di Cosmo, R.: Should we preserve the world's software history, and can we? In: Proc. 26th International Conference on Theory and Practice of Digital Libraries (TPDL). pp. 3–7 (2022). https://doi.org/10.1007/978-3-031-16802-4_1

14. Di Cosmo, R., Zacchiroli, S.: Software Heritage: Why and how to preserve software source code. In: Proc. 14th International Conference on Digital Preservation (iPRES) (2017), https://hdl.handle.net/11353/10.931064
15. Ferragina, P.: Pearls of Algorithm Engineering. Cambridge University Press (2023). https://doi.org/10.1017/9781009128933
16. Ferragina, P., Frasca, M., Marinò, G.C., Vinciguerra, G.: On nonlinear learned string indexing. IEEE Access **11**, 74021–74034 (2023). https://doi.org/10.1109/ACCESS.2023.3295434
17. Ferragina, P., Grossi, R.: The String B-tree: a new data structure for string search in external memory and its applications. J. ACM **46**(2), 236–280 (mar 1999). https://doi.org/10.1145/301970.301973
18. Ferragina, P., Grossi, R., Gupta, A., Shah, R., Vitter, J.S.: On searching compressed string collections cache-obliviously. In: Proc. 27th ACM Symposium on Principles of Database Systems (PODS). pp. 181–190 (2008). https://doi.org/10.1145/1376916.1376943
19. Ferragina, P., Luccio, F.: String search in coarse-grained parallel computers. Algorithmica **24**(3-4), 177–194 (1999). https://doi.org/10.1007/PL00008259
20. Ferragina, P., Manzini, G., Vinciguerra, G.: Compressing and querying integer dictionaries under linearities and repetitions. IEEE Access **10**, 118831–118848 (2022). https://doi.org/10.1109/ACCESS.2022.3221520
21. Ferragina, P., Venturini, R.: Compressed cache-oblivious string B-tree. ACM Trans. Algorithms **12**(4), 52:1–52:17 (2016). https://doi.org/10.1145/2903141
22. Fredkin, E.: Trie memory. Commun. ACM **3**(9), 490–499 (Sep 1960). https://doi.org/10.1145/367390.367400
23. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: Plug and play with succinct data structures. In: Proc. 13th International Symposium on Experimental Algorithms (SEA). pp. 326–337 (2014). https://doi.org/10.1007/978-3-319-07959-2_28
24. Grossi, R., Ottaviano, G.: Fast compressed tries through path decompositions. ACM J. Exp. Algorithmics **19** (jan 2015). https://doi.org/10.1145/2656332, implementation available at https://github.com/ot/path_decomposed_tries
25. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS). pp. 549–554 (1989). https://doi.org/10.1109/SFCS.1989.63533
26. Joannou, S., Raman, R.: Dynamizing succinct tree representations. In: Proc. 11th International Symposium Experimental Algorithms (SEA). pp. 224–235 (2012). https://doi.org/10.1007/978-3-642-30850-5_20
27. Joulin, A., Grave, E., Bojanowski, P., Douze, M., Jégou, H., Mikolov, T.: Fasttext.zip: Compressing text classification models. CoRR **abs/1612.03651** (2016), http://arxiv.org/abs/1612.03651
28. Krishnan, U., Moffat, A., Zobel, J.: A taxonomy of query auto completion modes. In: Proc. 22nd Australasian Document Computing Symposium (ADCS) (2017). https://doi.org/10.1145/3166072.3166081
29. Kurpicz, F.: Engineering compact data structures for rank and select queries on bit vectors. In: Proc. 29th International Symposium on String Processing and Information Retrieval (SPIRE). pp. 257–272 (2022). https://doi.org/10.1007/978-3-031-20643-6_19
30. Lasch, R., Oukid, I., Dementiev, R., May, N., Demirsoy, S.S., Sattler, K.: Fast & strong: The case of compressed string dictionaries on modern CPUs. In: Proc. 15th International Workshop on Data Management on New Hardware (DaMoN). pp. 4:1–4:10 (2019). https://doi.org/10.1145/3329785.3329924

31. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: ARTful indexing for main-memory databases. In: Proc. 29th IEEE International Conference on Data Engineering (ICDE). pp. 38–49 (2013). https://doi.org/10.1109/ICDE.2013.6544812
32. Lorentz, V., Di Cosmo, R., Zacchiroli, S.: The popular content filenames dataset: Deriving most likely filenames from the Software Heritage archive. Tech. rep. (Jul 2023), https://inria.hal.science/hal-04171177, preprint
33. Luo, C., Carey, M.J.: LSM-based storage techniques: a survey. VLDB J. **29**(1), 393–418 (2020). https://doi.org/10.1007/s00778-019-00555-y
34. Martínez-Prieto, M.A., Brisaboa, N.R., Cánovas, R., Claude, F., Navarro, G.: Practical compressed string dictionaries. Inf. Syst. **56**, 73–108 (2016). https://doi.org/10.1016/j.is.2015.08.008
35. Meta Platforms, Inc.: RocksDB, https://rocksdb.org/
36. Morrison, D.R.: PATRICIA—practical algorithm to retrieve information coded in alphanumeric. J. ACM **15**(4), 514–534 (Oct 1968). https://doi.org/10.1145/321479.321481
37. Navarro, G.: Compact Data Structures: A Practical Approach. Cambridge University Press (2016). https://doi.org/10.1017/CBO9781316588284
38. O'Neil, P.E., Cheng, E., Gawlick, D., O'Neil, E.J.: The log-structured merge-tree (LSM-tree). Acta Informatica **33**(4), 351–385 (1996). https://doi.org/10.1007/s002360050048
39. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts. Wiley, 10 edn. (2018)
40. Tsuruta, K., Köppl, D., Kanda, S., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: c-trie++: a dynamic trie tailored for fast prefix searches. Information and Computation **285**, 104794 (2022). https://doi.org/10.1016/j.ic.2021.104794
41. Vigna, S.: Broadword implementation of rank/select queries. In: Proc. 7th International Workshop on Experimental Algorithms (WEA). pp. 154–168 (2008). https://doi.org/10.1007/978-3-540-68552-4_12
42. Zhang, H., Andersen, D.G., Pavlo, A., Kaminsky, M., Ma, L., Shen, R.: Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In: Proc. ACM International Conference on Management of Data (SIGMOD). pp. 1567–1581 (2016). https://doi.org/10.1145/2882903.2915222
43. Zhang, H., Lim, H., Leis, V., Andersen, D.G., Kaminsky, M., Keeton, K., Pavlo, A.: Succinct range filters. ACM Trans. Database Syst. **45**(2) (jun 2020). https://doi.org/10.1145/3375660, fork of the implementation available at https://github.com/kampersanda/fast_succinct_trie
44. Zhang, W., Hou, L., Yin, Y., Shang, L., Chen, X., Jiang, X., Liu, Q.: TernaryBERT: distillation-aware ultra-low bit BERT. In: Proc. 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP). pp. 509–521 (2020). https://doi.org/10.18653/v1/2020.emnlp-main.37

A Proof of Fact 1

Proof. We start by recalling two basic identities of *rank* and *select* primitives

$$\mathit{rank}_1(x) = x - \mathit{rank}_0(x) + 1 \text{ and } \mathit{rank}_0(\mathit{select}_0(x)) = x.$$

From the above two identities, it also holds

$$\begin{aligned} \mathit{rank}_1(\mathit{select}_0(y)) &= \mathit{select}_0(y) - \mathit{rank}_0(\mathit{select}_0(y)) + 1 \\ &= \mathit{select}_0(y) - y + 1. \end{aligned} \tag{1}$$

Let p be the position of the currently visited internal node in the LOUDS bitvector B, i.e. the degree $d \geq 1$ of the current node is represented in $B[p, p+d] = 1^d 0$. We now show that the well-known formula [25,43] $\mathit{select}_0(\mathit{rank}_1(p+k)) + 1$ that allows going from the current node to its kth child ($0 \leq k < d$), can be computed with just select_0 and arithmetic operations. Clearly, it is enough to focus on the $\mathit{rank}_1(p+k)$ part of the formula. We distinguish two cases.

If the currently visited node is the root, i.e. $p = 0$, it is easy to see that

$$\mathit{rank}_1(p+k) = \mathit{rank}_1(k) = k + 1, \tag{2}$$

because $B[p, p+d] = B[0, d] = 1^d 0$.

Otherwise, if the current visited node is an internal node different from the root, then p has been computed with the known formula as $p = \mathit{select}_0(\mathit{rank}_1(p' + k')) + 1$, where p' is the starting position of the parent of the current node, and k' is the position of the current node among its siblings. Let us call $y = \mathit{rank}_1(p' + k')$, and thus $p = \mathit{select}_0(y) + 1$. Then, it holds

$$\begin{aligned} \mathit{rank}_1(p+k) &= \mathit{rank}_1(p) + k && \text{(since } B[p, p+k] \text{ is all 1s)} \\ &= \mathit{rank}_1(\mathit{select}_0(y) + 1) + k && \text{(by substitution of } p\text{)} \\ &= \mathit{rank}_1(\mathit{select}_0(y)) + k + 1 && \text{(since } B[\mathit{select}_0(y) + 1] = B[p] = 1\text{)} \\ &= \mathit{select}_0(y) - y + k + 2 && \text{(by Equation (1))} \end{aligned}$$

So, during the downward traversal, all the operations of the form $\mathit{rank}_1(p+k)$ can be replaced with arithmetic and select_0 operations on $y = \mathit{rank}_1(p' + k')$, whose value was computed at the parent of the current node (up from the root, where y is trivially given by Equation (2)). □

B Datasets Used in Previous Evaluations

Table 1. Datasets used in previous papers, ordered by their sizes in GBs, compared to the two datasets introduced in this paper, which are up to about 40× larger. The size of "Emails addresses" is not explicitly indicated in [43], but we derive it from the number and average length of email addresses indicated in that paper.

Dataset	# strings (M)	Size (GB)	Reference
xml	2.9	0.1	CoCo-trie [6]
protein	2.9	0.1	CoCo-trie [6]
enwiki-titles	8.5	0.1	PDT [24]
dna	13.7	0.1	CoCo-trie [6]
aol-queries	10.2	0.2	PDT [24]
Integer keys	50.0	0.4	FST [43]
tpcds-id	30.0	0.4	CoCo-trie [6]
Emails addresses	25.0	0.5	FST [43]
uk-2002	18.5	1.4	PDT [24]
synthetic	2.5	1.5	PDT [24]
url	40.5	2.7	CoCo-trie [6]
webbase-2001	114.3	7.1	PDT [24]
Filenames	2294.3	68.9	this paper
URLs	3654.1	272.7	this paper

C Index and Storage Space of the Two-Level Solution

Table 2. Space used by the storage level, where blocks of strings are compressed with rear coding, and by the indexing level built on the first string of each block by considering different block sizes.

Blocks	Solution	URLs		Filenames	
		Index (MB)	Storage (GB)	Index (MB)	Storage (GB)
4K	Array	1075.3	82.2	190.7	36.1
	PT	195.2		80.9	
8K	Array	492.5	81.2	91.3	36.0
	PT	95.0		39.0	
16K	Array	226.5	80.7	44.1	35.9
	PT	46.4		19.2	
32K	Array	104.3	80.5	21.3	35.9
	PT	22.8		9.5	