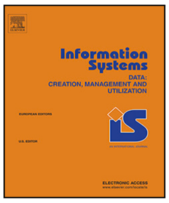
Information Systems 128 (2025) 102490

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is



Two-level massive string dictionaries[☆]

Paolo Ferragina [*], Mariagiovanna Rotundo [**], Giorgio Vinciguerra

Department of Computer Science, University of Pisa, L.go B. Pontecorvo 3, Pisa 56127, PI, Italy

ARTICLE INFO

Recommended by Matthias Weidlich

Dataset link: https://github.com/MariagiovannaRotundo/Two-level-indexing

Keywords:
String dictionary problem
Trie data structure
String compression
Algorithm engineering
Key–value stores

ABSTRACT

We study the problem of engineering space–time efficient data structures that support membership and rank queries on *very* large static dictionaries of strings.

Our solution is based on a very simple approach that decouples string storage and string indexing by means of a block-wise 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. On top of this, we design an in-memory cache that, given a sample of the query workload, augments the Patricia trie with additional information to reduce the number of I/Os of future queries.

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, compared to Patricia tries, do not provide significant benefits when used in a large-scale indexing setting, and (ii) our two-level approach enables the indexing and storage of 3.5 billion strings taking 273 GB in just less than 200 MB of internal memory and 83 GB of compressed disk space, 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 design.

1. Introduction

The string dictionary problem is a classic one in the string-matching field. Given a set S of n strings of variable length drawn from an alphabet Σ, the goal is to build a data structure that stores S and efficiently answers *membership queries* on any string $q \in \Sigma^+$, namely, checking whether $q \in S$. 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 kind of query is motivated by the fact that it also enables the implementation of several other queries, such as the *prefix search*, which finds all the strings in S prefixed by q, and the *range search*, which finds all the strings in S that fall in a given range. We do not consider update operations and thus assume that S is static. This is common in several storage systems, such as the ones based on LSM-trees [1,2], which support updates by periodically merging static sorted string sets.

The recent explosion of massive string dictionaries in several applications — such as databases [2,3], bioinformatic tools [4], search engines [5], code repositories [6], and string embeddings [7,8], just to name a few — has revitalized 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.

Different solutions have been proposed over the years. A trivial one, but widely used in practice, consists of using a binary-searched array of string pointers, which incurs random memory accesses and possibly I/Os. The classic solution hinges upon the trie data structure [9], a multiway tree that stores each string in S as a root-to-leaf path, and whose edges are labeled 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 [10] and refs therein) such as compacting subtries [10,11], using adaptive representations for its nodes [12–14], succinct representations of its topology [3,15], cache-aware or disk-based tree layouts [16,17], and even replacing it with learned models [18].

Among the most recent and performing variants of tries, we mention: ART [14], CART [19], Path Decomposed Trie (PDT) [20], Fast Succinct Trie (FST) [3], ctrie++ [11], and CoCo-trie [10]. According to

[☆] This is an extended version of: Paolo Ferragina, Mariagiovanna Rotundo, and Giorgio Vinciguerra. Engineering a Textbook Approach to Index Massive String Dictionaries. SPIRE 2023. DOI:10.1007/978-3-031-43980-3_16.

[*] Correspondence to: Department L'EMbeDS, Sant'Anna School of Advanced Studies, Pisa, Italy.

[**] Corresponding author.

E-mail addresses: paolo.ferragina@santannapisa.it (P. Ferragina), mariagiovanna.rotundo@phd.unipi.it (M. Rotundo), giorgio.vinciguerra@unipi.it (G. Vinciguerra).

https://doi.org/10.1016/j.is.2024.102490
Received 21 February 2024; Received in revised form 29 October 2024; Accepted 30 October 2024
Available online 8 November 2024

the most recent experimental results published in [10], we know that ART, CART, and ctrie++ are space-inefficient and offer query times on par with the other data structures. 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 to date. Nevertheless, they incur three main "limitations": they are very complex to implement; their code is highly engineered, and thus difficult to maintain or adapt to different scenarios (e.g., rank operations, adding satellite information); and, finally, they are designed to compress and index the string dictionary 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, namely, the ones in which the number n of strings and their total length are so large that S has to be kept in slow storage, such as HDDs or SSDs, or in far memories [21].

Inspired by the theoretical proposals of [17,22–24], our solution consists of decoupling string indexing and string storage via a two-level approach [25]. The on-disk storage level compresses the sorted strings in S via rear coding [16] and partitions them into blocks of fixed size. The indexing level exploits a succinctly-encoded Patricia trie [23,26] 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, decompressed, 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 I/Os without resorting to more complicated solutions [17,23].

A proper evaluation of this two-level approach demands the use of massive string datasets. Unfortunately, previous evaluations [3,10,20] are of little help here because they employ datasets with at most 233 million strings and of size at most 9.9 GB. For this reason, we introduce two datasets that are at least one order of magnitude larger: one consisting of URLs from various Web crawls (about 3.5 billion strings for a total of 272 GB) [27], the other consisting of names of source code files from the Software Heritage initiative (about 2 billion strings for a total of 69 GB) [28].

Thanks to large-scale experiments, we can provide (often unexpected) answers to several interesting questions arising in the design and engineering of massive string dictionaries:

- *Do we need to employ sophisticated compression techniques for the storage of sorted strings?* No, we show that simple methods like rear coding obtain very competitive compression ratios while providing much faster compression and decompression speeds than dictionary-based compressors (such as Gzip, Zstd, Xz, Brotli), grammar compressors (Re-Pair [29]), and even recent proposals (FSST [30]). Furthermore, when considering a block-wise compression of strings, which facilitates random access, dictionary-based compressors might actually obtain a worse compression ratio.
- *Do we need sophisticated data structures to index strings in memory?* No, we show that sophisticated string dictionaries like FST, PDT, and CoCo-trie are unnecessarily complex for this purpose because they do not provide substantial space–time performance advantage compared to our well-engineered succinct Patricia trie, which is also much faster to construct.
- *Does a faster in-memory index always improve the overall performance of the two-level string dictionary?* No, we show that the fastest in-memory index (namely, a binary-searched array of strings[1]) exhibits a worse search performance than the Patricia trie when considering not only the index access but also the disk access in scenarios with limited internal memory and large on-disk storage levels. This is due to the fact that using more internal memory for a faster index actually steals space available for caching disk pages and thus increases page faults. Our Patricia trie implementation, indeed, takes only at most 195 MB of internal memory (three orders of magnitude smaller than the dataset size), which is up to 5.2× smaller than the array-based solution, and this turns out to impact successfully the overall search performance.
- *Is caching of disk pages the best use we can make of the remaining internal memory space?* No, we show that we can construct a special in-memory cache that, given a query workload that is representative of future queries, augments the Patricia trie with additional information to reduce the number of I/Os and make queries faster compared to using just the operating system's cache (e.g., up to 8.1× faster with under 41 MB of in-memory cache).

Overall, we show that 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 concluding section of this paper.

Finally, we note this paper extends its conference version [32] by further elaborating on the content covered in Sections 2 and 3.2, and by introducing the following new contributions:

- The experimental evaluation of different compression techniques to adopt for the storage level (Section 4.1 and Figs. 3 and 5);
- A new approach that constructs an in-memory cache to speed up queries by conceptually interpolating between a Patricia trie and a compacted trie according to a given internal-memory budget (Section 3.3);
- The experimental evaluation of the impact of this new caching approach on our two-level string dictionary (Section 4.4 and Figs. 9–14).

2. Background

We now describe some basic ingredients of our two-level approach, starting from the main data structure we use to index and search strings in internal memory.

The Patricia trie [26] for a string set S is derived from the compacted trie of S by keeping for every single edge its first labeling character, and by storing at each node the length of the (prefix of the) string spelled out by that root-to-node path. Fig. 1 shows an example of a Patricia trie built on a set of 8 strings.

Even if the Patricia trie strips out some information from the compacted trie, it is still able to support the search for the lexicographic position of a query string q 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 random I/O instead of the $|q|$ random I/Os potentially incurred by the traversal of the compacted trie due to accessing its (possibly long) $|q|$ edge labels. This algorithm is called *blind search* in the literature [23,25]. It is a little bit more complicated than prefix searching in classic tries, because of the presence of only the first character at each edge label. Technically speaking, the blind search consists of three stages.

Stage 1: Downward traversal. Trace a downward path in the Patricia trie to locate a leaf l which points to one of the indexed strings sharing the Longest Common Prefix (LCP) with q (see [23] for the proof). The traversal compares the characters of q 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 q against the string s pointed to by leaf l, in order to determine their LCP length ℓ.

[1] This is a rudimentary but nonetheless effective solution in practice, as attested by its use on some real data systems like RocksDB [31].

Stage 3: Upward traversal. Traverse upward the Patricia trie 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 $s[\ell+1] \neq q[\ell+1]$), then we determine the lexicographic position of $q[\ell + 1]$ among the branching characters of u. Say this is the ith child of u, the lexicographic position of q 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 q is to the immediate right of the subtree descending from edge e if $q[\ell + 1] > s[\ell + 1]$, otherwise, it is to the immediate left of that subtree.

As an example, assume we are searching for the string q = "alarm" on the Patricia trie in Fig. 1. In Stage 1, we first trace the path that spells "al", the first two characters of q. Then, we reach a node in which no further branching is possible because the fourth letter of q does not match any outgoing edge. Therefore, we choose the leaf l as the leftmost descendant of the current node, i.e. the leaf corresponding to s_1 = "algebra". In Stage 2, we compare s_1 and q and determine their LCP length $\ell = 2$. In Stage 3, we traverse upward the Patricia trie from l until reaching the edge e where the mismatched character $q[\ell+1]$ = "a" is, which in our case is the second edge traversed in Stage 1, i.e. the one labeled "l". Then, we notice that the mismatched character $q[\ell + 1]$ is smaller than $s_1[\ell + 1]$, so the lexicographic position of q is to the immediate left of the subtree descending from edge e, i.e. it is between s_0 and s_1.

The topology of the Patricia trie 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) [15] and the Depth-First Unary Degree Sequence (DFUDS) [33]. 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 output bitvector: in level-wise left-to-right order for LOUDS, and in preorder for DFUDS. For our implementation of DFUDS, we follow [34] and prepend 110 to this binary representation; instead for LOUDS, we follow [3] and prepend no bits. See Fig. 1 for an example of LOUDS and DFUDS representations.

Regarding the compression of a lexicographically-sorted set of strings, we will concentrate on two simple techniques: front coding [16,25] and rear coding [16]. Front coding represents each string with two values: an integer denoting the length of the LCP between the considered string and the previous one in the sequence, and the remaining suffix of the considered string obtained by removing that LCP. If the string has no 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.

As an example, for the sorted set of strings {algebra, algebraic, algorithm, ant, anxiety, machine, three, typo}, front coding produces the pairs $\langle 0, \text{algebra}\rangle$ $\langle 7, \text{ic}\rangle$ $\langle 3, \text{orithm}\rangle$ $\langle 1, \text{nt}\rangle$ $\langle 2, \text{xiety}\rangle$ $\langle 0, \text{machine}\rangle$ $\langle 0, \text{three}\rangle$ $\langle 1, \text{ypo}\rangle$, while rear coding produces the pairs $\langle 0, \text{algebra}\rangle$ $\langle 0, \text{ic}\rangle$ $\langle 6, \text{orithm}\rangle$ $\langle 8, \text{nt}\rangle$ $\langle 1, \text{xiety}\rangle$ $\langle 7, \text{machine}\rangle$ $\langle 7, \text{three}\rangle$ $\langle 4, \text{ypo}\rangle$.

Rear coding may be more efficient than front coding since it does not encode the length of repeated prefixes [16,17]. It goes without saying that a set of strings (not necessarily sorted) can also be concatenated and compressed with dictionary-based and grammar-based compression techniques. In the former group, there are compressors (such as Gzip, Xz, and Zstd) based on the Lempel–Ziv algorithm [35], which processes the strings left-to-right by replacing each repeated substring with a reference to an earlier occurrence from a fixed-length sliding window. Another recently proposed dictionary-based compressor is FSST [30], which exploits a properly constructed table of 255 entries to replace as many frequently occurring substrings of up to 8 bytes with 1-byte codes. In the latter group, there are techniques such as Re-Pair [29], which compress repeated substrings by generating grammar rules that are then used to encode those repetitions via rule IDs. In Section 4.1 we will experiment with several of these approaches and show that our choice of rear coding is an effective one because it achieves a good compromise among compression speed, decompression speed, and compression ratio.

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 string of every block. The following Sections 3.1 and 3.2 detail these two levels, while Section 3.3 describes how to cache in internal memory some properly-chosen substrings to reduce disk I/Os.

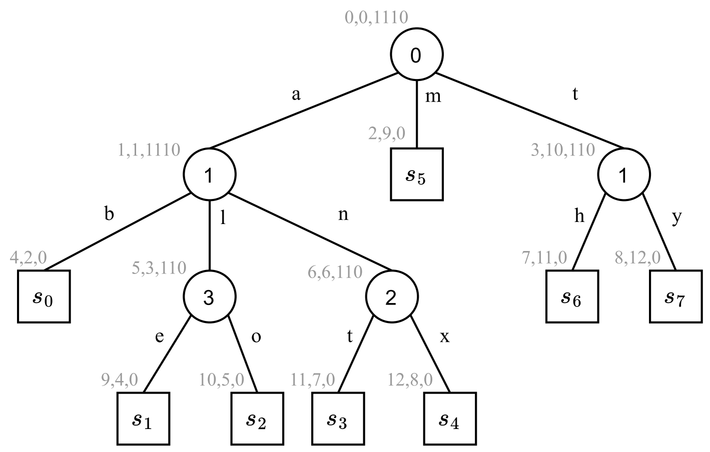
3.1. *Storage level*

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 zeros, 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. Of course, other encoders could be applied [36–40].

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. However, since the strings have variable length but the block size is fixed, to efficiently compute the rank of the query string q in S, we need to 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 [41, §6]. For simplicity, we store the integers $c(b)$ in an in-memory packed array that allocates some bits per element sufficient to contain the largest one. Since these integers are increasing, one could save some further space by using a randomly accessible compressed integer dictionary (see e.g. [42,43] and references therein), but this is deferred to subsequent studies.

Clearly, one could apply other compression techniques on top of or in place of rear coding, such as entropy coding, grammar compression, and dictionary compression. These techniques are useful for reducing the space of in-memory string dictionaries [30,41,44–46]. Hence we experiment with several of them in Section 4.1 (see also Fig. 4) and find that our choice of rear coding hits a sweet spot among compression speed, decompression speed, and compression ratio. We finally mention that, compared to the approach of creating variable-sized blocks with a fixed number of (front- or rear-coded) strings [41,46], 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.

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 [47].



	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 8 strings {abduct, algebra, algorithm, ant, anxiety, machine, three, typo} corresponding to the leaves $s_0, \ldots, s_7$. Outside each node, we write its position in the LOUDS order, in the DFUDS order, and its degree in unary, respectively. At the bottom, we show the two succinct representations: LOUDS on the left, and DFUDS on the right. Notice that the Labels sequence stores the first character of every trie edge, and the Lengths sequence stores the length of the substring spelled on the edge that leads to each node. In the LOUDS representation, there is also a Leaves sequence storing the ranks of the strings corresponding to the leaves.

3.2. Indexing level

The indexing level consists of a Patricia trie (PT) built on the first string of every block in the storage level. We succinctly encode the Patricia trie by considering one of two succinct representations of its topology, either LOUDS or DFUDS. Furthermore, we use two additional sequences: one for the single characters labeling the edges of the Patricia trie, and the other for the lengths of the string prefixes spelled out by root-to-node paths. 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 of the string prefixes, we store just the length of the substring spelled by the edge that leads to each node, since we can easily recover the original value 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.

Fig. 1 shows an example of the sequences created for the encoding of a Patricia trie.

The overall space usage can be upper bounded as follows. Let s be the number of strings indexed by the Patricia trie (or equivalently, the number of blocks in the storage level), and let $m \leq 2s-1$ be the number of nodes in the Patricia trie (in Fig. 1, $s = 8$ and $m = 13$). The topology representation needs $2m{-}1$ bits with LOUDS, or $2m{+}2$ bits with DFUDS. The sequence encoding the edge labels needs $(m-1)\log\sigma$ bits, where σ is the alphabet size of the branching characters. The sequence encoding the lengths of the string prefixes needs $(m-s)\log\ell$ bits, where ℓ is the maximum length value (which can be upper bounded by the maximum LCP length between consecutive sorted strings). Summing up and assuming 1-byte edge labels, i.e. $\log\sigma = 8$, the overall space is at most $10m + (m-s)\log\ell \leq (20+\log\ell)s$ bits. If LOUDS is used, further $s\log s$ bits are needed for the sequence mapping each leaf to the lexicographic rank of the corresponding string.

The rest of this Section details how to search for the lexicographic position of q within the strings indexed by the Patricia trie, by distinguishing between the two proposed succinct representations (i.e., LOUDS and DFUDS). This will in turn identify the disk block containing q's lexicographic position within the whole set S of indexed strings.

Downward traversal with LOUDS. To downward traverse the Patricia trie encoded with LOUDS, *rank* and *select* primitives are used [34]: $rank_b(i)$ counts the number of bits equal to b up to position i, while $select_b(i)$ 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 [3,15,34] that we can traverse the trie downwards by computing the position of the kth child of the node whose encoding starts at position p with the formula $select_0(rank_1(p+k))+1$. We show below that we do not need $rank_1$ because its result can be computed with proper arithmetic operations during the traversal. This fact allows us 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.*

Proof. We start by recalling two basic identities of *rank* and *select* primitives (recall that positions in the binary sequences are counted from 0):

$$rank_1(x) = x - rank_0(x) + 1, \tag{1}$$

and

$$rank_0(select_0(x)) = x. \tag{2}$$

From these two equalities, it follows that

$$\begin{aligned} rank_1(select_0(y)) &= select_0(y) - rank_0(select_0(y)) + 1 \\ &= select_0(y) - y + 1. \end{aligned} \tag{3}$$

Let us now consider the traversal of the Patricia trie via *rank* and *select* primitives, and 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 $B[p, p+d] = 1^d0$. We now show that the well-known formula $select_0(rank_1(p+k))+1$ 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. Thus we focus on the $rank_1(p+k)$ part of the formula and distinguish two cases.

If the currently visited node is the root of the Patricia trie, then $p = 0$ and $rank_1(p+k) = rank_1(k) = k+1$ because $B[p, p+d] = B[0, d] = 1^d 0$.

Otherwise, the currently visited node is an internal node of the Patricia trie, thus p has been computed with the known formula as $p = select_0(rank_1(p'+k')) + 1$, where p' is the starting position of its parent, and k' is its position among its siblings. Let us call $y = rank_1(p'+k')$, and thus $p = select_0(y) + 1$. Then, it holds

$$\begin{aligned}
rank_1(p+k) &= rank_1(p) + k && \text{(since } B[p, p+k] \text{ is all 1s)}\\
&= rank_1(select_0(y)+1) + k && \text{(by substitution of } p)\\
&= rank_1(select_0(y)) + k + 1 && \text{(since } B[select_0(y)+1] = B[p] = 1)\\
&= select_0(y) - y + k + 2 && \text{(by Eq. (3))}
\end{aligned}$$

So, during the downward traversal, all the operations of the form $rank_1(p+k)$ can be replaced with arithmetic and $select_0$ 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 Patricia trie. This rank is given by $rank_0(x) - rank_{10}(x)$, where the first value denotes the number of nodes (internal and leaves) that appear in LOUDS before the considered one, and the second value (for which *rank* is queried on the two bits 10) denotes the number of internal nodes that appear before position x. Now, the value $rank_0(x) = x - rank_1(x) + 1$ by Eq. (1), can be computed by substituting $rank_1(x)$ with the value returned by the arithmetic operations executed during the downward traversal, as detailed in the above proof of Fact 1. Instead, the value $rank_{10}(x)$ needs a proper data structure built on the LOUDS sequence.

Overall, we have thus proved that to support the downward traversal of a Patricia trie encoded with LOUDS we need to build just two succinct data structures on the LOUDS binary sequence that support $select_0$ and $rank_{10}$ operations. Due to their time efficiency [48], we will implement the former with the `sux` library [49], and the latter with the `sdsl` library [50].

Downward traversal with DFUDS. To downward traverse the Patricia trie encoded with DFUDS, we compute the position of the kth child of the node whose encoding starts at position p with the formula $close(succ_0(p) - (k+1)) + 1$ [34]. Here, $succ_0(p)$ returns the position of the first 0 that follows position p in the DFUDS sequence, and it is implemented by using a linear scan starting from p until a bit 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, $close(i)$ returns the position of the corresponding close one. For *close* we adopt the `sdsl::bp_support_sada` implementation of balanced parenthesis [50,51]. When a leaf is reached, we compute its rank among the leaves of the Patricia trie via $rank_1$ and $rank_{10}$ operations. By knowing the position where the encoding of a leaf starts, the $rank_1$ operation allows us to derive the number of nodes that appear in the sequence before it, while the $rank_{10}$ operation, as for LOUDS above, allows us to compute how many of these nodes are internal nodes. Thus subtracting those two values we get the rank of the reached leaf. Therefore, in our implementation of DFUDS, we exploit data structures that allow us to execute in constant time the 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 Patricia trie, 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 [15]), 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 out from the stack during the upward traversal.

Finalizing the query. After the upward traversal, we have identified the correct lexicographic position of the query string q among the strings indexed by the Patricia trie, and thus we can infer the block $\hat{b}$ in the storage level containing the lexicographic position among all indexed strings of S. Thus the search for q is concluded with a linear scan of $\hat{b}$ as described in Section 3.1.

If the Patricia trie fits in internal memory, the search costs overall $O(|q|)$ time and incurs at most 2 random I/Os: one at the end of the downward traversal to compare q with the first string in a block $\bar{b}$ (cf. Stage 2 in Section 2); and another random I/O to access the block $\hat{b}$ if this block is different from $\bar{b}$ (and possibly far from it, since the prefetcher could have loaded some blocks neighboring $\bar{b}$).

We can summarize this result as follows (note we are making the simplifying, yet reasonable, assumption that each input string fits into a disk block).

Theorem 1. *Given a set S of n variable-length strings, the proposed two-level string dictionary takes $O(s)$ space for the indexing level, where $s = O(n)$ is the number of blocks taken by the on-disk storage level storing the compressed S.*

If the indexing level fits in internal memory, then the rank of a query string q can be determined in $O(|q|)$ time and at most 2 random I/Os; otherwise, its downward traversal might incur in up to $|q|$ I/Os.

We remark that the additional random I/Os incurred by the Patricia trie that does not fit in internal memory can be reduced by using more sophisticated solutions or trie layouts [16,17,23].

In practice, our experiments in Section 4 show that the combination of rear coding (which obtains a small s) and our succinct encoding of Patricia tries make a very space-efficient indexing level (e.g. up to 195 MB for a string dictionary of 273 GB), which not only fits in the internal memory of any commodity machine, but it also allows us to use the remaining memory to further decrease the number of random I/Os via a suitably designed caching strategy, as we do in the next section.

3.3. Caching

We now describe an algorithmic approach that, given a query workload that is representative of future queries, constructs an in-memory cache with enough information to potentially reduce the number of I/Os of our two-level approach.

The high-level idea is to turn the Patricia trie into a *hybrid* data structure consisting of a mix of a compacted trie and a Patricia trie, namely, into a trie in which some properly chosen edges— hereafter termed *cached* edges— are fully labeled with substrings (as in a compacted trie), while the other edges are labeled with just the first character of these substrings (as in a Patricia trie). The choice between a full label or a single-character label for an edge will be made according to given the query workload; whereas, the number of cached edges will depend on a given *cache size* parameter, which we assume to be fixed in advance by the user according to the memory availability: the higher it is, the closer the hybrid trie is to a compacted trie; the smaller it is, the closer the hybrid trie is to a Patricia trie. Fig. 2 shows an example of a Patricia trie, a hybrid trie, and a compacted trie built on a set of 6 strings, where the two cached edges in the hybrid trie are highlighted with a thick line and lead to the strings s_4 and s_5.

As we will see, thanks to the cached edges, the lexicographic position of a query string q among the indexed strings of the Patricia trie can sometimes be identified immediately during its downward traversal, thereby reducing the traversal time and saving the random I/O required by Stage 2 of the lexicographic search described in the previous Section.

For selecting which edges to cache, we adopt a strategy based on their traversal frequency induced by the strings in the given query workload. More precisely, we consider as *candidate edges* for caching: the ones leading to an internal node, having a nonzero frequency and labeled with more than one character (since those with a single character must be kept anyway, as in the standard Patricia trie). Edges

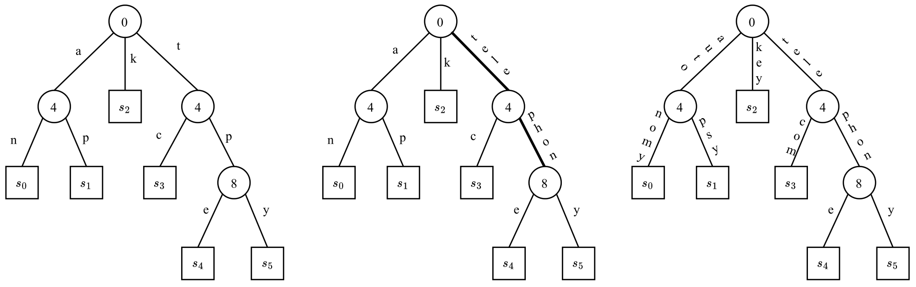


Fig. 2. The Patricia trie, a hybrid trie, and the compacted trie on the strings {autonomy, autopsy, key, telecom, telephone, telephony} corresponding to the leaves $s_0, \ldots, s_5$. The intuition behind the caching shown in the figure is that one of the two strings s_4 and s_5 is more frequent than all the other strings, and thus the edges on its leading path get fully represented (i.e., cached).

leading to a leaf can be dropped from the candidate set because they "distinguish" between two adjacent disk blocks, which are anyway fetched together by the disk caching. Then, we sort the candidate edges from the most- to the least-frequently accessed, and we cache edges in this order until the cache is full.

It is easy to observe that our cached edges satisfy a sort of *prefix-completeness property*: if the label of an edge e gets cached, so are all the labels of the edges in the path that leads to e, because they have an access frequency greater than or equal to that of e. For example, in Fig. 2, out of the three candidate edges "auto", "tele", and "phon", the hybrid trie at the center has cached the last two edges that form a downward path in the shown trie.

Succinct representation of the cache. Let n_e be the number of edges and let n_c be the number of cached edges. To implement the cache we augment the succinct representation of the Patricia trie in Section 3.2 with three sequences: a length-n_e bit sequence C that, for each edge, indicates whether an edge is cached or not, a string E that concatenates the substrings labeling the cached edges (without their first character, which is already stored in the labels sequence of the Patricia trie), and a length-n_c integer sequence D specifying the starting offsets of the cached edges into E. The elements in these sequences are ordered according to the succinct representation of the trie topology (either LOUDS or DFUDS).

So, for example, consider the Patricia trie in Fig. 2 encoded with LOUDS, and assume that the most frequent edges are the ones labeled with 't' and 'p' in the first and second level, respectively, which correspond to the thick edges in the hybrid trie at the center. The bit sequence C is 001000100, the string E concatenates the substrings "ele" and "hon", whereas the integer sequence D stores the offsets 0 and 3 specifying where the two edge labels start in E (we assume zero-based indexing).

Checking whether an edge is cached or not then amounts accessing $C[p]$ where p is the node index (in LOUDS or DFUDS order) the edge leads to; while the characters labeling a cached edge are found starting from $E[D[k]]$, where k is given by a $rank_1(p) - 1$ query on C. It goes without saying that alternatives to the use of $rank_1$ do exist: for example, we can replace it with a linear scan that counts the number of bits set before the current edge to get the index of the offset to access.

Semi-blind search algorithm. To traverse the hybrid trie given a query string q, we execute additional checks compared to the blind search algorithm (Section 2). Specifically, we check whether the traversed edge e is cached (and thus its label is fully available) or not (and thus its edge label consists of only the first character). If it is cached, the corresponding characters in q are compared with those at the edge and, if a mismatch is found, the lexicographic position of q is already available: say s is the string spelled by the root-to-e path and there is a mismatch $q[\ell+1] \neq s[\ell+1]$ on the label of edge e, then the lexicographic position of q is to the immediate right of the subtree descending from e if $q[\ell+1] > s[\ell+1]$, otherwise, it is to the immediate left of that subtree. If no mismatch is found, the downward traversal continues to the next proper edge. If a non-cached edge of length greater than 1 has to be traversed, the search continues as a standard downward search of a Patricia trie without considering the cache anymore, as specified in Section 3.2.

We notice that, if we keep track of the length ℓ' of the longest cached path matching q, then we can speed up the LCP computation phase (i.e. Stage 2) by starting from the $(\ell'+1)$-th character of the compared strings, rather than from their first one.

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 28× larger. The size of "Emails addresses" is not explicitly indicated in [3], but we derive it from the average length and number of email addresses indicated in that paper.

Dataset	# strings (M)	Size (GB)	Reference
xml	2.9	0.1	CoCo-trie [10]
protein	2.9	0.1	CoCo-trie [10]
enwiki-titles	8.5	0.1	PDT [20]
aol-queries	10.2	0.2	PDT [20]
trec-terms	32.2	0.2	CoCo-trie [10]
tpcds-id	30.0	0.4	CoCo-trie [10]
Integer keys	50.0	0.4	FST [3]
Emails addresses	25.0	0.5	FST [3]
uk-2002	18.5	1.4	PDT [20]
synthetic	2.5	1.5	PDT [20]
dna	367.4	6.5	CoCo-trie [10]
webbase-2001	114.3	7.1	PDT [20]
url	233.2	9.9	CoCo-trie [10]
Filenames	2294.3	68.9	This paper
URLs	3654.1	272.7	This paper

4. Experiments

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 32 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 chose the former. We also tested the `MAP_POPULATE` flag but decided not to use it because it did not impact the performance. 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. Datasets used in previous experimental evaluations of state-of-the-art solutions (i.e., FST [3], PDT [20], and CoCo-trie [10]) were quite small. Their size (shown in Table 1) was indeed no more than 0.5 GB and 25M strings for FST, 7.1 GB and 114.3M strings for PDT, and 9.9 GB and 233.2M strings for the CoCo-trie.

Since we want to evaluate our solution on big datasets, we introduce two new string collections extracted from real applications. The first, *URLs*, consists of web page addresses from various crawls[2] for a total size of 272.7 GB and 3.7 billion strings. The second, *Filenames*, consists of the name of many source-code files collected by Software Heritage [6,28,52,53] for a total size of 68.9 GB and 2.3 billion strings. So our datasets are up to one order of magnitude larger than the internal memory of our machine and larger than the ones used in previous evaluations by up to 15.6× in number of strings and up to 27.5× in size.

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, which does not contain whole paths, offers the opposite features, namely shorter strings (avg. 29.1, max. 16051) with even shorter LCPs among them (15.4), on a large alphabet (241 characters).

Outline of the experiments. In what follows, we first evaluate the performance of different compression techniques in the creation of the storage level (Section 4.1). Then, we evaluate the different data structures for the indexing level in isolation, i.e. without considering the access to the storage level that concludes the query (Section 4.2). After that, we evaluate the performance of the overall two-level approach (Section 4.3). Finally, we evaluate the performance of the proposed solution by adding edge caching (Section 4.4).

4.1. Evaluation of the storage level

We evaluate six compression techniques for the storage level: FSST [30], Gzip v1.10, Re-Pair [29],[3] Xz v5.2.4, Zstd v1.4.4, and our implementation of rear coding. For Gzip, we consider two flags, `-1` and `-9`, to achieve the best compression speed and the best compression ratio, respectively. Instead, we consider the flag `-12` for Zstd, and the flag `-6` for Xz, trying to ensure a good compression ratio while still being fast. Finally, about Re-Pair, we consider a different number of rules that impact the speed and compression ratio, namely: at most 100, 1000, or 10000 rules, or no limit on the number of rules.

As a first analysis, we apply these compressors to the whole dataset (as a unique huge block). Then, for the most efficient solutions, we concentrate on analyzing the block-wise compression performance for each dataset. As evaluation metrics, we use the compression and decompression speed (both expressed in MB/s, thus higher values are better), and the compression ratio (the ratio between the size of the compressed and the uncompressed data, expressed as a %, thus lower values are better).

Whole-dataset compression. Fig. 3 shows the compression and decompression speed vs the compression ratio of the considered compression techniques. Fig. 4 presents the same data but, for each metric, sorts the techniques from best to worst.

The results show that rear coding is the fastest solution in both compression and decompression speed, followed by FSST, Gzip, and Zstd; while Re-Pair and Xz are the slowest solutions. Instead, the best compression ratio is obtained by Xz, followed by Zstd, Gzip (−9 and −1), Re-Pair without limit on the rules, and rear coding; while the worst compression ratios are obtained by Re-Pair with limits on the number of rules and FSST. We recall to the reader that, however, the most space-efficient compressors (i.e., Xz, Zstd, and Gzip) do not support random access to the individual dictionary strings when applied to the whole dataset, as in the present experiment.

We also tested Brotli with flag `-11` but due to its high compression time, we interrupted the execution. The compressed file at the moment

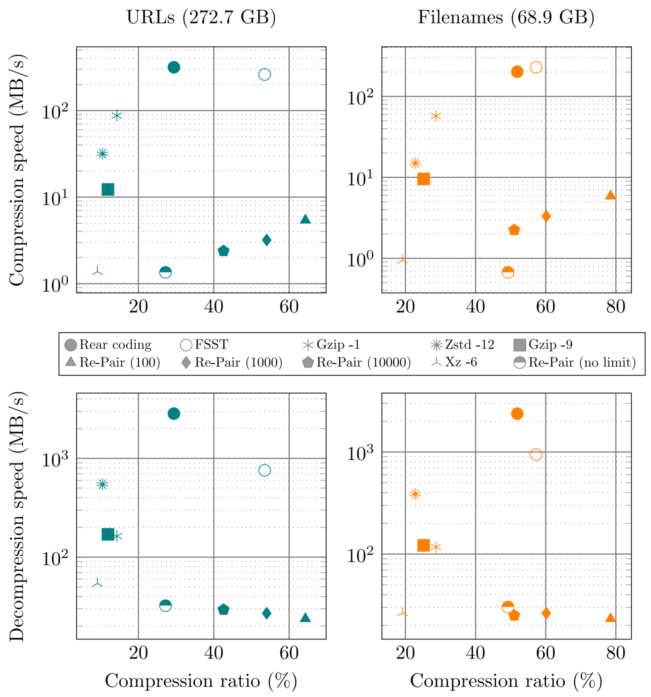


Fig. 3. Space and average query time of different data structures for the indexing level.

of interruption was larger than the one obtained by Gzip `-9`, and thus worse in terms of compression speed and compression ratio.

In light of these experimental results, we consider for the block-wise compression evaluation just rear coding and Zstd `-12` because they have shown, respectively, the best (de)compression speed, and a good trade-off between speed and compression ratio.

Block-wise compression. To support the fast random access to the individual strings of our indexed dictionaries, we applied rear coding and Zstd over blocks of different sizes. For Zstd, we use the APIs for streaming compression and decompression provided by the library,[4] whereas for rear coding we use our implementation.

Fig. 5 shows the performance of the two compressors on the datasets of URLs and Filenames with block sizes between 4 and 32 KiB. We notice that rear coding provides the fastest compression speed, from 69.3 up to 80.1× faster than Zstd on URLs and from 66.9 up to 72.8× faster than Zstd on Filenames. By looking at decompression time and compression ratio, rear coding and Zstd show close performance. In particular, on URLs, rear coding tends to be slightly faster in decompression time than Zstd on small blocks, while it is slower on large blocks. Instead, on Filenames, rear coding is slightly slower than Zstd for every considered block size. About the compression ratio, rear coding is significantly better than Zstd on Filenames, and slightly worse on URLs.

Overall, it is surprising to see that the very simple rear coding achieves in the considered scenario a performance on par with Zstd or even better. Therefore, for the following evaluations, we will adopt rear coding to implement a block-wise storage level since Zstd does not give significant advantages either in terms of decompression speed or in terms of compression ratio, and it is two orders of magnitude slower in compression speed. This latter is crucial to efficiently handle dynamic string dictionaries in storage systems that perform frequent rebuilds "from scratch", as it occurs in LSM-based approaches [2].

[2] Namely, the crawls eu-2015, gsh-2015, clueweb12, and uk-2014 [27].

[3] We use the implementation of Re-Pair available at https://github.com/acubeLab/eXtended-RePair/.

[4] https://facebook.github.io/zstd/zstd_manual.html

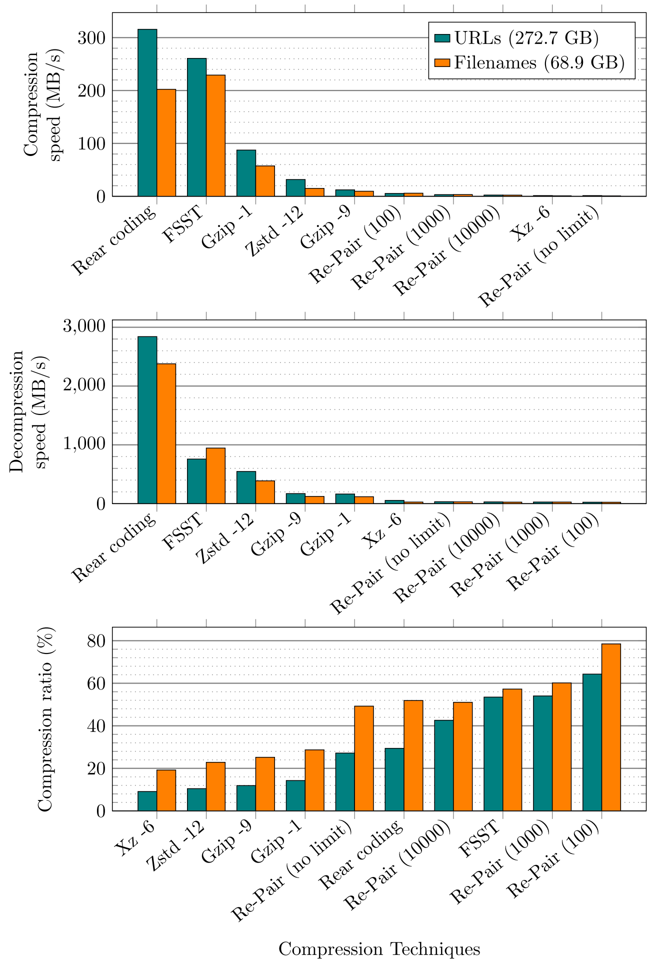


Fig. 4. Compression and decompression speed and compression ratio of the different compression techniques to compress the two considered datasets. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

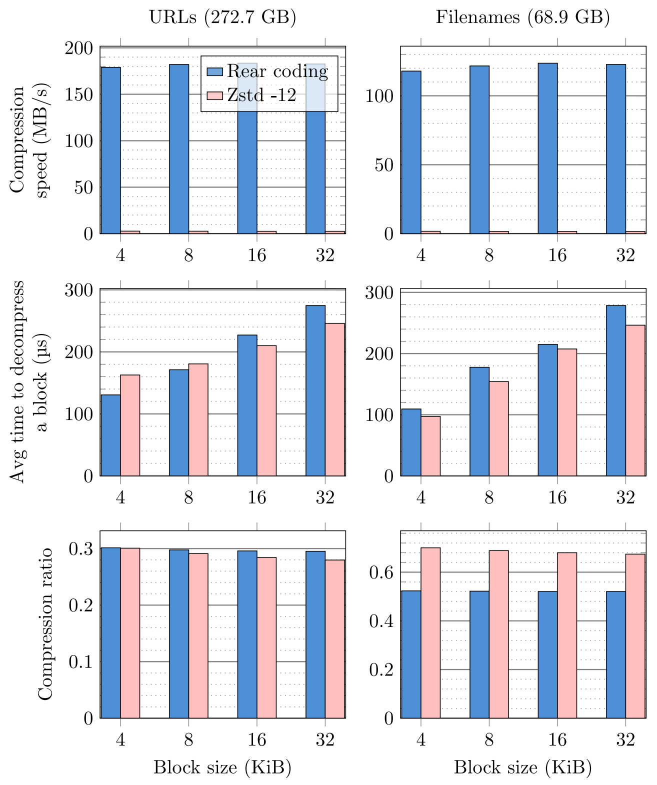


Fig. 5. Compression and decompression speed and compression ratio of rear coding and Zstd applied to the construction of the storage level. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

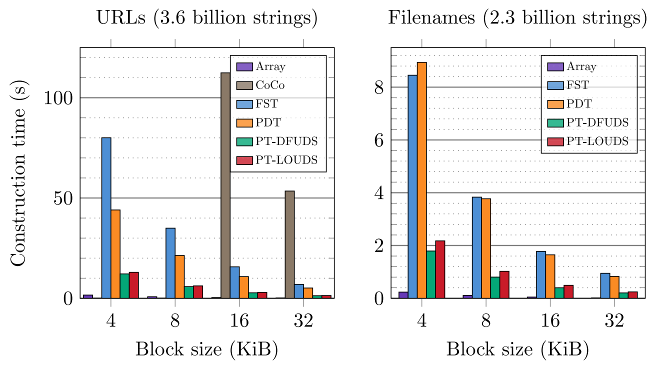


Fig. 6. Times needed to construct each data structure in the indexing level. Construction time for blocks of 4 KiB and 8 KiB are not shown for CoCo-trie due to its high memory consumption. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

4.2. *Evaluation of the indexing level*

We now evaluate different data structures for the indexing level in isolation, that is, without considering the access to the storage level. Specifically, to construct the in-memory index, we consider the set S' composed of the first string of every block truncated at its minimum distinguishing prefix, and then we discard S'. As the index, other than our succinct implementation of the Patricia trie with LOUDS and DFUDS (henceforth, PT-LOUDS and PT-DFUDS, respectively), we consider FST [3], PDT [20], CoCo-trie [10], and a simple and commonly-used solution [31,41] — that we name Array — which stores S' contiguously in an in-memory array and binary searches on it via an auxiliary packed (still in-memory) 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 which includes the lexicographic position of the query string q (actually, upon accessing the first string of a block we might find that q is in the preceding block, which nonetheless is likely to be loaded quickly thanks to disk prefetching). All solutions, except our Patricia trie, use a space that is proportional to the indexed distinguishing prefixes. On the other hand, the Patricia trie 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 indexing data structures fit in the internal memory of our machine and thus solve a query with at most two random I/Os to the storage level.

Construction time. Fig. 6 shows the time to construct the indexing data structures over the set S', available in internal memory. CoCo-trie is constructed only on URLs because the current implementation [10] 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 construct these CoCo-tries because we test their search time in Fig. 7).

Unsurprisingly, Array provides the fastest construction because it involves just string copies and offsets storage. Our PT-LOUDS and PT-DFUDS implementations have the second-fastest construction, which is

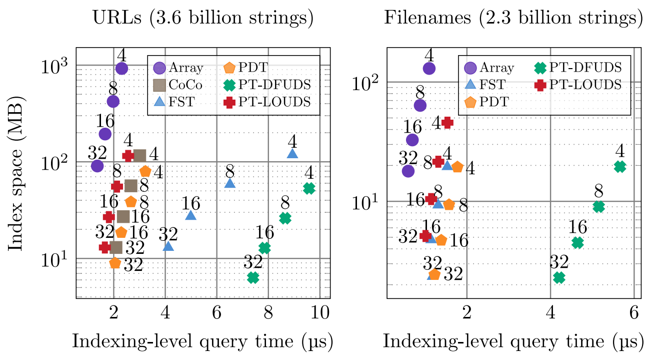


Fig. 7. Space and average query time of different data structures for the indexing level.

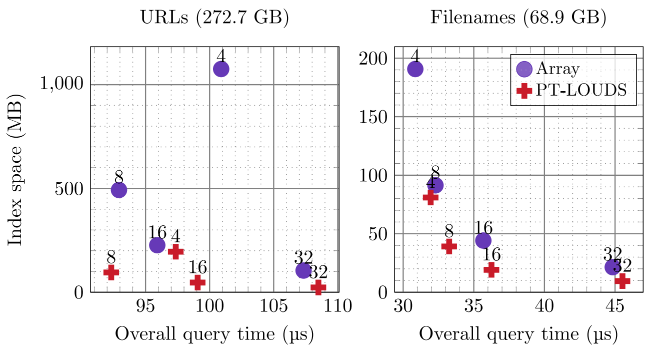


Fig. 8. Index space and query time of our two-level approach with Array or PT in the indexing level. The query time includes both the index and the disk access time.

based on scanning prefixes at increasing lengths of (ranges) of strings, determining sub-ranges corresponding to deeper levels of the PT, and handling these sub-ranges recursively in LOUDS order or DFUDS order.

Fig. 6 shows clearly that our PT-based indexes are significantly faster in construction than the other trie-based indexes such as FST, PDT, and CoCo-trie, by up to 7×, 5×, 42×, respectively.

Space–time performance of the indexing level. Fig. 7 shows the average query time to perform a membership query (on a random sample of 10% of the strings in the set S' of distinguishing prefixes) for various data structures implementing the indexing level, thus not accounting for the I/O cost to access the storage level on disk. For PT, due to the lack of the storage level, we skip Stage 2 of the blind search (cf. Section 2) and execute a full upward traversal to the root.

The best solutions are located near the bottom-left corner of Fig. 7, which corresponds to the fastest and most space-efficient performance. Our experimental results show that Array is the fastest but also the most space-hungry solution. As we increase the query time and decrease the space usage, we find that PDT and FST provide a good trade-off, but this latter only on the Filenames dataset due to its shorter strings that induce a shorter trie traversal. Our PT approaches, despite their simplicity, are very competitive and on the Pareto front 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-trie and PDT. In fact, we notice that the difference in space with those data structures is no more than 35 MB, which is not very significant given the size of the indexed dictionaries. On the other hand, our PT-DFUDS is the most space-efficient but also the slowest solution due to the more complex operations needed to traverse the succinct trie representation (hence, we leave as an open issue their engineering). We recall that CoCo-trie is not constructed for Filenames because the current implementation [10] supports only ASCII alphabets.

4.3. Evaluation of the whole two-level approach

Given the results of the previous sections, we restrict our evaluation of the overall solution (involving the indexing level in internal memory and the storage level on disk) just to Array and PT-LOUDS, since the other data structures are not competitive or too complex for this indexing setting, or their current implementations do not return the lexicographic rank of the query string among the indexed ones, being this a crucial information to jump to the correct disk block.

Let us comment on these limiting issues in some more detail. Returning the rank of the query string in the LOUDS-based FST requires adding an integer value to each leaf (as we did with our PT-LOUDS, cf. Fig. 1), thus increasing the space of FST (from 1.50 to 1.52× on URLs, and from 2.17 to 2.30× on Filenames, depending on the block size); 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 using PDT requires a more complex trie traversal thus increasing the query time. So Fig. 7 underestimates the space needs and query times of FST and PDT when they are used in the two-level setting, which justifies our choice of limiting the experiments below just to Array and PT-LOUDS (henceforth referred to simply as PT).

The following two paragraphs discuss the space and the performance of the two-level approach.

Space of the two-level approach. In Section 4.1, we already discussed the effectiveness of rear coding as a compressor for the storage level due to the good (de)compression speed and compression ratio.

Table 2 shows the exact space of the storage level with rear coding. With blocks of size 4–32 KiB, the URLs dataset (272.7 GB) is compressed to 80.5–82.2 GB, and the Filenames dataset (68.9 GB) is compressed to 35.9–36.1 GB.[5] Therefore, our approach reduces the space by up to 3.4× on URLs, and by up to 1.9× on Filenames, which is an interesting achievement given the simplicity of rear coding.

Table 2 shows also the exact space usage for the indexing level. Note that, to answer rank queries on the indexed strings, as stated in Section 3.1, we need to keep in memory the array of integers $c(b)$ counting the number of strings stored in the disk blocks preceding the bth one (which is why the index space in Table 2 is larger than the one reported in Fig. 7). On URLs, the indexing level with blocks of size 4–32 KiB takes 1075.3–104.3 MB with Array, and 195.2–22.8 MB with PT. On Filenames, the indexing level with blocks of size 4–32 KiB takes 190.7–21.3 MB with Array, and 80.9–9.5 MB with PT. Therefore, under the same block size, on average PT is 5.0× more compressed than Array on URLs, and 2.3× more compressed than Array on Filenames.

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.

Performance of the two-level approach. Fig. 8 shows the trade-off between the overall query time and the indexing level space for both the Array and the PT solutions.

In terms of Pareto front, on URLs, PT dominates Array with its configurations with 8–32-KiB blocks. On Filenames, PT dominates Array with its configurations with 4–32-KiB blocks, with the exception of Array with 4-KiB blocks, which is on the top-left part of the Pareto front.

In terms of query time, on URLs, PT with 8-KiB blocks is the fastest solution and is closely followed by Array with 8-KiB blocks, which requires 5.2× more memory. On Filenames, Array with 4-KiB blocks

[5] In Filenames, some strings are longer than the block size. If they happen to be the first strings of a block, and thus must be stored explicitly, we truncate them to the block size for simplicity of implementation. This happens for just 5 blocks of 4 KiB, and for 2 blocks of 8 KiB, thus it has a negligible impact on the compressed size.

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.

Block size	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	

is the fastest solution, closely followed by PT with 4-KiB blocks, which requires 2.4× less memory. For increasing block sizes from 8 to 32 KiB, both solutions with PT and Array get slower, because of the larger block to scan and decompress, but more space efficient, because of a more effective compression and fewer strings to index in internal memory.

Interestingly enough, on URLs, 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 internal memory available for the disk cache, hence making page faults more frequent, as we have verified with the `mincore` system call. The next section will show how to further reduce the number of page faults with our ad hoc caching strategy.

4.4. Evaluation of our edge-caching strategy

We now experiment with the caching approach described in Section 3.3 by considering two query workloads: one created by randomly picking strings from the datasets, and an *adversarial* one created by mutating randomly picked strings so that the search incurs exactly two random I/Os (if caching is not used). For each experiment, we vary the cache size and analyze the results in terms of four different metrics: the average number of random I/Os per query, the number of page faults,[6] the average time to identify the block where the query string is lexicographically located, and the average query time (that includes also time to scan the identified disk block). A key observation is about the number of random I/Os incurred during a single query: we count them as 1 if the (semi-)blind search accesses two neighboring blocks in the storage layer (since prefetching is likely to make the latter access inexpensive), and 2 otherwise. But a random I/O incurs a page fault only if the requested page is not in the operating system's cache, which might hold distant pages that have been accessed recently in the previous queries. Therefore, the overall number of random I/Os is an upper bound to the number of page faults.

4.4.1. Random query workload

Fig. 9 shows the actual space used by our edge-caching strategy when we impose different thresholds on its maximum size, which are assumed to be a power of two between 1 and 128 MB. Here (and in the following plots/histograms), the missing points/bars correspond to the cases in which the threshold was too low to store even the basic data structure needed for caching. On the other hand, too-high thresholds are not needed since the actual cache size reaches a plateau at 40.2 and 5.5 MB for a block size of 4 KiB on URLs and Filenames, respectively. We also notice that as the block size increases, the cache size gets smaller since fewer strings are indexed by the trie. For example, on URLs, as the block size doubles from 8 KiB to 16 KiB, the cache size halves from 22 MB to 11 MB. This is perfectly in line with the reduction

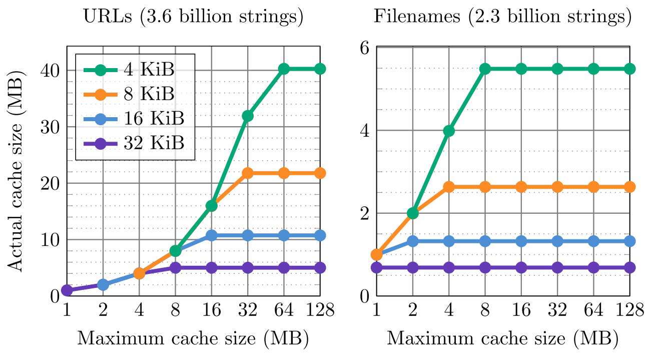


Fig. 9. Actual occupancy of the cache at different upper bounds on its maximum size, under a uniform query workload. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

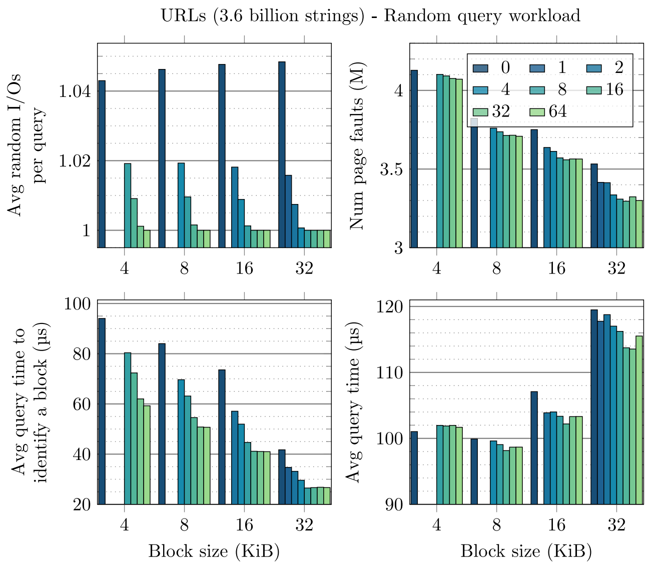


Fig. 10. Impact of the edge caching on a Patricia trie built on the URLs dataset, by considering a random query workload with 10M queries. The legend shows different thresholds on the maximum cache size expressed in MB. Missing bars are due to cases in which the threshold is too low to perform edge caching. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

of the size of the Patricia trie that, by doubling the block size, is built on around half of the strings.

We are now ready to evaluate the impact of our edge-caching strategy on a plain Patricia trie (namely, one in which the threshold on the maximum cache size is zero). In the following, caches larger than 64 MB for URLs and 8 MB for Filenames are not shown because of the previous observations on the plateaus.

[6] The number of page faults is found via the `mincore` system call, notice a page in our system is fixed to 4 KiB, regardless of the block size we vary in our storage layer.

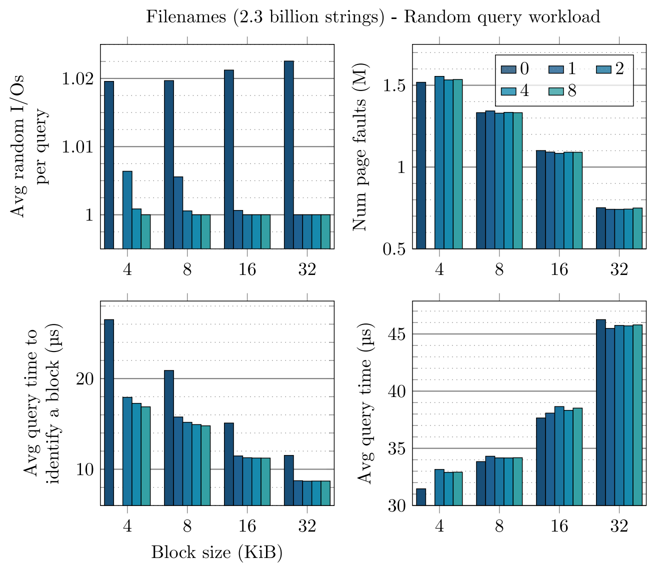


Fig. 11. Impact of the edge caching on a Patricia trie built on the Filenames dataset, by considering a random query workload with 10M queries. The legend shows different thresholds on the maximum cache size expressed in MB. Missing bars are due to cases in which the threshold is too low to perform edge caching. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Let us start by recalling that the number of I/Os needed to solve a query is between 1 and 2, according to the structural properties of our two-level indexing data structure. Fig. 10 shows the performance on URLs, where one notices that the average number of random I/Os per query (top-left plot) is at most 1.05 without caching, and then it decreases up to (the minimum value) 1 by using larger and larger cache sizes. This is because a larger fraction of the queries get solved by identifying the correct block directly during the downward traversal, having fully available the edge labels needed for that traversal, and thus not incurring the single I/O needed for the LCP computation phase.

As it happens for the random I/Os, also the page faults (top-right plot) decrease as the cache size increases. But here, as the block size increases, the page faults decrease more sharply as fewer pages are randomly accessed because the Patricia trie gets smaller. This impacts positively on the internal memory available for the operating system and on the time for its traversal. Consequently, the average time needed to identify the block where a query string is located (bottom-left plot) decreases sharply too, both as the cache size and the block size increase.

Surprisingly, we notice that this decrease in time to identify a block is not reflected in the overall average query time (bottom-right plot), which does not improve that much with a large cache. This is because even if we have identified the block without any I/Os, we still have to fetch it from the disk and scan it. Clearly, the time of this scan increases with the block size, as is evident from the figure. Furthermore, the use of edge caching induces a small overhead because of its more complex code, as evident for example for a block size of 4 KiB.

Let us now turn our attention to the other dataset: Filenames. Fig. 11 shows that the average number of random I/Os per query (top-left plot) is already small even without edge caching, and reaches 1 with edge caching. Consequently, the page faults (top-right plot) are not improved by our caching strategy, contrary to what happened for URLs. The smaller Filenames dataset, indeed, is composed of fewer pages and thus it is more likely that a page is found in the operating system's cache. The average time needed to identify the block where a query is located (bottom-left plot) shows the same trend of URLs but with a less noticeable effect of the larger edge cache. As a consequence of these observations, it is not surprising that the average query time on this dataset (bottom-right plot) does not improve with edge caching but

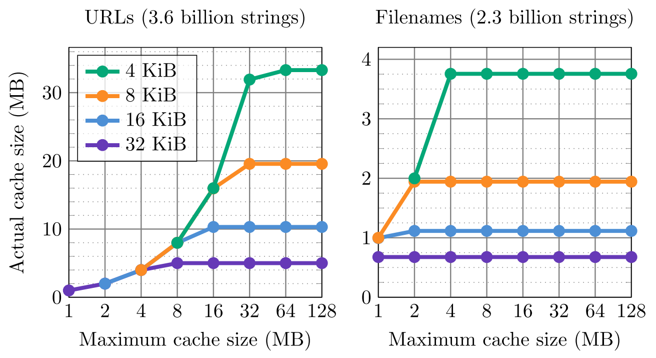


Fig. 12. Actual cache size at different thresholds on the maximum cache size under a uniform query workload. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

actually, unlike URLs, incurs a higher overhead due to the cost of the block scan and the code complexity.

4.4.2. Adversarial query workload

Stimulated by the figures of the previous section, we studied the impact of our edge caching on an *adversarial* query workload, namely one in which the queried strings are forced to require 2 random I/Os. Specifically, we generate each of these query strings by randomly picking a string from the dataset and mutating one character chosen at a random position (the new character is chosen at random between 32 and 255, which excludes ASCII control characters), and then we ensure it elicits 2 random I/Os before adding it to the query workload.

Fig. 12 shows the actual space used by the edge cache at different thresholds on its maximum size. Here (and in the following histograms), the missing bars correspond to the case in which the threshold was too low to perform edge caching, as it occurred in the random query workload of Section 4.4.1. For example, on URLs and Filenames with a block size of 4 KiB, the actual cache size reaches a plateau at 33.3 MB and 3.7 MB, respectively, thus slightly earlier than in the random query workload (where those figures were 40.2 MB and 5.5 MB). We also notice that the actual used cache halves in size as the block size doubles, as it occurred in the random query workload.

We are now ready to evaluate the impact of our caching strategy on the performance of a plain Patricia trie. In the following, caches larger than 64 MB for URLs and 4 MB for Filenames are not shown because of the previous observations on the plateaus.

Fig. 13 shows the performance on URLs. Here, unlike in the random query workload, the average number of random I/Os per query (top-left plot) is 2 for the plain Patricia trie, and it immediately decreases to 1 already with a few MBs of cache size. For example, with a block size of 4 KiB and 32 KiB, just 8 MB and 1 MB of cache are enough, respectively, to reduce the average number of random I/Os from 2 to at most 1.1. This reduction in random I/Os impacts the number of page faults (top-right plot) that, for any block size, reduces by a significant fraction, which is between 87.6% and 93.4%. If we look instead at their overall number, we notice that by increasing the block size from 4 KiB to 32 KiB, this number goes from around 3.2M to 1.7M (i.e. a reduction of 46.2%).

This reduction of the page faults due to edge caching has a significant impact on the average time needed to identify the block where a query is located (bottom-left plot) that, with respect to a plain Patricia trie, decreases from 6.0 to 50.5×. This time decreases also as the block size increases due to the smaller and faster-to-traverse trie.

Finally, unlike in the random query workload (Section 4.4.1), the overall average query time (bottom-right plot) significantly improves with edge caching, showing a decrease by a factor between 2.8 and 8.1×. This difference in the two query workloads is due to the significant reduction in page faults, which in turn depends on the reduction

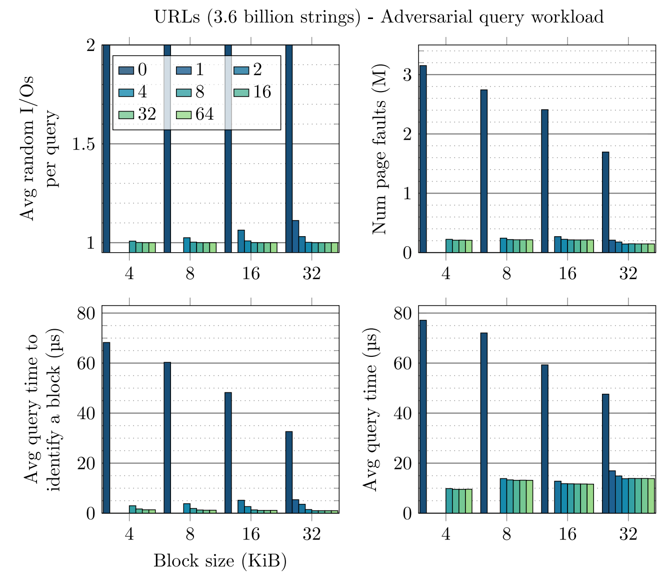


Fig. 13. Impact of the edge caching on the URLs dataset on an adversarial query workload with 10M queries. The legend shows the different thresholds on the maximum cache size in MB. Missing points are due to cases in which the threshold is too low to perform edge caching. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

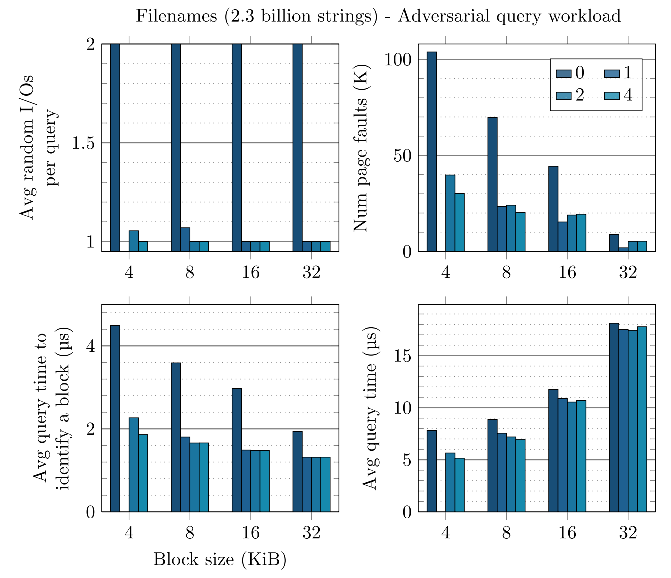


Fig. 14. Impact of the edge caching on the Filenames dataset on an adversarial query workload with 10M queries. The legend shows the different thresholds on the maximum cache size in MB. Missing points are due to cases in which the threshold is too low to perform edge caching. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

in random I/Os enabled by edge caching. Interestingly, larger block sizes do not impact much on the scan phase since the query strings are mutations of the dataset strings and thus mismatches cause the scan to stop earlier compared to the random query workload.

These observations and trends on URLs also hold on the Filenames dataset. Indeed, Fig. 14 shows that the average number of random I/Os per query (top-left plot) is 2 for the plain Patricia trie, and it rapidly decreases to 1 already with 1 or 2 MB of the edge cache. As a consequence, the number of page faults (top-right plot) is between 38.8 and 78.7% smaller than the one with the plain Patricia trie, and the average time to identify the block where a query is located (bottom-left plot) improves by between 32.0% to 58.6%, and the overall average query time (bottom-right plot) improves by between 1.8% to 33.9%. Thus, compared to the random query workload on Filenames (Section 4.4.1), edge caching has a positive effect here too. Nonetheless, the improvements on Filenames are smaller than the ones observed on URLs due to its smaller size, which impacts the smaller number of pages occupied in the storage level, and thus in turn a reduced load of the operating system's cache.

5. Conclusion

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). In addition, we further improve the performance of our approach by enabling edge caching, which reduces the number of page faults and, consequently, the total query time. 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 [31] is based on (static) runs of strings with in-memory array-based indexes.

As future work, other than investigating the impact of our two-level storage solution on these storage systems we suggest: for the indexing level, combining Patricia tries with dynamic succinct tree representations [54] or proper compressors for node fan-outs (similar to FST and CoCo-trie); and, for the storage level, designing data-aware solutions that take into account the query distribution and/or the string distribution to reduce the average time for block decompression/scan. For the storage level, it may also be considered and evaluated the use of variable-length codes [37,38] to compress the lengths of the longest common prefixes. These would require additional data structures to directly access the encoded length of the first string of the blocks and to retrieve two blocks from the storage level to answer a query, but it might reduce the space needed to store these integers and the number of blocks in the storage level and this could impact positively on the overall performance.

CRediT authorship contribution statement

Paolo Ferragina: Writing – review & editing, Writing – original draft, Validation, Supervision, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization. **Mariagiovanna Rotundo:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Giorgio Vinciguerra:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Methodology, Investigation, Formal analysis, Data curation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this article.

Acknowledgments

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 was 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 European Union – 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).

Data availability

Datasets and code are available at https://github.com/MariagiovannaRotundo/Two-level-indexing.

References

[1] P.E. O'Neil, E. Cheng, D. Gawlick, E.J. O'Neil, The log-structured merge-tree (LSM-tree), Acta Inform. 33 (4) (1996) 351–385, http://dx.doi.org/10.1007/s002360050048.

[2] C. Luo, M.J. Carey, LSM-based storage techniques: a survey, VLDB J. 29 (1) (2020) 393–418, http://dx.doi.org/10.1007/s00778-019-00555-y.

[3] H. Zhang, H. Lim, V. Leis, D.G. Andersen, M. Kaminsky, K. Keeton, A. Pavlo, Succinct range filters, ACM Trans. Database Syst. 45 (2) (2020) http://dx.doi.org/10.1145/3375660, Fork of the implementation available at https://github.com/kampersanda/fast_succinct_trie.

[4] R. Chikhi, J. Holub, P. Medvedev, Data structures to represent a set of k-long DNA sequences, ACM Comput. Surv. 54 (1) (2021) http://dx.doi.org/10.1145/3445967.

[5] U. Krishnan, A. Moffat, J. Zobel, A taxonomy of query auto completion modes, in: Proc. 22nd Australasian Document Computing Symposium, ADCS, 2017, http://dx.doi.org/10.1145/3166072.3166081.

[6] R. Di Cosmo, Should we preserve the world's software history, and can we? in: Proc. 26th International Conference on Theory and Practice of Digital Libraries, TPDL, 2022, pp. 3–7, http://dx.doi.org/10.1007/978-3-031-16802-4_1.

[7] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, T. Mikolov, FastText.zip: Compressing text classification models, CoRR (2016) URL http://arxiv.org/abs/1612.03651.

[8] W. Zhang, L. Hou, Y. Yin, L. Shang, X. Chen, X. Jiang, Q. Liu, TernaryBERT: distillation-aware ultra-low bit BERT, in: Proc. 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP, 2020, pp. 509–521, http://dx.doi.org/10.18653/v1/2020.emnlp-main.37.

[9] E. Fredkin, Trie memory, Commun. ACM 3 (9) (1960) 490–499, http://dx.doi.org/10.1145/367390.367400.

[10] A. Boffa, P. Ferragina, F. Tosoni, G. Vinciguerra, CoCo-trie: data-aware compression and indexing of strings, Inf. Syst. 120 (2024) http://dx.doi.org/10.1016/j.is.2023.102316, Implementation available at https://github.com/aboffa/CoCo-trie.

[11] K. Tsuruta, D. Köppl, S. Kanda, Y. Nakashima, S. Inenaga, H. Bannai, M. Takeda, c-trie++: A dynamic trie tailored for fast prefix searches, Inform. and Comput. 285 (2022) 104794, http://dx.doi.org/10.1016/j.ic.2021.104794.

[12] A. Acharya, H. Zhu, K. Shen, Adaptive algorithms for cache-efficient trie search, in: Proc. International Workshop on Algorithm Engineering and Experimentation, ALENEX, 1999, pp. 300–315, http://dx.doi.org/10.1007/3-540-48518-X_18.

[13] D. Baskins, A 10-minute description of how Judy arrays work and why they are so fast, 2002, URL http://judy.sourceforge.net/doc/10minutes.htm.

[14] V. Leis, A. Kemper, T. Neumann, The adaptive radix tree: ARTful indexing for main-memory databases, in: Proc. 29th IEEE International Conference on Data Engineering, ICDE, 2013, pp. 38–49, http://dx.doi.org/10.1109/ICDE.2013.6544812.

[15] G. Jacobson, Space-efficient static trees and graphs, in: Proc. 30th IEEE Symposium on Foundations of Computer Science, FOCS, 1989, pp. 549–554, http://dx.doi.org/10.1109/SFCS.1989.63533.

[16] P. Ferragina, R. Grossi, A. Gupta, R. Shah, J.S. Vitter, On searching compressed string collections cache-obliviously, in: Proc. 27th ACM Symposium on Principles of Database Systems, PODS, 2008, pp. 181–190, http://dx.doi.org/10.1145/1376916.1376943.

[17] P. Ferragina, R. Venturini, Compressed cache-oblivious string B-tree, ACM Trans. Algorithms 12 (4) (2016) 52:1–52:17, http://dx.doi.org/10.1145/2903141.

[18] P. Ferragina, M. Frasca, G.C. Marinò, G. Vinciguerra, On nonlinear learned string indexing, IEEE Access 11 (2023) 74021–74034, http://dx.doi.org/10.1109/ACCESS.2023.3295434.

[19] H. Zhang, D.G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, R. Shen, Reducing the storage overhead of main-memory OLTP databases with hybrid indexes, in: Proc. ACM International Conference on Management of Data, SIGMOD, 2016, pp. 1567–1581, http://dx.doi.org/10.1145/2882903.2915222.

[20] R. Grossi, G. Ottaviano, Fast compressed tries through path decompositions, ACM J. Exp. Algorithmics 19 (2015) http://dx.doi.org/10.1145/2656332, Implementation available at https://github.com/ot/path_decomposed_tries.

[21] E.L. Miller, G.V. Neville-Neil, A. Benetopoulos, P. Mehra, D. Bittman, Pointers in far memory, Commun. ACM 66 (12) (2023) 40–45, http://dx.doi.org/10.1145/3617581.

[22] J.L. Clark, PATRICIA-II. Two-level overlaid indexes for large libraries, Int. J. Parallel Program. 2 (4) (1973) 269–292, http://dx.doi.org/10.1007/BF00985662.

[23] P. Ferragina, R. Grossi, The string B-tree: a new data structure for string search in external memory and its applications, J. ACM 46 (2) (1999) 236–280, http://dx.doi.org/10.1145/301970.301973.

[24] P. Ferragina, F. Luccio, String search in coarse-grained parallel computers, Algorithmica 24 (3–4) (1999) 177–194, http://dx.doi.org/10.1007/PL00008259.

[25] P. Ferragina, Pearls of Algorithm Engineering, Cambridge University Press, 2023, http://dx.doi.org/10.1017/9781009128933.

[26] D.R. Morrison, PATRICIA—Practical algorithm to retrieve information coded in alphanumeric, J. ACM 15 (4) (1968) 514–534, http://dx.doi.org/10.1145/321479.321481.

[27] P. Boldi, A. Marino, M. Santini, S. Vigna, BUbiNG: Massive crawling for the masses, ACM Trans. Web 12 (2) (2018) 12:1–12:26, http://dx.doi.org/10.1145/3160017, Datasets of URLs available at https://law.di.unimi.it/datasets.php.

[28] V. Lorentz, R. Di Cosmo, S. Zacchiroli, The Popular Content Filenames Dataset: Deriving Most Likely Filenames from the Software Heritage Archive, Tech. Rep., 2023, URL https://inria.hal.science/hal-04171177, preprint.

[29] N.J. Larsson, A. Moffat, Off-line dictionary-based compression, Proc. IEEE 88 (11) (2000) 1722–1732, http://dx.doi.org/10.1109/5.892708.

[30] P. Boncz, T. Neumann, V. Leis, FSST: Fast random access string compression, PVLDB 13 (12) (2020) 2649–2661, http://dx.doi.org/10.14778/3407790.3407851.

[31] Meta Platforms, Inc., RocksDB. URL https://rocksdb.org/.

[32] P. Ferragina, M. Rotundo, G. Vinciguerra, Engineering a textbook approach to index massive string dictionaries, in: Proc. 30th International Symposium on String Processing and Information Retrieval, SPIRE, 2023, pp. 203–217, http://dx.doi.org/10.1007/978-3-031-43980-3_16.

[33] D. Benoit, E.D. Demaine, J.I. Munro, R. Raman, V. Raman, S.S. Rao, Representing trees of higher degree, Algorithmica 43 (4) (2005) 275–292, http://dx.doi.org/10.1007/s00453-004-1146-6.

[34] G. Navarro, Compact Data Structures: A Practical Approach, Cambridge University Press, 2016, http://dx.doi.org/10.1017/CBO9781316588284.

[35] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inform. Theory 23 (3) (1977) 337–343, http://dx.doi.org/10.1109/TIT.1977.1055714.

[36] D. Lemire, L. Boytsov, Decoding billions of integers per second through vectorization, Softw. Pract. Exp. 45 (1) (2015) 1–29, http://dx.doi.org/10.1002/SPE.2203.

[37] I.O. Zavadskyi, Compressed unordered integer sequences with fast direct access, in: Proc. 33rd Data Compression Conference, DCC, IEEE, 2023, p. 375, http://dx.doi.org/10.1109/DCC55655.2023.00053.

[38] I.O. Zavadskyi, Binary-coded ternary number representation in natural language text compression, in: Proc. 32nd Data Compression Conference, DCC, IEEE, 2022, pp. 419–428, http://dx.doi.org/10.1109/DCC52660.2022.00050.

[39] I.O. Zavadskyi, M. Kovalchuk, Binary mixed-digit data compression codes, in: Proc. 30th International Symposium on String Processing and Information Retrieval, SPIRE, 2023, pp. 381–392, http://dx.doi.org/10.1007/978-3-031-43980-3_31.

[40] G.E. Pibiri, R. Venturini, Techniques for inverted index compression, ACM Comput. Surv. 53 (6) (2021) 125:1–125:36, http://dx.doi.org/10.1145/3415148.

[41] M.A. Martínez-Prieto, N.R. Brisaboa, R. Cánovas, F. Claude, G. Navarro, Practical compressed string dictionaries, Inf. Syst. 56 (2016) 73–108, http://dx.doi.org/10.1016/j.is.2015.08.008.

[42] A. Boffa, P. Ferragina, G. Vinciguerra, A learned approach to design compressed rank/select data structures, ACM Trans. Algorithms 18 (3) (2022) http://dx.doi.org/10.1145/3524060.

[43] P. Ferragina, G. Manzini, G. Vinciguerra, Compressing and querying integer dictionaries under linearities and repetitions, IEEE Access 10 (2022) 118831–118848, http://dx.doi.org/10.1109/ACCESS.2022.3221520.

[44] J. Arz, J. Fischer, LZ-compressed string dictionaries, in: Proc. 24th Data Compression Conference, DCC, 2014, pp. 322–331, http://dx.doi.org/10.1109/DCC.2014.36.

[45] N.R. Brisaboa, A. Cerdeira-Pena, G. de Bernardo, G. Navarro, Improved compressed string dictionaries, in: Proc. 28th ACM International Conference on Information and Knowledge Management, CIKM, 2019, pp. 29–38, http://dx.doi.org/10.1145/3357384.3357972.

[46] R. Lasch, I. Oukid, R. Dementiev, N. May, S.S. Demirsoy, K. Sattler, Fast & strong: The case of compressed string dictionaries on modern CPUs, in: Proc. 15th International Workshop on Data Management on New Hardware, DaMoN, 2019, pp. 4:1–4:10, http://dx.doi.org/10.1145/3329785.3329924.

[47] A. Silberschatz, P.B. Galvin, G. Gagne, Operating System Concepts, tenth ed., Wiley, 2018.

[48] F. Kurpicz, Engineering compact data structures for rank and select queries on bit vectors, in: Proc. 29th International Symposium on String Processing and Information Retrieval, SPIRE, 2022, pp. 257–272, http://dx.doi.org/10.1007/978-3-031-20643-6_19.

[49] S. Vigna, Broadword implementation of rank/select queries, in: Proc. 7th International Workshop on Experimental Algorithms, WEA, 2008, pp. 154–168, http://dx.doi.org/10.1007/978-3-540-68552-4_12.

[50] S. Gog, T. Beller, A. Moffat, M. Petri, From theory to practice: Plug and play with succinct data structures, in: Proc. 13th International Symposium on Experimental Algorithms, SEA, 2014, pp. 326–337, http://dx.doi.org/10.1007/978-3-319-07959-2_28.

[51] K. Sadakane, G. Navarro, Fully-functional succinct trees, in: Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, 2010, pp. 134–149, http://dx.doi.org/10.1137/1.9781611973075.13.

[52] J. Abramatic, R. Di Cosmo, S. Zacchiroli, Building the universal archive of source code, Commun. ACM 61 (10) (2018) 29–31, http://dx.doi.org/10.1145/3183558.

[53] R. Di Cosmo, S. Zacchiroli, Software heritage: Why and how to preserve software source code, in: Proc. 14th International Conference on Digital Preservation, IPRES, 2017, URL https://hdl.handle.net/11353/10.931064.

[54] S. Joannou, R. Raman, Dynamizing succinct tree representations, in: Proc. 11th International Symposium Experimental Algorithms, SEA, 2012, pp. 224–235, http://dx.doi.org/10.1007/978-3-642-30850-5_20.