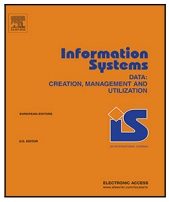
Information Systems 120 (2024) 102316

Contents lists available at ScienceDirect

Information Systems

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



CoCo-trie: Data-aware compression and indexing of strings☆

Antonio Boffa *, Paolo Ferragina, Francesco Tosoni, Giorgio Vinciguerra

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

ARTICLE INFO

Recommended by Dennis Shasha

Dataset link: https://github.com/aboffa/CoCo-trie

Keywords:
String dictionaries
Tries
Data compression
Succinct data structures
Key-value stores

ABSTRACT

We address the problem of compressing and indexing a sorted dictionary of strings to support efficient lookups and more sophisticated operations, such as prefix, predecessor, and range searches. This problem occurs as a key task in a plethora of applications, and thus it has been deeply investigated in the literature since the introduction of tries in the '60s.

We introduce a new data structure, called the COmpressed COllapsed Trie (CoCo-trie), that hinges on a pool of techniques to compress subtries (of arbitrary depth) into succinctly-encoded and efficiently-searchable trie macro-nodes with a possibly large fan-out. Then, we observe that the choice of the subtries to compress depends on the trie structure and its edge labels. Hence, we develop a data-aware optimisation approach that selects the best subtries to compress via the above pool of succinct encodings, with the overall goal of minimising the total space occupancy and still achieving efficient query time. We also investigate some variants of this approach that induce interesting space–time trade-offs in the CoCo-trie design.

Our experimental evaluation on six diverse and large datasets (representing URLs, XML data, DNA and protein sequences, database records, and search-engine dictionaries) shows that the space–time performance of well-established and highly-engineered data structures solving this problem is very input-sensitive. Conversely, our CoCo-trie provides a robust and uniform improvement over all competitors for half of the datasets, and it results on the Pareto space–time frontier for the others, thus offering new competitive trade-offs.

1. Introduction

Big data constitute a key production factor in the current digital economy. Governments and tech companies are now stipulating ambitious digital agendas aimed at overcoming the limitations of currently-known methods and technologies. These raise new challenges in the design of several modern software systems, such as next-generation green and edge computing platforms, massive key–value and media stores, high-performance networks and storage, search engines, big-data analytic tools, etc. In this paper, we concentrate on string dictionaries, which do undoubtedly constitute a core component of a plethora of big-data applications such as search engines [1–3], RDF and key–value stores [4–6], scalable distributed storage systems [7], computational biology tools [8,9], and n-gram language models [10,11], just to mention a few.

Let S be a *sorted set* of n variable-length strings $s_1, s_2, \ldots, s_n$ drawn from an alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$. The string dictionary problem consists of storing S in a *compressed format* while supporting some key query operations. One of the most powerful is the `rank` operation, which returns the number of strings in S lexicographically smaller than or equal to a pattern $P[1, p]$. Some other classic operations such as `lookup`(P) (returns a unique stringID for P if $P \in S$, and -1 otherwise), `access`(i) (returns the string in S whose stringID is i), `predecessor`(P) (returns the lexicographically largest string in S smaller than P), `prefix_search`(P) (returns all strings in S that are prefixed by P), and `longest_prefix_match`(P) (returns the longest prefix of P which is shared with one of the strings in S) can be implemented through the `rank` operation, possibly using compact auxiliary data structures [12]. In this paper, we assume that the set S is static, which allows for considerably more space–time efficient solutions compared to a scenario with in-place updates, as we will show in Section 5.4. Moreover, we note that static sorted sets of strings are core components of several dynamic and scalable key–value storage engines based on LSM-trees [13].

String dictionaries are typically approached via the *trie* data structure, which dates back to the '60s [14, §6.3]. Since then, as we will survey in Section 2, researchers have put a lot of effort into improving

☆ This is an extended version of Antonio Boffa, Paolo Ferragina, Francesco Tosoni, and Giorgio Vinciguerra. Compressed string dictionaries via data-aware subtrie compaction. SPIRE 2022. DOI:10.1007/978-3-031-20643-6_17.

* Corresponding author.
E-mail addresses: antonio.boffa@phd.unipi.it (A. Boffa), paolo.ferragina@unipi.it (P. Ferragina), francesco.tosoni@phd.unipi.it (F. Tosoni), giorgio.vinciguerra@unipi.it (G. Vinciguerra).

https://doi.org/10.1016/j.is.2023.102316
Received 2 November 2023; Accepted 8 November 2023
Available online 17 November 2023

the time and space efficiency of the naïve pointer-based implementation of tries. Some of these solutions compact paths [12,15–17] or subtrees [18–24], succinctly encode node fan-outs [25–30], apply sophisticated string transformations [31,32] or proper disk-based layouts [33–35], and some even replace the trie with learned models [36]. However, the upsurge of interest in big data, the new architectural features of modern computers, and the demanding requirements posed by applications and users have led to a revamped interest in this data structure because of some clear limitations of the previous proposals. One of the key issues addressed in recent results has been the design of compressed storage schemes for tries that offer reduced space occupancy without (much) impairing their efficient query time, see e.g. [5,16,19,22–24,30,37–40]. These proposals are offering many interesting space–time trade-offs over various datasets but without a clear winner. So choosing the appropriate storage solution is still quite a daunting task, thus requiring specific algorithm-engineering expertise and accurate analysis of the input datasets.

1.1. Our contribution

In this paper, we tackle this long-standing problem by introducing a fully-new approach to compress and index a sorted string dictionary. We start from the classic trie data structure, built on the input strings, and make the following contributions:

- We propose a compressed representation of individual subtries, and provide a concrete motivating example to show that its effectiveness depends on the "shape" of the collapsed subtrie and its possibly long "edge labels" (Section 3). This shows that approaching trie compression by collapsing its subtries in an "agnostic way", e.g. by statically fixing their height as done in prior work, is not optimal.
- In light of this, we design a new approach to trie compression that hinges upon a data-aware optimisation scheme that selects the best subtries to collapse and compress, based on a pool of known and new succinct encodings, with the overall goal of minimising the total space occupancy and still achieving efficient query time. The result is a new data structure, called CoCo-trie, which stands for COmpressed and COllapsed trie. Technically speaking, the CoCo-trie orchestrates three main tools: a novel compressed representation for collapsed subtries, a pool of succinct encoding schemes to compress the edge labels, and a data-aware optimisation procedure that selects the *best* subtries to collapse in order to minimise the overall occupied space while still guaranteeing efficient queries due to the shorter trie traversal and the efficiently-searchable encoding schemes (Section 4).
- We corroborate our theoretical results with an experimental evaluation on several datasets that offer different characteristics (because they are originating from a variety of sources, such as URLs, XML data, DNA and protein sequences, database records, and dictionaries of search engines), and comparing the CoCo-trie against five highly-engineered state-of-the-art competitors (namely, ART [29], CART [41], ctrie++ [24], FST [5], and PDT [17]). To the best of our knowledge, this is the very first work experimenting with all these implementations together, over a wide variety of datasets. Our results show that the performance of our competitors is very input-sensitive, in the sense that no solution dominates the others in space and time on all the datasets. Differently, CoCo-trie turns out to be a robust, uniformly efficient, and flexible data structure since, on half of the datasets, it improves all experimented state-of-the-art solutions, while on the others it is on the Pareto space–time frontier, thus offering new competitive and interesting trade-offs (Section 5).

Our contributions are significant *a fortiori* if one looks at them through the lenses of the about sixty years of research and development in the mature and competitive field of string dictionaries, which we advance both in knowledge and space–time performance.

For the sake of presentation, we summarise in Table 1 the main notation used throughout the paper. And, as a final note, we list below the significant new material we added to the preliminary version of this work that appeared in [42].

1. An extensive review of existing design techniques for representing the trie data structure, either explicitly or in compressed form (Section 2).
2. An analysis that generalises the example based on gamma code to the more query-efficient Elias-Fano-based code for representing the edge labels of a compacted trie (Section 4.2.1).
3. An explanation of the algorithm engineering techniques used to implement a new version of the CoCo-trie. This new version improves the space occupancy of the previous implementation (by 4.9%) and is up to 20% faster (Section 5.1).
4. New experiments with novel and bigger datasets that offer new significant insights on how the CoCo-trie and its competitors perform in real-world scenarios (Section 5.2.1).
5. An extended discussion about the effectiveness of our trie compression and compaction strategies on the experimental performance of the CoCo-trie (Section 5.3), including their impact on the construction time (Section 5.3.5).
6. Novel experiments evaluating the impact of the query workload on the time efficiency of CoCo-trie and the tested state-of-the-art competitors (Section 5.4.1).

2. Related work

String dictionaries are generally implemented via the *trie* data structure, which dates back (according to D. Knuth [14]) to 1959 due to René de la Briandais [43]. A trie, also called *digital tree* or *prefix tree*, is a type of multi-ary search tree, with edges between nodes that are labelled by individual characters (aka, uncompacted trie) or by multiple characters (aka, compacted trie). We refer to classic literature like [44] and [14, §6.3] for an introduction to the trie data structure. Herein, we describe some existing design techniques for efficiently representing the trie either explicitly or in compressed form, and comment upon their space–time performance to better contextualise our contribution.

Unary path compaction. This well-known technique consists in collapsing any unary path into a single edge labelled by the concatenation of the path labels so that each internal trie node gets at least two children. In this way, the number of trie nodes gets upper-bounded by the number of indexed strings rather than by their total length as, instead, occurs in uncompacted tries. The Patricia trie [15] is a further evolution that retains only the first character and the length of each (collapsed) edge label. This needs the introduction of the so-called *blind-search* algorithm [33] for lexicographically searching a pattern by recovering only one string of the dictionary, even if the edge labels are represented partially via their single starting characters. However, a Patricia trie acts as an index so it does not relieve the problem of storing the dictionary of strings in little space [49].

Adaptive node representations. Branching out of a node is one of the most time-consuming operations in the trie traversal for string search, because of cache-miss effects. A central design decision when engineering a trie implementation is thus the definition of the layout of an internal node. In general, such a layout should handle efficiently in space and access time different amounts of branching characters, which can vary greatly among nodes. The adaptive trie [19] addresses this issue by choosing among several, yet simple, representations of node layouts, from short arrays to hash tables, and up to B-trees depending on the node fan-out. Judy [50] uses 8 bits per branching character and,

Table 1
Summary of main notations used in the article.

Symbol	Definition
S	Input set of strings elements
n	Number of strings in the input set S
N	Number of nodes in the uncompacted trie built on the input set S
w	Machine word size (in bits)
Σ, σ	Alphabet of the strings in S, and its size
v_ℓ	Macro-node collapsing ℓ levels
$c^i_{v_\ell}$	ith integer macro-character of the collapsed macro-node v_l
m	Number of branching macro-characters
u	Universe of the branching macro-characters ($c^m_{v_\ell} - c^1_{v_\ell} + 1$)
$h(v)$	Height of the subtrie rooted at node v
$C(v_\ell)$	Space cost of encoding the collapsed macro-node v_ℓ
$\Sigma_{v_\ell}, \sigma_{v_\ell}$	Local alphabet of the symbols appearing in the edge labels of the macro-node v_ℓ, and its size
α	Parameter to relax the space cost of macro-nodes, i.e. $C(v_\ell)$, thus enabling the choice of a larger ℓ

Table 2
Datasets characteristics: number of strings in millions, dataset size in MB, average longest common prefix length, average length, maximum length, and alphabet size of the strings in the datasets.

Name	Description	$n/10^6$	Dataset size in MB	Avg lcp	Avg length	Max length	σ
`url`	URLs crawled from the web [45]	233.2	9 999.9	38.3	41.8	1 921	82
`dna`	unique 31-mers from a DNA sequence [46]	367.4	6 566.7	14.9	16.8	31	26
`tpcds-id`	customers ids in TPC-DS-3TB [47]	30.0	476.4	13.4	14.8	15	16
`trec-terms`	terms appearing in TREC GOV2 [48]	32.2	285.7	6.5	7.8	1 545	36
`protein`	sequences of amino acids [46]	2.9	171.4	36.7	53.3	16 191	26
`xml`	rows of an XML dump of dblp [46]	2.9	110.8	34.4	36.5	248	95

depending on how many children are present out of the 256 possible branches, it uses one of several inner node layouts for representing both the branching characters (e.g. via a sorted array, or a bitmap of size 256) and the corresponding pointers to children. Similarly, the Adaptive Radix Tree (ART) [29] selects among four inner node types (each designed for handling up to 4, 16, 48, and 256 children, respectively), and it uses SIMD instructions to efficiently find the branch to take. Our CoCo-trie frees from the need of engineering complex node layouts and policies to switch among them since it transforms edge labels (possibly much longer than 1 byte) into integers, which are then compressed and indexed via a set of proper integer encoders, from which we select the best one in a data- and space-aware manner on a per-node basis. In Section 5, we consider ART and its compact static version CART [5,41] for experimental comparison against our proposal.

Fan-out compaction. Differently from the adaptive node representations for the list of branching symbols and edges, some works suggested alternative implementations for the frequent and time-consuming branching step in the trie traversal. For example, the ternary search tree [26], turns an arbitrary fan-out into a ternary one: depending on how the current character in the search string compares with the one in the node (less, equal, or greater) a different branch is taken. This makes the branching out of a node easily manageable, but it requires storing three pointers per node. The double-array trie [51], instead, uses two integer arrays indexed by the node numbers that allow following a branch with a given symbol in constant time, but it does not support efficient `rank` and its space is still high to require compression [52]. Bonsai trie [25] and its descendant m-Bonsai [30] were introduced to exploit compact hash tables to represent the trie nodes, which makes branching faster but loses the lexicographic order of the string that is crucial to implement an efficient `rank` (and other operations that depend on it, such as range searches). Instead, in our CoCo-trie, we propose a fan-out compaction technique that orchestrates a sophisticated and order-preserving integer encoding of edge labels together with an optimisation strategy driven by the distribution of the edge labels involved in the fan-out.

Subtrie compaction. The high-level idea here lies in representing entire subtries via a single macro-node encoded in a proper way, e.g. packed in a few computer words or represented via a space–time efficient data structure, thereby achieving space compaction and a faster traversal of the trie structure. In this sense, the LPC-trie [18] provides a valuable example of a binary-trie implementation that compacts complete subtries located at any level of the trie into single macro-nodes. The burst trie [21] applies a similar strategy as well: it substitutes entire subtries with small "containers", such as linked lists, binary search trees, and splay trees. Its evolution, the HAT-trie [20], employs cache-conscious hash tables [53] as containers, however, it does not support efficient `rank` (and other operations that depend on it, such as range searches).

The MassTree [27] uses a B-tree to index longer substrings as macro-characters (i.e. up to 64 bits) to efficiently branch out of a trie node. We do not experiment with Masstree because [39] shows that it uses from 1.8 to 3× more space than ART.

More recently, other compaction schemes have been proposed, which exploit efficient operations carried out over machine words. For instance, the c-trie [22,23] packs $w/\log\sigma$ consecutive alphabet symbols in a single w-bit word. Or, also, the Height Optimised Trie (HOT) [39], whose salient algorithmic ingredient consists in packing a *fixed* number of nodes of a *binary* Patricia trie into a "compound node" having a fan-out equal to 32, which is then stored using a few memory words and accessed via SIMD operations. We do not experiment with HOT [39] because its implementation only supports strings whose length does not exceed 256, while our datasets contain much longer strings (see Table 2).

The latest solution featuring the packed approach is the dynamic *improved compact trie*, or ctrie++ for short [24], a hybrid between the packed c-trie [22] and the z-fast trie [54]. Like these two tries, the ctrie++ gets decomposed in a *macro* trie that includes several *micro* tries, with the latter indexing substrings with up to $w/\log\sigma$ characters. Some traits of our solution are reminiscent of this packing strategy, however, the number of characters indexed in our CoCo-trie is not fixed but varies among its trie nodes, driven by a novel data-aware space-optimisation strategy described in Section 4.2.2. In order to show the impact of this optimisation strategy, we experimentally compare our CoCo-trie against the ctrie++ in Section 5.

Path decomposition. This is a fairly powerful technique, introduced for the very first time in [55], and later extended in [12] to design a static and cache-friendly trie. The key idea consists in identifying a "good" root-to-leaf path, according to various strategies (e.g. enforcing balancedness, query-awareness, etc.) and then contracting it to a node with as many children as the subtries hanging off that path; the procedure is then carried out recursively downwards. Notwithstanding the increased fan-out could represent a limitation, researchers devised proper compression techniques that made the resulting trie asymptotically efficient in time and space [12], dynamic [16], and very efficient in practice too [17]. We include the extremely compact and efficient implementation of Path Decomposed Tries (PDT) of [17] in our experiments of Section 5.

Succinct encoding of the trie structure. Recently, numerous trie representations making use of succinct data structures have been proposed, achieving interesting theoretical and practical results. The Marisa trie [28], just to mention one of them, is a static compressed Patricia trie stored in a compressed form via the Level-Order Unary Degree Sequence (LOUDS) representation. LOUDS encodes the trie topology by visiting the nodes in level-wise order and appending 1^m0 to a bitvector B, where m is the degree of the visited node. Navigating the trie is then possible (see [56, §8.1] for details) via `rank` and `select` primitives on B.[1] Another well-known succinct representation of trees is the Depth-First Unary Degree Sequence (DFUDS), which, similarly to LOUDS, encodes the trie topology in a bitvector B but visits the nodes in preorder. DFUDS allows implementing some operations more efficiently than LOUDS, such as computing the subtree size or the number of leaves to the left of a node (see [56, §8.3] for details), so that algorithm designers choose one or the other depending on the string queries they wish to support on the indexed dictionary.

Now, the most recent and promising solution leveraging a succinct trie-structure representation is offered by the Fast Succinct Trie (FST) [5], which features query-efficient encoding schemes (LOUDS-Dense) for the upper levels of the trie and space-efficient encoding schemes (LOUDS-Sparse) for its lower levels. Because of its known space–time efficiency, we include the FST in our experiments of Section 5.

Burrows–Wheeler transform. The XBW [31] is a compressed indexing approach for labelled trees, based upon the Burrows–Wheeler Transform [58]. It achieves space occupancy up to their kth order entropy and supports efficient sub-path searches. Variants of this approach [32] have been applied to compress the Permuterm index [59] in order to support sophisticated and time-efficient wildcard queries over a dictionary of strings. Its reduced memory footprint makes its space performance similar to `bzip2`, and better than Front Coding. The LZ78-parsing scheme inspired other solutions too [38]. These succinct schemes achieve entropy bounds but unfortunately result in much slower query times compared to all the other viable solutions solving the very same problem, and thus are not experimented in this paper.

String-adapted B-tree variants. If I/O efficiency is mandatory, the choice turns out to be the String B-tree [33], which is a sophisticated combination of B-trees with Patricia tries. Recently, it was made cache-oblivious [34] and compressed [35] by combining in a sophisticated way the best-known techniques in string and labelled tree compression, as well as a proper memory layout of trees for supporting cache-oblivious trie traversals. To the best of our knowledge, an implementation of this scheme is yet to be released; this may be due to its rather complex structure.

[1] For $b = 0$ or 1, the $\texttt{rank}_b(i)$ primitive counts the number of b-bits up to position i, while $\texttt{select}_b(i)$ returns the position of the ith b-bit. Both operations can be supported in $\mathcal{O}(1)$ time using succinct auxiliary structures [57].

Non-trie solutions. The work of [37] contributed to this field with simple static and compressed string dictionaries based on front-coding, succinct structures, and full-text indexes [60,61]. Unfortunately enough, these proposals do not guarantee any good asymptotic I/O and space bounds [35]; nonetheless, they seem to be as competitive as PDT in practice. The z-fast trie (ZFT) [54,62] is a dynamic and asymptotically-efficient trie structure based upon the concept of *fat binary search*. In our view of it, ZFT can be regarded as another "non-trie solution", as it transforms a trie into a set of properly-built hash tables. This way, each prefix search is carried out with a (fat) binary search over the length of the searched string by leveraging those hash tables. Recently, the work of [40] introduced another compressed string dictionary based on a hierarchical front-coding with ideas leveraging longest common prefixes and suffix arrays to speed up searches.

We do not experiment with the implementation provided in [37,40] as we were unable to run these codebases due to some old software dependencies and incompatibilities with modern compilers. We do not experiment with ZFT as its space and lookup performance have been shown to be dominated by ctrie++ [24], which is included in our experiments.

To summarise, in light of this literature review, we will experiment in Section 5 with ART [29] and its compact version CART [5,41], ctrie++ [24], FST [5], and PDT [17], because they are either the state of the art or they offer efficient approaches to compact trie representations. To the best of our knowledge, this represents the very first work experimenting with all these implementations together over a wide variety of datasets that allow us to show the benefits and the drawbacks of all of them.

3. A motivating example

"Subtrie compaction" is a common technique in the design of compressed string dictionaries hinging on the trie data structure. It has been mainly investigated in the restricted context of either bounding the subtrie height, to fit the branching substring into one machine word [22–24], or in bounding the macro-node fan-out, so that more space–time efficient data structures can be used for it [20,21,39].

In what follows, we first introduce a novel macro-node representation of a subtrie and then provide a concrete example of the impact this technique can have on the space–time efficiency of the resulting compressed trie.

Our technique consists of properly choosing (i) the different heights of the subtries to collapse into macro-nodes, and (ii) the coding mechanism to represent the corresponding branching substrings (associated with the collapsed edge labels). In this way, the resulting trie representation adapts its space occupancy to the trie structure and to the distribution of the edge labels, while still preserving efficient time performance for the traversal operations.

The question we wish to address here is whether that *adaptive choice* is necessary or not.

Consider the tries $\mathcal{A}$ and $\mathcal{B}$ of Fig. 1 built respectively on the two sets of strings $S_1 = \{\mathrm{AG}, \mathrm{AT}, \mathrm{CA}, \mathrm{CC}\}$ and $S_2 = \{\mathrm{AA}, \mathrm{AC}, \xi\xi', \xi\xi\}$, where ξ denotes the last symbol in a (potentially large) alphabet Σ, and ξ' denotes the symbol preceding ξ in Σ. In $\mathcal{A}$, the alphabet $\{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$ consists of just 4 symbols, so we need 2 bits to represent them. In $\mathcal{B}$, the alphabet is assumed to be $\Sigma = \{\mathrm{A}, \mathrm{C}, \ldots, \xi', \xi\}$ and its symbols can be represented with $b = \lceil \log_2 |\Sigma| \rceil$ bits.

Let us now consider two scenarios for the encoding of both tries above: one in which the trie $\mathcal{T} \in \{\mathcal{A}, \mathcal{B}\}$ succinctly encodes the individual branching symbols; the other one in which the two levels of $\mathcal{T}$ are *collapsed* at the root node, thereby creating a macro-root $\mathcal{T}^c$ with branching macro-symbols of length 2 symbols. For evaluating the space cost of encoding $\mathcal{T}$ and $\mathcal{T}^c$ we consider the following succinct scheme: for every node in level order, we store the first branching symbol explicitly and then encode the *gap* between successive symbols using some coding tool, say γ-code.

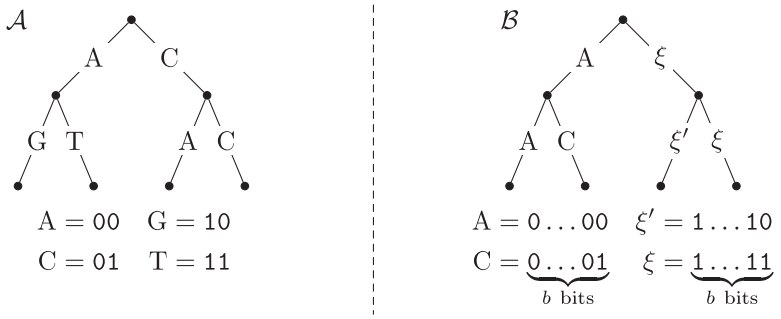


Fig. 1. Two tries $\mathcal{A}$ and $\mathcal{B}$ built on two sets of four strings: $\{AG, AT, CA, CC\}$ on the left, and $\{AA, AC, \xi\xi', \xi\xi\}$ on the right. $\mathcal{A}$ uses just four alphabet symbols, and $\mathcal{B}$ uses a much larger alphabet in which ξ' and ξ are the last two symbols.

Let us recall that the γ-code of a positive integer x consists of a number of 0s equal to the number of bits minus one of the binary representation of x, followed by that binary representation, e.g. $\gamma(6) = 00\,110$. Thus, $\gamma(x)$ takes $2\lfloor \log_2 x \rfloor + 1$ bits.

Fact 3.1. *The succinct representation of the edge labels in the trie $\mathcal{A}$ takes 9 bits. Instead, the succinct representation of the edge labels in the collapsed trie $\mathcal{A}^c$ takes 7 bits.*

Proof. The succinct representation of the edge labels in the trie $\mathcal{A}$ takes $3 + 3 + 3 = 9$ bits. In fact, the encoding of the edge labels $\{A, C\}$ of the root is $A\,\gamma(C - A) = 00\,\gamma(1) = 001$, then the encoding of the edge labels $\{G, T\}$ of the first node at the second level is $G\,\gamma(G - T) = 10\,\gamma(1) = 101$, and finally the encoding of the edge labels $\{A, C\}$ (again) of the second node at the second level is $A\,\gamma(C - A) = 001$.

If, instead, we collapse the two levels of $\mathcal{A}$ in the root of $\mathcal{A}^c$, this gets four children whose edge labels are $\{AG, AT, CA, CC\}$, and their succinct representation takes 7 bits. In fact, we encode the first branching macro-symbol as $AG = 0010$, followed by the encoding of the other three branching macro-symbols as: $\gamma(AT - AG) = \gamma(0011 - 0010) = \gamma(1) = 1$, $\gamma(CA - AT) = \gamma(0100 - 0011) = \gamma(1) = 1$, and $\gamma(CC - CA) = \gamma(0101 - 0100) = \gamma(1) = 1$.

Therefore, in terms of space occupancy, the succinct representation of $\mathcal{A}$ is worse than the one of $\mathcal{A}^c$. This result is even more evident when accounting for the space required to store the trie topology, simply because $\mathcal{A}$ has more nodes than $\mathcal{A}^c$. We conclude that, in this setting, it is better to collapse the trie as $\mathcal{A}^c$.

Surprisingly, one comes to the opposite conclusion with $\mathcal{B}$, despite having the same topology of $\mathcal{A}$. Here, the larger alphabet together with the different distribution of the edge labels changes the optimal choice.

Fact 3.2. *The succinct representation of the edge labels in the trie $\mathcal{B}$ takes at most $5b + 1$ bits. Instead, the succinct representation of the edge labels in the collapsed trie $\mathcal{B}^c$ may take up to $6b - 1$ bits.*

Proof. We can indeed represent the edge labels $\{A, \xi\}$ of the root with $A\,\gamma(|\Sigma| - 1)$ which takes at most $3b - 1$ bits; the root gets followed by the encoding of the edge labels $\{A, C\}$ of the first node at the second level, namely $A\,\gamma(C - A) = A\,\gamma(1)$ which takes $b + 1$ bits, and by the encoding of the edge labels $\{\xi', \xi\}$ of the second node at the second level, which is $\xi'\,\gamma(1)$ which also takes $b + 1$ bits.

Conversely, the succinct representation of $\mathcal{B}^c$ may take up to $6b - 1$ bits, since we encode AA with $2b$ bits set to 0, followed by $\gamma(AC - AA) = \gamma(1) = 1$, then by $\gamma(\xi\xi' - AC) = \gamma(01 \ldots 101)$ (which takes $4b - 3$ bits, because the γ-encoded number consists of $2b - 1$ bits), and finally by $\gamma(\xi\xi - \xi\xi') = \gamma(1) = 1$. □

Hence, differently from the previous example on $\mathcal{A}$, here it is better not to collapse $\mathcal{B}$ because its succinct encoding takes $b - 2$ bits less than the one of $\mathcal{B}^c$, and b can make this gap arbitrarily large, up to the point that the cost of representing their topology becomes negligible.

This example shows that there is no *a priori* best choice about which subtrie to collapse, thus opening a significant deal of possible improvements to the known trie representations. In particular, the "best" choice depends upon several features, such as the trie structure, the number of distinct branching symbols at each node and their distribution among the trie edges. Consequently, designing a principled approach to finding that "best" choice for each individual trie node is a quite complex task, that we rigorously investigate throughout the rest of the paper.

4. CoCo-trie: Compressed collapsed trie

The simplest and most used approach to collapsing tries is to obtain the trie $\mathcal{T}_\ell$ by collapsing ℓ levels of the subtries rooted at the nodes whose distances from the root of $\mathcal{T}$ are multiple of ℓ [22–24]. In this way, one can seek for a pattern $P[1, p]$ over $\mathcal{T}_\ell$ by traversing at most p/ℓ (macro-)nodes and by executing p/ℓ branches over (macro-)characters (e.g., ℓ is the number of characters that fit into a RAM word). Obviously, increasing ℓ reduces the number of branching steps, but it may increase (i) the computational cost of each individual step, given that the number and the length of the branching (macro-)characters increase; and, (ii) the space occupancy of the overall trie, given that shared paths within the collapsed subtries are turned into distinct substrings by macro-characters (see e.g. the paths "e$" and "es" descending from v in Fig. 2, which share "e").

In order to address in a principled way the above issues, we start by dealing with three main questions:

Q1: Can we tackle in an algorithmic way the issues (i) and (ii) above as ℓ increases?

Q2: How does the choice about the number ℓ of levels to collapse depend on the dictionary of strings?

Q3: Should the choice of ℓ be *global*, and thus unique to the entire trie, or should it be *local*, and thus vary among trie nodes?

These questions admit surprising answers in theory, which have equally-surprising impacts in practice. In particular, we will:

- answer **Q1** affirmatively, by resorting to a pool of succinct encoding schemes for compressing the possibly long edge labels (i.e., branching macro-characters);
- show for **Q2** that the choice for ℓ has to account for the topology and edge labelling of the trie $\mathcal{T}$, and thus the characteristics of its indexed strings;
- show for **Q3** that one has to find locally, i.e., node by node, the best value of ℓ, via a suitably-designed optimisation procedure aimed at minimising the overall space occupancy of the resulting collapsed trie.

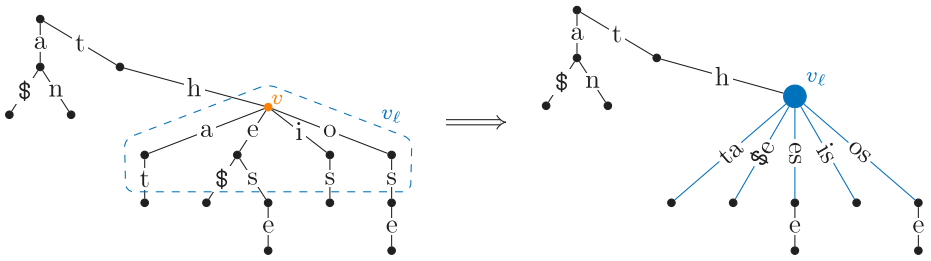


Fig. 2. Collapsing $\ell = 2$ levels of the subtrie rooted at v.

Our algorithmic answer to all these questions consists of six main steps, whose final result will be our CoCo-trie data structure. Firstly, we introduce a novel compressed encoding for the collapsed subtries (Section 4.1). Secondly, we provide an optimisation procedure that efficiently chooses the subtries to collapse within the trie $\mathcal{T}$ and compress each of them by that compressed encoding (Section 4.2). Thirdly, we empower the previous optimisation procedure by defining a *pool* of compressed encodings, thus driving the choice of the subtries to collapse also in terms of the *best* compression scheme for each of them (Section 4.3). Fourthly, we introduce a further compression step that exploits the local alphabet of the edge labels in the collapsed subtries (Section 4.4). Fifthly, we show how to introduce some flexibility in our data-aware optimisation procedure by trading space occupancy with query time (Section 4.5). And lastly, we describe how to implement the `rank` operation over the resulting compressed trie structure (Section 4.6).

4.1. Compressed encoding of collapsed subtries

Let us be given a trie $\mathcal{T}$ whose edges are drawn from an integer alphabet $\Sigma = \{0, \ldots, \sigma - 1\}$ and sorted increasingly at each node. The special character 0 (indicated with \$) is the string terminator. We formalise the notion of collapsed subtries as follows.

Definition 4.1. Given an internal node v of a trie $\mathcal{T}$ and an integer $\ell \geq 1$, the *collapsing* of ℓ levels of the subtrie of $\mathcal{T}$ rooted at v consists in replacing this subtrie with a *macro-node* v_ℓ such that (i) the edges branching out of v_ℓ are labelled with substrings which correspond to the paths of length ℓ descending from v in $\mathcal{T}$, and (ii) the children of v_ℓ are the nodes at distance ℓ from v in $\mathcal{T}$. If the branching substrings are shorter than ℓ, we pad them with the character \$.

This is depicted in Fig. 2, where five paths of length $\ell = 2$ are collapsed to form the five branching edges $\{\text{at}, \text{e\$}, \text{es}, \text{is}, \text{os}\}$ of v_ℓ.

To encode a string s branching out of v_ℓ, we initially right-pad it with $\ell - |s|$ characters \$, if $|s| < \ell$; then, we assign it the integer

$$int_\ell(s) = \sum_{i=1}^{\ell} s[i] \cdot \sigma^{\ell - i}. \tag{1}$$

Intuitively, we model every branching *macro-character* of v_ℓ as an integer $int_\ell(s)$ drawn from the integer alphabet $\Sigma^\ell = \{0, \ldots, \sigma^\ell - 1\}$. Furthermore, we observe that int_ℓ is monotonic, in the sense that, given two strings s' and s'' such that s' is lexicographically smaller than s'', then $int_\ell(s') < int_\ell(s'')$. This is an important property because we want to support lexicographic searches over the CoCo-trie, and thus properly answer the `rank` operation.

Now, suppose that v_ℓ has m branching macro-characters, which are represented as integers $c^1_{v_\ell}, \ldots, c^m_{v_\ell}$ by the int_ℓ function above. Our compression scheme explicitly encodes the first macro-character $x = c^1_{v_\ell}$ using a fixed-size representation, taking $\log \sigma^\ell$ bits,[2] and represents the other $m - 1$ macro-characters by encoding the sequence $c^i_{v_\ell} - x$ for $i = 2, \ldots, m$ with Elias-Fano (EF) [63,64], which takes $(m-1)(2 + \log \frac{u}{m-1})$ bits, where $u = c^m_{v_\ell} - c^1_{v_\ell}$ is the universe size of the sequence. To decompress the EF sequence, we also need to store some small metadata taking $\log\log \frac{u}{m} \leq \log\log \frac{\sigma^\ell}{m}$ bits.

Crucially, the speed of the `rank` operation (which we will discuss in Section 4.6) depends on the ability to perform efficient searches on the compressed sequence of macro-character, which is indeed one of the reasons we use EF. Notice that other efficiently-searchable integer encoding schemes could be used in place of EF or along with it, and indeed we do so in Section 4.3.

Summing up, the space occupancy in bits of the collapsed and compressed macro-node v_ℓ is (excluding EF's metadata)

$$C(v_\ell) = \log \sigma^\ell + (m-1)\left(2 + \log \frac{u}{m-1}\right) + 2, \tag{2}$$

where the first term corresponds to the space taken by the first macro-character, the second term accounts for the space needed to store the $(m-1)$ EF-coded integers, and the last 2 bits account for the contribution of the node v_ℓ to the space required by a succinct trie representation (we use LOUDS, see Section 2).

We underline that the subtraction of x has a subtle, yet paramount, impact on the space occupancy of our trie representation. It indeed removes any possible redundancy given by the longest common prefix (shortly, lcp) among the substrings labelling the edges branching out of v_ℓ. For instance, if we have $\ell = 2$ and the four branching macro-characters $\{\text{ha}, \text{he}, \text{hi}, \text{ho}\}$, then our encoding scheme stores $x = int_\ell(\text{ha})$ explicitly as the integer equal to $\text{h}\cdot\sigma^1 + \text{a}\cdot\sigma^0$, and it encodes the following three branching macro-characters $\{\text{he}, \text{hi}, \text{ho}\}$ as the difference with x. For example, it encodes "he" as $int_\ell(\text{he}) = (\text{h}\cdot\sigma^1 + \text{e}\cdot\sigma^0) - x = (\text{h}\cdot\sigma^1 + \text{e}\cdot\sigma^0) - (\text{h}\cdot\sigma^1 + \text{a}\cdot\sigma^0) = \text{e} - \text{a}$. So our encoding scheme stores the lcp "h" only once in x, thereby getting rid of much redundancy in the edge labels, and saving a big deal of space, especially when ℓ gets longer. As a matter of fact, we are reducing the value of the integers $c^i_{v_\ell}$, which are upper-bounded by σ^ℓ, to the values $c^i_{v_\ell} - c^1_{v_\ell}$, which are upper-bounded by $\sigma^{\ell - |lcp|}$.

4.2. On the choice of the subtries to collapse

Given the above compressed encoding of collapsed tries, we are ready to answer questions **Q2** and **Q3** of Section 4 about the choice and the number of levels of the subtries to collapse, which may vary among all trie nodes.

4.2.1. CoCo-trie with the Elias-Fano code: an analysis

First of all, we need to generalise the discussion of Section 3 about the simple γ-code, to the case of the more query-efficient, yet compressed, EF-based encoding we have introduced in the previous section.

We consider the special situation of a trie $\mathcal{T}^{(m)}$ with a very regular structure, in which each internal node has got exactly m children, with $m \leq \sigma$. We will show that, on such a regular trie, the optimal choice of ℓ depends on the parameters m and σ. On a generic trie, the situation is even more complicated, because m and σ may significantly vary at each internal node. As such, we will conclude that the optimal choice

[2] We omit ceilings for the sake of simplicity.

of ℓ cannot be *global*; hence, we will need to design an optimiser that systematically determines the value of ℓ *locally* at each trie node, with the goal of minimising the global space occupancy of the whole trie (see Section 4.2.2).

Now, given $\mathcal{T}^{(m)}$, we notice that we do not need to store its topology. Thus, collapsing a subtrie of height ℓ yields a macro-node whose representation takes a number of bits upper-bounded by

$$B_m(\ell) = \log \sigma^\ell + \left(m^\ell - 1\right)\left(2 + \log \frac{\sigma^\ell}{m^\ell - 1}\right), \tag{3}$$

where we used Eq. (2), the fact that $u \leq \sigma^\ell$ and that the number of branching macro-characters is m^ℓ.

For a given m, we can study the behaviour of $B_m(\ell)$ and thereby define an optimisation procedure that determines which value of ℓ minimises the space occupancy of the compressed trie representation. Now, under the above-mentioned assumptions, by compacting $\ell \geq 1$ levels we are actually compacting $N_m(\ell) = 1 + m + m^2 + \cdots + m^{\ell-1} = (m^\ell - 1)/(m-1)$ nodes of $\mathcal{T}^{(m)}$; in fact, at the kth level of $\mathcal{T}^{(m)}$ there are exactly m^k nodes. We can hence infer that each node of $\mathcal{T}^{(m)}$ accounts for a $1/N_m(\ell)$ fraction of a macro-node's space cost. Consequently, we can upper-bound the space for the compacted representation of each node in $\mathcal{T}^{(m)}$ as

$$\delta_m(\ell) = \frac{B_m(\ell)}{N_m(\ell)} = (m-1)\left(2 + \frac{\log \sigma^\ell}{m^\ell - 1} + \log \frac{\sigma^\ell}{m^\ell - 1}\right). \tag{4}$$

It goes without saying that a space optimiser should choose the value for ℓ which minimises $\delta_m(\ell)$.

Example. Let us fix $\sigma = 16$ and consider the 16 different regular tries $\mathcal{T}^{(m)}$ in which each internal node has got exactly m children, for $m = 1, 2, \ldots, 16$. Let us also assume that these tries are very deep. For each of these tries, we want to minimise Eq. (4) and determine the value of ℓ for which the compaction of $\mathcal{T}^{(m)}$ is space-optimal.

The case $m = 1$ is not interesting, since the trie degenerates to a simple unary path, for which every ℓ is equally space-optimal. Let us thus draw our attention to the case of $m > 1$. For a given fan-out $m = 2, 3, \ldots, \sigma$, we need to determine the value of $\ell \geq 1$ that minimises $\delta_m(\ell)$. For $\sigma = 16$, by numerical simulation is easy to get convinced that whenever $2 \leq m \leq 11$ the function $\delta_m(\ell)$ admits a global minimum at $\ell = 1$. Conversely, for $12 \leq m \leq 15$ the function $\delta_m(\ell)$ admits a global minimum at $\ell = 2$. Lastly, for $m = 16$ we have that $\delta_{16}(\ell)$ is a monotonically-decreasing function: this means that the greater ℓ is, the lower the cost for representing the regular trie. This latter is not a surprising outcome, since balanced tries with the maximum admissible fan-out ($m = \sigma = 16$) are extremely easy to compress: the optimal choice is to compact the whole trie (all levels, $\ell = +\infty$) into a single macro-node. Hence, for a trie whose alphabet size is $\sigma = 16$, we have

$$\arg\min_\ell \delta_m(\ell) = \begin{cases} 1 & \text{if } 2 \leq m \leq 11 \\ 2 & \text{if } 12 \leq m \leq 15 \\ +\infty & \text{if } m = 16 \end{cases}$$

Summing up, we have shown that for each different (regular) trie $\mathcal{T}^{(m)}$ a possibly different optimal ℓ does exist, but finding its best value is a non-trivial task, as ℓ depends on different parameters, including the alphabet dimension σ and the node fan-out m. □

4.2.2. Our optimisation approach

We now get down to the details of our data-aware optimisation scheme that, given an input a generic trie $\mathcal{T}$, identifies which subtries of $\mathcal{T}$ to collapse (and for which height ℓ each one), in order to minimise the space occupancy of its resulting representation.

Our algorithm performs a post-order traversal of $\mathcal{T}$, starting from the root. Let $h(v)$ denote the height of the subtrie rooted at v (and reaching its descending leaves in $\mathcal{T}$). For each node v, the algorithm evaluates the cost of encoding the entire subtrie descending from v by taking into account the space cost $C(v_\ell)$ of Eq. (2) referring to the subtrie of v limited to height ℓ, plus the optimal space cost $C^*(d)$ of encoding recursively the entire subtries hanging from the nodes d descending from v at distance ℓ. We vary $\ell = 1, \ldots, h(v)$, thereby determining the minimum space occupancy $C^*(v)$. Formally, if $desc(v, \ell)$ is the set of descendants of v at distance ℓ in $\mathcal{T}$ (recall that $\ell \leq h(v)$), we have

$$C^*(v) = \min_{\ell=1,\ldots,h(v)} \left\{ C(v_\ell) + \sum_{d \in desc(v,\ell)} C^*(d) \right\}. \tag{5}$$

Note that, if v is a leaf, we simply set $C^*(v) = C(v_1) = 2$, because a leaf cannot be further collapsed and its cost in the LOUDS representation is 2 bits. Clearly, because of the post-order visit, the values $C^*(d)$ are available whenever we compute $C^*(v)$.

When the root of $\mathcal{T}$ is eventually visited, the topology and the encoding of all (macro-)nodes of our CoCo-trie have already been fully determined. Thus, we know which subtries to collapse and for which height ℓ, which may vary from one subtrie to another. By Eq. (5), the resulting data structure is the space-optimal one using the encoding scheme described in Section 4.1.

The following result estimates the space–time efficiency of this optimisation approach.

Theorem 1. *The CoCo-trie of a given input trie $\mathcal{T}$ of height h and N nodes can be computed in $\mathcal{O}(Nh)$ time and $\mathcal{O}(N)$ space.*

Proof. Starting from a node v of height $h(v)$, we can compute $C(v_\ell)$ for any $\ell = 1, 2, \ldots, h(v)$ by obtaining incrementally all the optimisation parameters u_ℓ (universe) and m_ℓ (branching) from the already (inductively) known $u_{\ell-1}$ and $m_{\ell-1}$.

To compute the universe size u_ℓ for v_ℓ we need to determine the int_ℓ-code of the leftmost and rightmost length-ℓ substrings descending from v, and these can be computed by extending the respective $int_{\ell-1}$-codes computed at the previous step with one character, in constant time. This costs overall $\mathcal{O}(Nh)$ time because, for each node, we have to visit the leftmost and rightmost branching substrings that are of length at most h.

To compute m_ℓ (i.e. the number of children of the collapsed macro-node v_ℓ), we need to visit once the whole subtrie rooted at v. Knowing $m_{\ell-1}$, we add to it the number of leaves at the ℓth level. Performing for every node v a complete visit of its whole subtrie costs overall $\mathcal{O}(Nh)$ time: indeed, each of the N nodes has at most h different ancestors and thus belongs to at most h different subtries, thereby getting visited at most h times.

For every node v we maintain just the optimal C^*-cost, thus the required space amounts to $\mathcal{O}(N)$.

We finally remark that in the above optimisation scheme we can fix a constant upper-bound to the maximum number ℓ of collapsed levels so that the above time cost becomes $\mathcal{O}(N)$. This is actually the approach we take in our experimental section, where we bound ℓ for each node v by setting $h(v) = w/\log\sigma$ in Eq. (5), where w is the RAM word size in bits (see also Section 4.4). This feature may remind similar mechanisms adopted in ctrie [22,23] and ctrie++ [24], where a subtrie is packed into a machine word. However, our optimisation scheme provides more flexibility and thus it is more powerful because the height of the subtrie to collapse is not chosen in advance and equal over the whole trie, but it is adaptively chosen on a per-node basis and in a data-aware manner according to the subtrie topology and the distribution of its edge labels. Additionally, the height of the subtrie to collapse can also be tailored to the specific characters appearing in the subtrie rather than to the characters appearing in the whole trie (global alphabet), as we will see in Section 4.4.

4.3. A pool of succinct encoding schemes

Thus far, we represented the $m - 1$ integers $c^i_{v_\ell}$ representing the branching macro-characters of a macro-node v_ℓ via the EF-encoding of

the increasing sequence $c^i_{v_\ell} - x$, for $i = 2, \ldots, m$, where $x = c^1_{v_\ell}$ is the first branching character we stored explicitly. This sequence of $m-1$ integers is drawn from a universe of size $u = c^m_{v_\ell} - c^1_{v_\ell} + 1$. Depending on m, u and the values of the branching macro-characters, it may be beneficial in time, in space, or both, to resort to other kinds of encodings.

On the grounds of this observation and inspired by the hybrid integer-encoding literature [65–69], we now equip the CoCo-trie optimisation algorithm of the previous section with an assortment of encoding mechanisms so that the compressed representation of every single node can be chosen in a *data-aware* manner from them. This amounts to redefining the bit cost $C(v_\ell)$ of storing the macro-node v_ℓ so as to consider the cost in bits of other compression schemes besides EF. Specifically, when evaluating $C(v_\ell)$ during the traversal (see Eq. (5)), we select the compression scheme that gives the minimum bit-representation size for all collapsed subtries descending from v and return that bit size as the result for $C(v_\ell)$.

For our experimental study in Section 5, alongside EF, we adopt as the pool of integer encoders for the macro-characters: packed encoding (PA), characteristic bitvectors (BV), and dense encoding (DE). PA uses $\log u$ bits for each $c^i_{v_\ell}$, for a total of $(m-1)\log u$ bits. BV uses u bits initially set to 0, and then sets to 1 the $m-1$ bits corresponding to each $c^i_{v_\ell}$. DE comes into use whenever $u = m-1$, i.e. for representing a sequence of consecutive increasing integers. These encoding schemes allow us to implement the predecessor search easily, as needed by the `rank` operation in our CoCo-trie (see Section 4.6).

As a *desideratum*, we require that the bit size of the representation resulting from any of the deployed encoding schemes should be evaluated in constant time by means of the sole parameters m and u of a given node, both of which are readily available during the tree traversal (see the proof of Theorem 1). All the aforementioned encoding schemes meet this requirement.

This notwithstanding, one could employ other compression schemes that do not have a closed-form expression for their bit size (e.g. the γ-codes of the differences between macro-characters as in Section 3) either by running them and measuring the actual space occupancy or by using upper bounds for it (see e.g. [56, §2.9]). In this case, the cost stated in Theorem 1 should be re-evaluated, and could be larger.

4.4. Squeezing the universe of the macro-characters

We now describe an additional step to further decrease the space required by the compression of the macro-characters by means of an *alphabet-aware encoding*. The idea lies in replacing the encoding function int_ℓ defined in Eq. (1) and depending on the global alphabet Σ of the whole trie, with a new one that depends on the size of the local alphabet of the branching edges of the macro-node v_ℓ.

Specifically, let $\Sigma_{v_\ell} \subseteq \Sigma$ be the alphabet of symbols occurring in the edge labels of the collapsed macro-node v_ℓ. For example, in Fig. 2 we have $\Sigma = \{\$, \mathrm{a}, \mathrm{e}, \mathrm{h}, \mathrm{i}, \mathrm{n}, \mathrm{o}, \mathrm{s}, \mathrm{t}\}$ and $\Sigma_{v_\ell} = \Sigma \setminus \{\mathrm{h}, \mathrm{n}\}$. By changing $\sigma = |\Sigma|$ in Eq. (1) with $\sigma_{v_\ell} = |\Sigma_{v_\ell}|$, we can squeeze the size of the universe of the branching macro-characters of v_ℓ from σ^ℓ to $\sigma^\ell_{v_\ell}$. This, in turn, reduces the magnitude and the distance between consecutive integers associated with the branching macro-characters and thus allows a more effective compression. Also, we reduce the first term of Eq. (2) to $\log \sigma^\ell_{v_\ell}$.

Clearly, each macro-node v_ℓ adopting this optimisation must store a mapping between Σ and the local alphabet Σ_{v_ℓ}, which is implemented with a bitvector $B[0, \sigma - 1]$ where $B[i] = 1$ if symbol i appears in Σ_{v_ℓ} [70]. In practice, σ is typically small (e.g. 256 for 8-bit ASCII alphabets) so that rank/select operations over B can be executed without additional space via a few bit-manipulation instructions (see e.g. [71, §3.2]).

Of course, this optimisation requires modifying $C(v_\ell)$ to account for both the more efficient macro-characters representation due to the squeezed universe and the size of the alphabet mapping (i.e. σ bits).

Overall, the time complexity for building the CoCo-trie becomes $\mathcal{O}(Nh^2)$ since we cannot compute u_ℓ incrementally as described in the proof of Theorem 1; the space complexity instead does not change, as we do not store the bitmaps $B_\ell[1, \sigma]$, but we compute them incrementally while visiting the subtries as in the proof of Theorem 1. This time complexity might appear prohibitive, but in Section 5 we will show that the practical building time is reasonable and worth achieving excellent and Pareto optimal performance at query time.

4.5. Dealing with the space–time trade-off

Under some scenarios, it might be of interest to slightly readjust our optimisation procedure to take into account the query performance too, while possibly giving up just a little of the space optimality. To accomplish this space–time trade-off, we rely on the intuition that the more levels are collapsed, the faster a trie traversal will be. But, on the other hand, as we collapse more levels, the fan-out of each macro-node increases and so the time to branch out of each individual macro-node increases as well. However, we experimentally observed that this is not a major concern because our compressed encoding of collapsed subtries is in practice extremely efficient to be navigated; thus the time reduction given by increasing the number of collapsed levels dominates the increased access time due to the larger node fan-out.

With this in mind, we modify the algorithm of Section 4.2 to relax the search for the minimum space occupancy as follows. At each visited internal node v, we compute $C^*(v)$ as usual and denote by ℓ^* the value of ℓ minimising the right-hand side of Eq. (5). Then, we find the largest value $\ell \in \{\ell^*, \ell^* + 1, \ldots, h(v)\}$ that allows representing the collapsed node within a constant factor $\alpha \geq 0$ more than $C^*(v)$. Note that this new approach has no impact on the construction complexity. We experiment with it in Section 5, where α will be expressed as a percentage.

4.6. Query operations

The operation `lookup`(P) in the CoCo-trie begins from the root macro-node v_ℓ by computing the integer $y = int_\ell(P[1, \ell]) - c^1_{v_\ell}$. Then, we seek for y into the increasing sequence $c^i_{v_\ell} - c^1_{v_\ell}$, for $i = 2, \ldots, m$: if the search fails, we return -1; otherwise, we obtain an index j of y, and proceed with the recursion in the jth child of the macro-node. We iteratively consume multiple characters at once from the pattern P as we descend the CoCo-trie via LOUDS. When P is exhausted, we return the unique LOUDS index of the node we reached.

As for the operation `rank`(P), we switch to the DFUDS encoding for the trie topology as it allows us to compute the rank of a leaf efficiently, takes the same space of LOUDS, and it is still efficient in navigating the trie downwards [56, §8.3]. At each macro-node v_ℓ corresponding to the pattern substring $P[k, k + \ell - 1]$, we compute the integer $y = int_\ell(P[k, k + \ell - 1]) - c^1_{v_\ell}$ and seek for the largest index j such that $c^j_{v_\ell} \leq y$, and we keep searching recursively into the jth child of v_ℓ. In the case the pattern substring we are considering in v_ℓ is shorter than ℓ, i.e. $k + \ell - 1 > |P|$, then we proceed downwards to the rightmost descendant of the jth child of v_ℓ (recall that int_ℓ right-pads the given pattern suffix with \$ characters until it has length ℓ). In any case, we eventually reach a leaf node and return efficiently its rank thanks to the DFUDS encoding of the trie structure.

For solving the operation `access`$(i) = s$ with DFUDS, we jump in constant time to the ith leaf and reconstruct s backwards as the navigation proceeds to the root. Specifically, at each macro-node v_ℓ encountered along the upward path, we decode the macro-character $c^j_{v_\ell}$, where j is the position of the previously-visited macro-node among v_ℓ's children.

For solving the operation `prefix_search`(P) with DFUDS, we proceed downwards from the root. At each macro-node v_ℓ corresponding to the pattern substring $P[k, k + \ell - 1]$, we compute the integer $y = int_\ell(P[1, \ell]) - c^1_{v_\ell}$ and seek for y into the increasing sequence $c^i_{v_\ell} - c^1_{v_\ell}$: if the search fails, we return an empty string set, meaning that no

strings in S are prefixed by P. In the case the pattern substring we are considering in v_ℓ is shorter than ℓ, i.e. $k + \ell - 1 > |P|$, then we return all strings in the subtries descending from the macro-characters $c^a_{v_\ell}, \ldots, c^b_{v_\ell}$ of v_ℓ whose prefixes match with $P[k, |P|]$; a pre-order visit of the corresponding $b - a + 1$ subtries allows to retrieve the string set in lexicographic order.

Finally, for both `lookup`(P) and `rank`(P), we observe that the step that searches for y in a macro-node can be efficiently supported in all the encoding schemes introduced in Section 4.3. Indeed, if the macro-node stores $m - 1$ elements in a universe of size u, EF allows implementing the search in $\mathcal{O}(\log \frac{u}{m})$ time using a well-known algorithm based on binary search [56, §4.4.2], PA supports it in $\mathcal{O}(\log m)$ time via a standard binary search, BV supports it in $\mathcal{O}(1)$ time using succinct auxiliary structures (recall Footnote 1), and DE supports it in $\mathcal{O}(1)$ time since the child to descend to is given by the value of y. Moreover, for `access`(i), we observe that the step of decoding a macro-character $c^j_{v_\ell}$ can be done $\mathcal{O}(1)$ time in all three encoding schemes, since they support random access according to j.

5. Experiments

We now experimentally show that:

1. The choice of the subtries to collapse differs significantly from dataset to dataset and from node to node. This provides a clear and concrete answer to questions **Q2** and **Q3** in Section 4, thus motivating our study.
2. The alphabet-aware compression of macro-nodes (Section 4.4) improves the space of CoCo-trie by up to 38% and the query time by up to 29%.
3. The technique that allows us to trade off space with query time via a parameter α (Section 4.5) improves the query time of CoCo-trie by up to 33% at the cost of a slight increase in space (no more than 24%).
4. The performance of our highly-engineered competitors is very input-sensitive, in the sense that no solution dominates the others in space and time on all the datasets.
5. With respect to our competitors, the CoCo-trie results space–time efficient, robust, and flexible: in fact, on three datasets it significantly improves the space–time performance of all competitors whereas in the three other datasets it is on the space–time Pareto front of the best competitors (thus offering other competitive and interesting trade-offs).

Given the variety of the datasets (six) and highly-engineered competitors (five) we experimentally test in this section, the above results suggest that our CoCo-trie may be regarded as the state-of-the-art reference for the static string dictionary problem.

Readers interested in reproducing our experiments can find the source code and the datasets at https://github.com/aboffa/CoCo-trie.

5.1. Implementation notes

Our implementation of the CoCo-trie is composed of three main data structures: denoted with L, E, and F. The first one is the bit-array L containing the succinct LOUDS representation of the (collapsed) trie topology. The second one is the array E containing, for each macro-node v_ℓ, the first branching macro-character $c^1_{v_\ell}$ followed by the encoding of the remaining macro-characters $c^i_{v_\ell} - c^1_{v_\ell}$, the information needed to decode them (i.e. a field representing which encoding has been selected from the ones described in Section 4.3, and the alphabet in σ bits if the solution of Section 4.4 is used for v_ℓ), the value of ℓ (Section 4.2), and a flag indicating whether the string labelling the path from the trie root to v is in the dictionary (since a string in the dictionary could be the prefix of another). The last data structure is the array F containing, for each macro-node v_ℓ, a pointer to E where the encoding of $c^1_{v_\ell}$ starts. We random-access F via the node identifiers implicitly given by the succinct LOUDS representation in L.

As we mentioned at the end of Section 4.2, we bound the number of levels to be inspected by the optimiser so that the result of int_ℓ does not exceed the word size w of the machine. This can be achieved by setting $\ell < \lceil w / \log \sigma_{v_\ell} \rceil$ for each macro-node v_ℓ. For example, in our 128-bit (`uint128_t`) implementation, we can collapse into a single macro-node a (sub)trie built over an alphabet size $\sigma = 4$ and having a height at most $\ell = 128/2 = 64$ (indeed this happens in practice, see Fig. 3).

The CoCo-trie is built on the sdsl library [72],[3] ds2i library [73], and sux library [74]. To navigate the succinct LOUDS representation of the collapsed trie topology (contained in the bitvector L) the following operations are needed: `nodemap`, `nodeselect`, `leafrank`, and `child` [56, §8.1]. These operations can be implemented with constant-time rank/select operations on the bitvector L. More precisely, from the sdsl library, we use `sdsl::rank_support_v` for $\texttt{rank}_1$ and $\texttt{rank}_{00}$. From the sux library, we use `sux::SimpleSelectZero` for the implementation of $\texttt{select}_0$. Recent studies [75] show that these implementations are the fastest, and thus the CoCo-trie is extremely fast in navigating the topology of the collapsed trie. Moreover, the pointers in the array F are stored in a `sdsl::int_vector` whose entries take $\lceil \log(\#\text{macro-nodes}) \rceil$ bits each. From the ds2i library, we leverage the efficient Elias-Fano implementation in `ds2i::compact_elias_fano`.

Finally, we write *CoCo-trie* α% to denote the space-relaxation technique of Section 4.5 with parameter $\alpha \geq 0$ ($\alpha = 0$ corresponds to the space-optimal solution).

5.2. Experimental setting

We perform our experiments on a machine equipped with a 2.30 GHz Intel Xeon Platinum 8260M CPU and 384 GiB of RAM, running Ubuntu 20.04.3. We compile our codebase using `g++-11.1` and the `C++-20` language standard.

5.2.1. Datasets

We aimed at choosing very diverse datasets in terms of sources (such as URLs, XML data, DNA and protein sequences, database records, and dictionaries of search engines) and features (such as number n of strings, total size in MB, alphabet size σ, and average/maximum length of the lcp between consecutive sorted strings) to capture and depict a broad spectrum of performance among the tested string dictionaries. All strings in each dataset are different.

We will experiment with the following six datasets:

`url` contains URLs of a Web crawl taken in 2015, starting from the site europa.eu without any domain restriction [45].[4] Storing a set of URLs into a string dictionary is fundamental for crawlers [59]. URLs datasets have also been used in [16,17,24,37,40], but ours is one order of magnitude larger.

`dna` consists of all the unique substrings of $k = 31$ bases from a DNA sequence obtained from files available at the Gutenberg Project site: namely, from 01hgp10 to 21hgp10, plus 0xhgp10 and 0yhgp10.[5] These unique substrings are called k-mers, and storing them into a string dictionary allows performing subsequent efficient k-mer-prefix searches (and suffix searches, if the dictionary is built on their reversal) and k-mer counting (even on a prefix/suffix of length $l < k$). Datasets with k-mers have also been used in other evaluations [16,37].

[3] We use the https://github.com/vgteam/sdsl-lite fork.
[4] https://law.di.unimi.it/webdata/gsh-2015/
[5] http://pizzachili.dcc.uchile.cl/

`tpcds-id` are strings of 16 characters taken from the `c_customer_id` column of the `customer` table in the TPC-DS-3TB dataset [76].[6] Many research papers on query processing and optimisation use the TPC-DS standard benchmark [77, 78], and we note that storing database columns into string dictionaries allows solving range queries and joins [79].

`trec-terms` is the set of terms appearing in the text of the TREC GOV2 collection. The space–time efficient indexing of a set of terms appearing in a collection of Web documents is a fundamental task in Information Retrieval [59]. This dataset has also been used in [48].

`protein` contains different sequences of amino acids.[5] This dataset has also been used in [24].

`xml` is a part of the XML dump of the dblp.org website.[5] This dataset has also been used in [24].

We preprocess each dataset to keep, for each string s, the shortest prefix of s that distinguishes it from all the other strings in the dataset. This suffices to index the dictionary strings and support the `rank` operation as well as all other query operations mentioned at the beginning of this paper. In fact, it is well known that, for any trie-based data structure, the remaining suffixes can be concatenated and compressed into a separate array, and efficiently retrieved when needed [56, §8.5.3]. Table 2 summarises the datasets and their characteristics after this preprocessing.

5.2.2. State-of-the-art competitors

As argued in Section 2, we consider as competitors of our CoCo-trie the following (static) string dictionary implementations because they are either the state-of-the-art or they offer space–time efficient approaches to trie representations:

CART: a compact version of ART [29] obtained by constructing a plain ART and then converting it to a static compact version [5,41, 80].

PDT: the Centroid Path Decomposed Trie [17]. We experiment with both the vbyte version that encodes the labels of the edges with vbyte [81], and the csp version that adds another layer of compression on top of the edge labels.

FST: the Fast Succinct Trie [5]. We use a slightly-modified code [82] that solves lookup queries rather than range-query filtering. We show the full space–time performance of FST by varying its parameter $R = 2^i$ for $i = 0, \ldots, 10$.

We also tested the following representatives of the dynamic approaches, which will allow us to show that the restriction to a static scenario allows for considerably more space–time efficient string dictionaries:

ART: the Adaptive Radix Tree [29].

ctrie++: the improved compact trie [24].

5.2.3. Query workloads

Given that our competitors do not implement `rank` (despite their design does support it), we decided to measure the performance of `lookup`. We can reasonably expect that `rank` would perform similarly to `lookup`, because of the way the former can be derived from the latter in trie-based (rather than hash-based) data structures, as the ones we experimentally test here.

Given a dataset of n strings, we measure the `lookup` time by averaging the performance of 3 repetitions of a batch of n string searches, where half of the strings are taken from the datasets and half are generated randomly. Changing this proportion impacts the performance of the tested trie-based string dictionaries, so we vary it in an experiment in Section 5.4.1.

To generate these random strings not in the datasets, we (i) extract a randomly-chosen string belonging to the dataset and truncate it to the average lcp of the entire dataset, and (ii) append to it a random string whose length matches the average length of the strings in the dataset. This way, the queries we generate mimic a fair query workload that guarantees a balance between existent and non-existent queried strings, and for the latter that the traversal does not stop at the very first steps because of a mismatch.

5.3. An analysis of the CoCo-trie design

In this section, we dig into an analysis of the effectiveness of the individual techniques and optimisations adopted in the design of CoCo-trie.

5.3.1. Distribution of the number of collapsed levels

Fig. 3 shows our first experimental result: the number of macro-nodes collapsing a certain amount of levels forms a non-trivial distribution whose shape differs from dataset to dataset. This provides a clear concrete answer to both questions **Q2** and **Q3** posed in Section 4: the number of levels to be collapsed in a subtrie greatly depends on the indexed strings, and it must be chosen locally on a per-node basis. Therefore, the data-aware optimisation approach to subtrie compaction implemented in our CoCo-trie is essential to achieve the most from these features.

In particular, on `url`, `xml`, and `trec-terms` the CoCo-trie optimiser selects many times the lowest possible values of ℓ (each horizontal axis ranges from $\ell = 1$ to the largest ℓ over all macro-nodes v_ℓ). On `dna`, the CoCo-trie optimiser collapses at most $\ell = 23$ levels at a time, selects $\ell = 1$ for 33% of the time, and a value between 2 and 15 63% of the time. For `protein`, instead, the CoCo-trie optimiser selects high values of ℓ (very often $\ell \approx 30$) so that, in the end, the distribution resembles a Gaussian one. The results on `tpcds-id` are also of interest for their simplicity: due to the regularity of this dataset, the CoCo-trie optimiser creates a macro-node for the root that collapses $\ell = 11$ levels, and each of its 4096 children collapses $\ell = 4$ levels.

5.3.2. Effectiveness of the alphabet-aware encoding

An important feature of our CoCo-trie is the alphabet-aware encoding of macro-nodes described in Section 4.4. Our experiments indeed suggest that operating on the smaller alphabet Σ_{v_ℓ} at each macro-node v_ℓ, rather than on the global alphabet Σ, can simultaneously save a considerable amount of space and decrease the query time too. Specifically, on the `dna` dataset, this alphabet-aware encoding technique makes the CoCo-trie use 38% less space and be 16% faster in answering queries. On `url`, it obtains a 3% improvement in space and a 29% improvement in query time. On `protein`, it obtains a 7% improvement in space and a 5% improvement in query time. On `xml`, it obtains a 4% improvement in space and a 14% improvement in query time. On `trec-terms`, it obtains a 12% space improvement and a 27% improvement in query time. Lastly, we notice that the dataset `tpcds-id` does not benefit from this technique, as indeed our optimiser does not choose the alphabet-aware encoding for any node.

As a result, since the alphabet-aware encoding of macro-nodes is either effective or does not harm the performance of the CoCo-trie, we always enable it in the following experiments.

[6] http://www.tpc.org/tpcds/

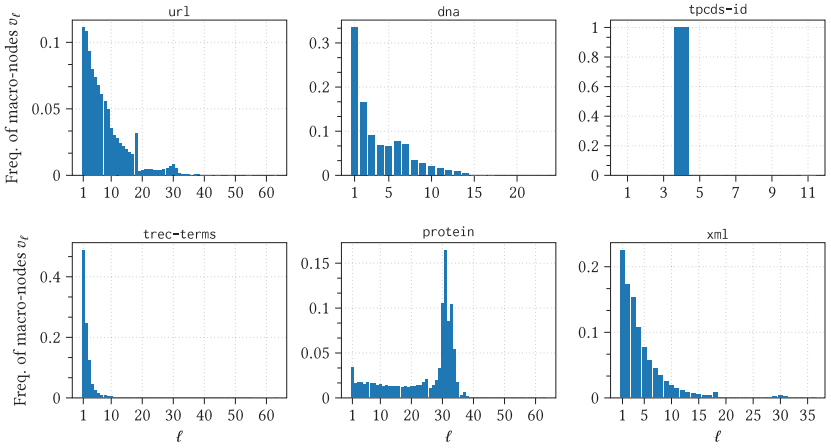


Fig. 3. Normalised frequency of macro-nodes obtained by collapsing subtries consisting of ℓ levels. Greater values of ℓ yield fewer but (individually) more space-consuming macro-nodes. The CoCo-trie optimiser chooses ℓ on a per-node basis to minimise the space taken by the overall compressed trie, and this choice is highly dependent on the features of the input strings, as these plots show.

5.3.3. Effectiveness of the space-relaxation technique

The space-relaxation technique described in Section 4.5, albeit simple, performs very well in practice. We evaluate it by testing the values $\alpha = 5, 10, 15$, and 20%, and by measuring the change in the average number of collapsed levels, the space occupancy, and the query time of the resulting CoCo-trie (these last two measures are also reported in Fig. 5, and commented next).

On `url`, by going from a space-optimal solution ($\alpha = 0\%$) to the space-relaxed solution with $\alpha = 5\%$, we increase the average number of collapsed levels by 3.13% (i.e., from 8.65 to 8.92). This small improvement has a considerable impact on the query time, which improves by 15%, at the cost of a 16% increase in space. As we further increase α from 5% to 10%, we notice another 3.3% improvement in the average number of collapsed levels, which in turn induces another 2.6% improvement in query time at the cost of an 8% increase in space. For further increments of α (i.e., beyond 10%), there is no significant improvement in the query time.

On `dna`, by going from the space-optimal solution with $\alpha = 0\%$ to the space-relaxed solution with $\alpha = 5\%$, we increase the average number of collapsed levels by 3% that, in turn, significantly improves the query time by 23% at the cost of a 6% increase in the total space occupancy. Further increasing α from 5% to 20% does not change the performance much, the query time increases too and thus the space relaxation is of no help.

On `tpcds-id`, the space–time performance of the CoCo-trie remains nearly the same for all tested values of α.

On `trec-terms`, by increasing α from 0% to 5% we incredibly improve the query time by 33%, at the cost of increasing the space usage by only 4%. Further increasing α from 5% up to 20% slightly improves the query time (up to 6%) but it also increases the space occupancy (up to 12%).

On `protein`, by going from the space-optimal solution with $\alpha = 0\%$ to the space-relaxed solution with $\alpha = 5\%$, we increase the average number of collapsed levels by 7% (from 24.3 to 26.0). In turn, this improves the query time by 19% at the cost of a 9% increase in space. In this dataset, if we further increase α to 10%, 15%, or to 20%, we do not observe significant improvements in query time.

Finally, on `xml`, by going from the space-optimal solution with $\alpha = 0\%$ to the space-relaxed solution with $\alpha = 5\%$, we increase the average number of collapsed levels by 4.3%. In turn, this improves the query time by 15% at the cost of an 8% increase in space. If we further increase α up to 20%, the query time keeps decreasing achieving especially good time performance.

Looking at these results, we suggest to the final user of the CoCo-trie, who wants to use a little bit more space but solve the query faster, to set $\alpha = 10\%$. This value of α has shown the best overall performance on this variegated collection of datasets.

5.3.4. Breakdown of the space taken by the CoCo-trie components

Fig. 4 shows the percentage of the space occupied by the various components of the CoCo-trie: the node structure (which includes, for each macro-node v_ℓ, the value of ℓ, a flag indicating whether the string labelling the path from the root to v_ℓ is in the dictionary, and the pointer to the encoding of the branching macro-characters of v_ℓ), the topology (i.e. the LOUDS representation of the collapsed trie), the first macro-characters of each macro-node (encoded using $\ell \log \sigma$ bits), the first macro-character of each macro-node with its alphabet (encoded using $\ell \log \sigma_{v_\ell}$ bits), and the macro-characters following the first one that are encoded with one of the succinct encoding schemes we described in Section 4.3.

We can notice that the doughnut charts for the different datasets are very diverse. This supports our thesis that CoCo-trie is able to adapt to the very varied characteristics of the tested string dictionaries over which it is built.

First of all, let us discuss the `tpcds-id` dataset. Due to its regularity, the CoCo-trie space occupancy is mainly made of topology and compressed labels. As discussed in Section 5.3.1, our optimiser creates a macro-node for the root that collapses $\ell = 11$ levels, and each of its 4096 children collapses $\ell = 4$ levels. CoCo-trie internal macro-nodes contribute to the node structure, first macro-characters and topology components; instead, the leaves solely contribute to the topology component. So we have just 4097 internal nodes for 30×10^6 leaves and that explains why the space occupied by the node structure and first macro-characters is extremely low (0.06% and 0.03% respectively) and the space for topology is so high. This is the only dataset where the alphabet-aware encoding has not been applied to any node, and this is why the percentage of space occupied by first macro-characters remap and compressed macro-characters with the alphabet-remap encoding is 0%.

On all the other datasets, the space usage of the topology ranges from 5.1% to 24.4%, and the node structure occupies from 15.2% to 25.1% of the total compressed space. Fig. 4 shows that in all these cases the alphabet-remapping technique, introduced in Section 4.4, is very effective since our optimiser often selects to encode the first macro-character using the local alphabet Σ_{v_ℓ}. In fact, the space usage of the first macro-characters compressed with the alphabet-aware encoding is almost equal to (on `trec-terms`) or more than the one without this technique. Similarly, the space usage for compressing the sequence with the other macro-characters (i.e., not the first ones) with the alphabet-aware encoding is more than the one without this technique on all datasets but `xml`.

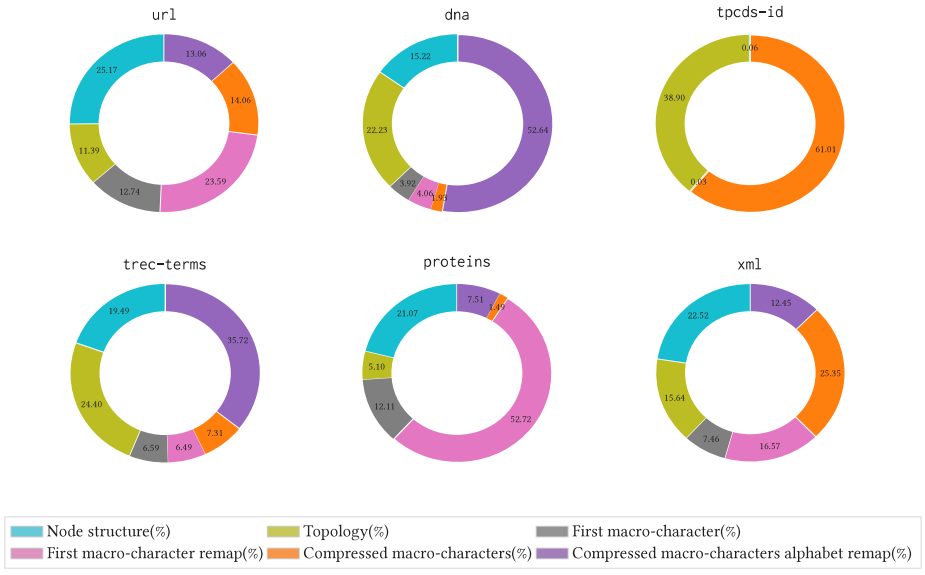


Fig. 4. Percentage of space occupied by the various components of the CoCo-trie built on top of the selected datasets.

5.3.5. Construction time

As discussed in Section 4.4, the construction of the CoCo-trie with the alphabet-aware encoding runs in $\mathcal{O}(Nh^2)$ time. This may seem prohibitive for growing values of h, as indeed we have considered $h = 64$ in Section 5.1. Nonetheless, in practice, given an uncompacted trie, building the CoCo-trie is from 3× to 12× slower than the closest solution on the Pareto front for each dataset (as commented in the next section). This slowdown is smaller than the expected one (i.e., $\mathcal{O}(h^2)$) and it may be due to the existence of many subtrees whose height is much smaller than 64, as in the case of the numerous leaves for which $h = 0$. More precisely, on `url` CoCo-trie is 7.2× slower than PDT(csp) (which requires 1.65 µs/key), on `dna` it is 5.2× slower than FST (0.67 µs/key), on `tpcds-id` it is 4.2× slower than FST (0.36 µs/key), on `trec-terms` it is 3.9× slower than FST (0.34 µs/key), on `protein` it is 12× slower than PDT(csp) (7.7 µs/key), and on `xml` it is 3.0× slower than PDT(csp) (1.1 µs/key). We argue that paying, on average, 6× more construction time is well rewarded by having Pareto-optimal performance at query time, no matter the characteristics of the underlying dataset, and especially in a scenario of build-once-query-many-times is considered. Yet, we believe one may speed up the construction with a more space–time efficient representation of the input uncompacted trie navigated by the CoCo-trie optimiser. This input trie is currently pointer-based, potentially leading to a cache miss for each visited node. Compacted trie representations, as the ones discussed in Section 2, could provide viable solutions to address this issue; however, we prefer to leave these implementation details to future work.

5.4. Space–time performance comparison

Fig. 5 shows the results about the space and time performance of the CoCo-trie and of the static competitors: note that the more the performance of a data structure is close to the bottom-left corner of a picture, the best it is. Moreover, we point out that both y- and x-axis change with the datasets, whose original size in MBs is shown in each plot title.

CART, though fast, is generally very space-inefficient. On `url` it is also particularly slow because of the large size of the dataset and of the large average-lcp among the indexed strings, which causes longer trie traversals and thus more cache misses.

Whichever is the setting of its parameter R, FST is dominated in space and time performance by our CoCo-trie (and also by other data structures) on all the datasets. This is especially evident on `dna`, `tpcds-id`, `protein`, and `xml`. We argue that this is due to the high average-lcp of the strings in these datasets that require FST to perform longer trie traversals that proceed one character at a time (indeed, FST does not compact unary paths).

PDT is always among the most space-efficient data structures (especially in its csp configuration, which uses a form of grammar compression on the edge labels), but its query time performance is not competitive on `dna`, `tpcds-id` and `trec-terms`. This can be explained by looking at the average height of the nodes in PDT and CoCo-trie. In fact, PDT nodes on `dna`, `tpcds-id` and `trec-terms` have an average height equal to 6.0, 5.4, and 9.6, respectively. Conversely, the average height of the macro-nodes in the CoCo-trie (see the $\alpha = 0\%$ configuration in Fig. 5) is instead 3.5, 2.8, and 5.25. This means that the CoCo-trie requires nearly half of the accesses to nodes, on average. And in fact, CoCo-trie is 2.6× faster than PDT(csp) on `dna`, 2.3× faster on `trec-terms` using its same space, and it is 2.6× faster and 1.8× more succinct than PDT(csp) on `tpcds-id`. This means that, on these three datasets, our subtrie collapsing-and-compression technique is extremely effective not only in reducing the space occupancy but also in speeding up the query operation. On the other three datasets, CoCo-trie always has some configuration on the Pareto front of PDT (mainly improving the time efficiency), thus offering other competitive space–time trade-offs.

Compared to CART, FST, and PDT, which are very input-sensitive as no solution dominates the others in space and time on all the datasets, the CoCo-trie results space–time efficient, robust, and flexible over all six datasets: in fact, it significantly dominates the space–time performance of all competitors on `dna`, `tpcds-id`, and `trec-terms`; and it is on the Pareto front of the best competitors on `url`, `protein`, and `xml` by offering faster query operations.

Finally, we show the performance of our dynamic competitors, ART and ctrie++, in the separate Table 3 because they are far from the plot range of Fig. 5. As the table shows, their space usage is from 2.2× to 130× larger than CoCo-trie while being only slightly faster on `trec-terms` and `protein`. This increased space usage is expected due to the extended functionality of the dynamic competitors.

5.4.1. Impact of the query workload

In this last experiment, we dig into the time performance of the tested data structures over the various query workloads described in

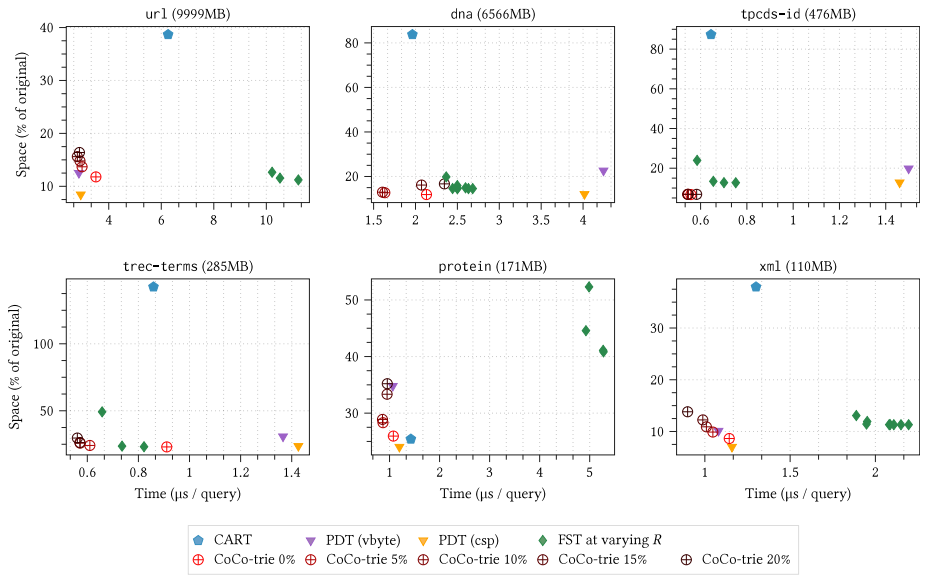


Fig. 5. CoCo-trie dominates the space–time performance of all static competitors on the `dna`, `tpcds-id`, and `trec-terms` datasets; and it is on the Pareto front of the best competitors on `url`, `protein`, and `xml`.

Table 3
Space–time performance of the dynamic competitors, ART and ctrie++, compared to CoCo-trie.

Dataset	ART query time	ART space usage	ctrie++ query time	ctrie++ space usage
`url`	1.04×	6.45×	1.02×	27.36×
`dna`	1.29×	15.90×	1.18×	81.24×
`tpcds-id`	1.19×	42.50×	1.30×	130.76×
`trec-terms`	1.16×	11.19×	0.98×	50.42×
`protein`	1.54×	2.21×	0.89×	12.86×
`xml`	1.02×	6.80×	1.09×	31.99×

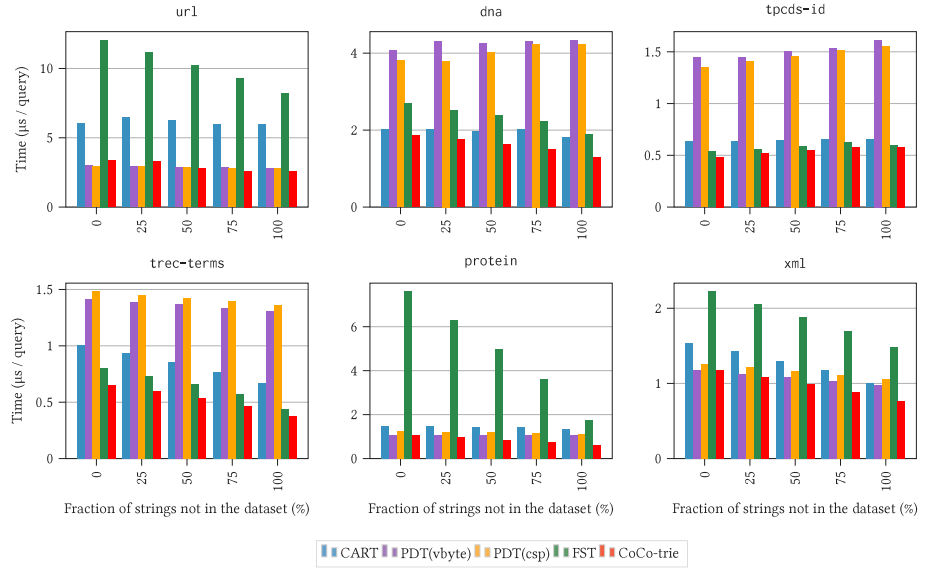


Fig. 6. Time performance of the lookup query for the fastest tested solutions as we vary the percentage of queries for strings not in the dataset (generated as described in Section 5.2.3).

Section 5.2.3 and generated by varying the percentage of strings taken from the datasets or generated randomly from 0% to 100% (in Fig. 5 this percentage was set to 50%). The results are shown in Fig. 6. For CoCo-trie and FST, we report the most time-efficient configuration by varying α and R, respectively.

Firstly, one may expect the query time to decrease as the amount of random strings increases, since the trie traversal should stop earlier due to a missing branch. This happens for all the tested implementations on `url`, `xml`, and `protein`, but it does not happen on `dna` and `tpcds-id`. This is due to the interplay between the size of the alphabets of the strings in these two latter datasets, and the fact that their lcp is close to the average length. So that, there is a high probability of randomly-generating strings that share a long lcp with some other dictionary string. As such, we observe an increased query time in all

tested solutions (this is particularly evident for PDT), since each trie needs to traverse multiple levels for solving those queries.

Secondly, the figure suggests also that the alphabet-aware encoding introduced in Section 4.4 is particularly effective when we query the CoCo-trie over many non-dictionary strings. Indeed, during the traversal we check whether the query substring corresponding to the current macro-node intersects with the macro-node's local alphabet so to instantly recognise an unsuccessful `lookup`, thereby achieving faster performance compared to our competitors. Nevertheless, the search in the CoCo-trie could continue to compute `rank`, as described in Section 4.6.

Overall, the CoCo-trie is the fastest on all datasets and query workloads except on `url`, where it is nonetheless the fastest for a large number of randomly-generated query strings.

6. Conclusions and future work

We have introduced a new trie-based design of compressed string dictionaries that collapses and compresses subtries chosen by means of a novel space-optimisation procedure hinging on a pool of new encoding techniques. The experimental results over a variety of six datasets and five highly-engineered competitors suggest that our CoCo-trie does advance the state of the art of string dictionaries.

Our novel design scheme paves the way for the *multi-criteria* optimisation of trie data structures that take into account possibly other encoding schemes (for macro-nodes) and multi-objective functions (e.g., over time, space, query distribution, etc.). It could also be interesting to investigate the use of hash-based approaches [53] in our space-optimisation scheme because this could allow faster `lookups` but at the cost of giving up the `rank` operation. Finally, we mention some issues about dynamic string dictionaries that would deserve further investigation: the CoCo-trie could be "dynamised" either by (i) adopting an LSM-based approach, which is very well-known in large write-intensive systems (such as in key–value stores [13]); or by (ii) the use of buffers or dynamic containers in macro-nodes to hold newly inserted strings.

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 paper.

Data availability

The source code and the datasets are available at https://github.com/aboffa/CoCo-trie.

Acknowledgements

This work was supported by the European Union – Horizon 2020 Programme 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] B.-J.P. Hsu, G. Ottaviano, Space-efficient data structures for top-k completion, in: Proc. 22nd International Conference on World Wide Web, WWW, 2013, pp. 583–594, http://dx.doi.org/10.1145/2488388.2488440.

[2] S. Gog, G.E. Pibiri, R. Venturini, Efficient and effective query auto-completion, in: Proc. 43rd ACM International Conference on Research and Development in Information Retrieval, SIGIR, 2020, pp. 2271–2280, http://dx.doi.org/10.1145/3397271.3401432.

[3] Y.M. Kang, W. Liu, Y. Zhou, QueryBlazer: Efficient query autocompletion framework, in: Proc. 14th International Conference on Web Search and Data Mining, WSDM, 2021, pp. 1020–1028, http://dx.doi.org/10.1145/3437963.3441725.

[4] R. Mavlyutov, M. Wylot, P. Cudré-Mauroux, A comparison of data structures to manage URIs on the web of data, in: Proc. 12th European Semantic Web Conference, ESWC, 2015, pp. 137–151, http://dx.doi.org/10.1007/978-3-319-18818-8_9.

[5] H. Zhang, H. Lim, V. Leis, D.G. Andersen, M. Kaminsky, K. Keeton, A. Pavlo, SuRF: practical range query filtering with fast succinct tries, in: Proc. ACM International Conference on Management of Data, SIGMOD, 2018, pp. 323–336, http://dx.doi.org/10.1145/3183713.3196931.

[6] S. Luo, S. Chatterjee, R. Ketsetsidis, N. Dayan, W. Qin, S. Idreos, Rosetta: a robust space-time optimized range filter for key-value stores, in: Proc. ACM International Conference on Management of Data, SIGMOD, 2020, pp. 2071–2086, http://dx.doi.org/10.1145/3318464.3389731.

[7] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, Bigtable: a distributed storage system for structured data, ACM Trans. Comput. Syst. 26 (2) (2008) http://dx.doi.org/10.1145/1365815.1365816.

[8] A. Apostolico, M. Crochemore, M. Farach-Colton, Z. Galil, S. Muthukrishnan, 40 Years of suffix trees, Commun. ACM 59 (4) (2016) 66–73, http://dx.doi.org/10.1145/2810036.

[9] V. Mäkinen, D. Belazzougui, F. Cunial, A.I. Tomescu, Genome-Scale Algorithm Design, Cambridge University Press, 2015, http://dx.doi.org/10.1017/CBO9781139940023.

[10] S.J. Huston, A. Moffat, W.B. Croft, Efficient indexing of repeated n-grams, in: Proc. 4th International Conference on Web Search and Web Data Mining, WSDM, 2011, pp. 127–136, http://dx.doi.org/10.1145/1935826.1935857.

[11] G.E. Pibiri, R. Venturini, Efficient data structures for massive N-gram datasets, in: Proc. 40th ACM International Conference on Research and Development in Information Retrieval, SIGIR, 2017, pp. 615–624, http://dx.doi.org/10.1145/3077136.3080798.

[12] 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.

[13] C. Luo, M.J. Carey, LSM-based storage techniques: a survey, VLDB J. 29 (1) (2020/01/01) 393–418, http://dx.doi.org/10.1007/s00778-019-00555-y.

[14] D.E. Knuth, The Art of Computer Programming, Vol. 3, second ed., Addison-Wesley, 1998.

[15] 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.

[16] S. Kanda, D. Köppl, Y. Tabei, K. Morita, M. Fuketa, Dynamic path-decomposed tries, ACM J. Exp. Algorithmics 25 (2020) 1–28, http://dx.doi.org/10.1145/3418033.

[17] R. Grossi, G. Ottaviano, Fast compressed tries through path decompositions, ACM J. Exp. Algorithmics 19 (1) (2014) http://dx.doi.org/10.1145/2656332, URL https://github.com/ot/path_decomposed_tries.

[18] S. Nilsson, M. Tikkanen, Implementing a dynamic compressed trie, in: Proc. 2nd International Workshop on Algorithm Engineering, WAE, 1998, pp. 25–36.

[19] 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.

[20] N. Askitis, R. Sinha, Engineering scalable, cache and space efficient tries for strings, VLDB J. 19 (5) (2010) 633–660, http://dx.doi.org/10.1007/s00778-010-0183-9.

[21] S. Heinz, J. Zobel, H.E. Williams, Burst tries: a fast, efficient data structure for string keys, ACM Trans. Inf. Syst. 20 (2) (2002) 192–223, http://dx.doi.org/10.1145/506309.506312.

[22] T. Takagi, S. Inenaga, K. Sadakane, H. Arimura, Packed compact tries: a fast and efficient data structure for online string processing, IEICE Trans. Fundam. Electron. Commun. Comput. Sci. 100-A (9) (2017) 1785–1793, http://dx.doi.org/10.1587/transfun.E100.A.1785.

[23] P. Bille, I.L. Gørtz, F.R. Skjoldjensen, Deterministic indexing for packed strings, in: Proc. 28th Annual Symposium on Combinatorial Pattern Matching, CPM, 78, 2017, pp. 6:1–6:11, http://dx.doi.org/10.4230/LIPIcs.CPM.2017.6.

[24] 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.

[25] J.J. Darragh, J.G. Cleary, I.H. Witten, Bonsai: a compact representation of trees, Softw. - Pract. Exp. 23 (3) (1993) 277–291, http://dx.doi.org/10.1002/spe.4380230305.

[26] J.L. Bentley, R. Sedgewick, Fast algorithms for sorting and searching strings, in: Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, 1997, pp. 360–369, http://dx.doi.org/10.5555/314161.314321.

[27] Y. Mao, E. Kohler, R.T. Morris, Cache craftiness for fast multicore key-value storage, in: Proc. 7th European Conference on Computer Systems, EuroSys, 2012, pp. 183–196, http://dx.doi.org/10.1145/2168836.2168855.

[28] S. Yata, Dictionary compression by nesting prefix/patricia tries, in: Proc. 17th Meeting of the Association for Natural Language, 2011.

[29] 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.

[30] A. Poyias, S.J. Puglisi, R. Raman, m-Bonsai: a practical compact dynamic trie, Internat. J. Found Comput. Sci. 29 (8) (2018) 1257–1278, http://dx.doi.org/10.1142/S0129054118430025.

[31] P. Ferragina, F. Luccio, G. Manzini, S. Muthukrishnan, Compressing and indexing labeled trees, with applications, J. ACM 57 (1) (2009) 4:1–4:33, http://dx.doi.org/10.1145/1613676.1613680.

[32] P. Ferragina, R. Venturini, The compressed permuterm index, ACM Trans. Algorithms 7 (1) (2010) 10:1–10:21, http://dx.doi.org/10.1145/1868237.1868248.

[33] 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.

[34] M.A. Bender, M. Farach-Colton, B.C. Kuszmaul, Cache-oblivious string B-trees, in: Proc. 25th ACM Symposium on Principles of Database Systems, PODS, 2006, pp. 233–242, http://dx.doi.org/10.1145/1142351.1142385.

[35] 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.

[36] 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.

[37] 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.

[38] J. Arz, J. Fischer, Lempel-Ziv-78 compressed string dictionaries, Algorithmica 80 (7) (2018) 2012–2047, http://dx.doi.org/10.1007/s00453-017-0348-7.

[39] R. Binna, E. Zangerle, M. Pichl, G. Specht, V. Leis, HOT: a height optimized trie index for main-memory database systems, in: Proc. ACM International Conference on Management of Data, SIGMOD, 2018, pp. 521–534, http://dx.doi.org/10.1145/3183713.3196896.

[40] 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.

[41] 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.

[42] A. Boffa, P. Ferragina, F. Tosoni, G. Vinciguerra, Compressed string dictionaries via data-aware subtrie compaction, in: Proc. 29th International Symposium on String Processing and Information Retrieval, SPIRE, 2022, pp. 233–249, http://dx.doi.org/10.1007/978-3-031-20643-6_17.

[43] R. De La Briandais, File searching using variable length keys, in: Proc. Western Joint Computer Conference, 1959, pp. 295–298, http://dx.doi.org/10.1145/1457838.1457895.

[44] S. Sahni, Tries, in: D.P. Mehta, S. Sahni (Eds.), Handbook of Data Structures and Applications, Chapman and Hall/CRC, 2004, http://dx.doi.org/10.1201/9781420035179.pt5.

[45] P. Boldi, A. Marino, M. Santini, S. Vigna, BUbiNG: massive crawling for the masses, ACM Trans. Web 12 (2) (2018) http://dx.doi.org/10.1145/3160017.

[46] Pizza&Chili corpus, 2023, URL http://pizzachili.dcc.uchile.cl/texts.html. (Accessed June 2022).

[47] R.O. Nambiar, M. Poess, The making of TPC-DS, in: Proc. 32nd International Conference on Very Large Data Bases, VLDB, 2006, pp. 1049–1058, URL http://www.tpc.org/tpcds/.

[48] D. Belazzougui, P. Boldi, R. Pagh, S. Vigna, Theory and practice of monotone minimal perfect hashing, ACM J. Exp. Algorithmics 16 (2008) http://dx.doi.org/10.1145/1963190.2025378.

[49] 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.

[50] 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.

[51] J.-I. Aoe, K. Morimoto, T. Sato, An efficient implementation of trie structures, Softw. - Pract. Exp. 22 (9) (1992) 695–721, http://dx.doi.org/10.1002/spe.4380220902.

[52] S. Kanda, K. Morita, M. Fuketa, Compressed double-array tries for string dictionaries supporting fast lookup, Knowl. Inf. Syst. 51 (3) (2017) 1023–1042, http://dx.doi.org/10.1007/s10115-016-0999-8.

[53] N. Askitis, J. Zobel, Cache-conscious collision resolution in string hash tables, in: Proc. 12th International Conference on String Processing and Information Retrieval, SPIRE, 2005, pp. 91–102, http://dx.doi.org/10.1007/11575832_11.

[54] D. Belazzougui, P. Boldi, S. Vigna, Dynamic Z-fast tries, in: Proc. 17th International Symposium on String Processing and Information Retrieval, SPIRE, 2010, pp. 159–172, http://dx.doi.org/10.1007/978-3-642-16321-0_15.

[55] D.D. Sleator, R.E. Tarjan, A data structure for dynamic trees, J. Comput. System Sci. 26 (3) (1983) 362–391, http://dx.doi.org/10.1016/0022-0000(83)90006-5.

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

[57] 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.

[58] M. Burrows, D.J. Wheeler, A Block-Sorting Lossless Data Compression Algorithm, Tech. Rep. 124, Digital Equipment Corporation, 1994.

[59] C.D. Manning, P. Raghavan, H. Schütze, Introduction to Information Retrieval, Cambridge University Press, 2008, http://dx.doi.org/10.1017/CBO9780511809071.

[60] G. Navarro, Indexing highly repetitive string collections, part II: compressed indexes, ACM Comput. Surv. 54 (2) (2020) http://dx.doi.org/10.1145/3432999.

[61] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Comput. Surv. 39 (1) (2007).

[62] D. Belazzougui, P. Boldi, R. Pagh, S. Vigna, Monotone minimal perfect hashing: searching a sorted table with $O(1)$ accesses, in: Proc. 20th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, 2009, pp. 785–794, http://dx.doi.org/10.5555/1496770.1496856.

[63] P. Elias, Efficient storage and retrieval by content and address of static files, J. ACM 21 (2) (1974) 246–260, http://dx.doi.org/10.1145/321812.321820.

[64] R.M. Fano, On the Number of Bits Required To Implement an Associative Memory. Memo 61, Massachusetts Institute of Technology, Project MAC, 1971.

[65] F. Silvestri, R. Venturini, VSEncoding: efficient coding and fast decoding of integer lists via dynamic programming, in: Proc. 19th ACM International Conference on Information and Knowledge Management, CIKM, 2010, pp. 1219–1228, http://dx.doi.org/10.1145/1871437.1871592.

[66] G. Ottaviano, R. Venturini, Partitioned Elias-Fano indexes, in: Proc. 37th ACM International Conference on Research and Development in Information Retrieval, SIGIR, 2014, pp. 273–282, http://dx.doi.org/10.1145/2600428.2609615.

[67] J. Kärkkäinen, D. Kempa, S.J. Puglisi, Hybrid compression of bitvectors for the FM-index, in: Proc. 24th Data Compression Conference, DCC, 2014, pp. 302–311, http://dx.doi.org/10.1109/DCC.2014.87.

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

[69] 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.

[70] F. Claude, G. Navarro, Practical rank/select queries over arbitrary sequences, in: Proc. 15th International Symposium on String Processing and Information Retrieval, SPIRE, 2008, pp. 176–187, http://dx.doi.org/10.1007/978-3-540-89097-3_18.

[71] P. Pandey, M.A. Bender, R. Johnson, R. Patro, A general-purpose counting filter: making every bit count, in: Proc. ACM International Conference on Management of Data, SIGMOD, 2017, pp. 775–787, http://dx.doi.org/10.1145/3035918.3035963.

[72] 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.

[73] G. Ottaviano, N. Tonellotto, R. Venturini, Optimal space-time tradeoffs for inverted indexes, in: Proc. 8th ACM International Conference on Web Search and Data Mining, WSDM, 2015, pp. 47–56, http://dx.doi.org/10.1145/2684822.2685297.

[74] 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.

[75] 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.

[76] M. Pöss, R.O. Nambiar, D. Walrath, Why you should run TPC-DS: a workload analysis, in: Proc. 33rd International Conference on Very Large Data Bases, VLDB, 2007, pp. 1138–1149.

[77] M. Boissier, Robust and budget-constrained encoding configurations for in-memory database systems, PVLDB 15 (4) (2022) 780–793, http://dx.doi.org/10.14778/3503585.3503588.

[78] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, T. Neumann, Query optimization through the looking glass, and what we found running the join order benchmark, VLDB J. 27 (5) (2018) 643–668, http://dx.doi.org/10.1007/s00778-017-0480-7.

[79] H. Garcia-Molina, J.D. Ullman, J. Widom, Database Systems: The Complete Book, second ed., Prentice Hall Press, 2008.

[80] ART and CART implementations, 2023, URL https://github.com/efficient/fast-succinct-trie/tree/master/third-party/art. (Accessed June 2022).

[81] H.E. Williams, J. Zobel, Compressing integers for fast file access, Comput. J. 42 (3) (1999) 193–201, http://dx.doi.org/10.1093/comjnl/42.3.193.

[82] FST implementation, 2023, URL https://github.com/kampersanda/fast_succinct_trie. (Accessed June 2022).