C^2: Cache-Conscious Succinct Tries with Adaptive Unary Path Compression

Kepan Zhang
Georgia Institute of Technology
USA
zhangkepan03@gmail.com

Tiancheng Zhao
Georgia Institute of Technology
USA
tzhao350@gatech.edu

Helen Xu
Georgia Institute of Technology
USA
hxu615@gatech.edu

Abstract

Succinct tries are powerful string dictionaries because of their low memory footprint and fast query performance. However, existing succinct trie implementations face two key challenges to spatial locality: 1) they incur unnecessary cache misses during queries, especially during trie navigation operations, and 2) they waste significant space when the data contains many unary paths. We propose C^2, a set of two techniques: C$_1$ introduces a more cache-friendly layout for the bitvector underlying succinct tries, and C$_2$ compresses redundant unary paths. We thoroughly redesign three state-of-the-art succinct tries: FST, CoCo-trie, and Marisa, producing C^2-FST, C^2-CoCo, and C^2-Marisa. Experiments on six diverse datasets show that the C$_1$ optimization improves query performance by 1.58×, 1.12×, and 1.42×, respectively, compared to the original FST, CoCo-trie, and Marisa. Furthermore, the C$_2$ optimization achieves a 1.3× smaller memory footprint on average. The succinct tries optimized with both aspects of C^2 achieve better space-time tradeoffs than their original versions and other state-of-the-art succinct tries, while using significantly less space than non-succinct tries like ART and C-ART.

Keywords

succinct, tries, cache, optimization, path, compression

1 Introduction

Succinct data structures have become essential in applications like large-scale databases, graph processing, and text indexing, especially in memory-constrained scenarios [36]. They are string-data representations that use close to the theoretical minimum number of bits while still supporting efficient queries [20, 30]. We focus on ***succinct tries*** [11, 12, 23, 42, 44], which match or exceed the performance of regular tries with orders of magnitude less memory.

Succinct tries separate the trie ***topology*** from the string ***data***, encoding the topology in a compact bitvector that uses only about 2 bits per node on average compared to at least 128 bits for two pointers in traditional tries [24]. Navigation requires auxiliary index operations (e.g., "`rank`" and "`select`") on top of this bitvector.

Challenges to locality. Existing bitvector-based succinct tries exhibit suboptimal locality in two respects: 1) the *bitvector layout* in the topology, and 2) the ***unary paths***, or maximal trie paths of at least 2 nodes in which every node except the last has exactly one child, in the data. For example, in Figure 3, paths 0-1-4 ("ca") and 0-2-5 ("su") are internal unary paths, whereas path 7-14-18 ("ie") is a suffix unary path. Sections 3 and 4 detail both challenges.

Table 1 confirms locality challenges due to navigation operations across state-of-the-art succinct tries. For example, a child navigation in Marisa [42] incurs at least 3 cache misses to access the topology bitvector compared to 1 in a pointer-based trie to follow the pointer.

Table 1: Average number of LLC misses per query on the two largest datasets. FST [44], CoCo-trie [12] and Marisa [42] are bitvector-based succinct tries, while C-ART [43] is a pointer-based compact trie. Tries prefixed with C^2- are our cache-optimized versions.

Dataset	*FST*	C^2-*FST*	*CoCo'*	C^2-*CoCo*	*Marisa*	C^2-*Marisa*	*C-ART*
`wiki` [41]	82	78	66	49	56	58	**23**
`log` [18]	235	230	57	**45**	58	62	117

Table 2: Unary-path statistics for the two largest datasets. #Branch denotes the number of branching edges. Given a path length ℓ, we report the percentage of paths with lengths $\ell = 1$, $1 < \ell \leq 3$, and $\ell > 3$. ℓ_{AVG} and ℓ_{MAX} denote the average and maximum lengths of compressible unary paths.

Dataset	*# Branch*	$\ell = 1$	$1 < \ell \leq 3$	$\ell > 3$	ℓ_{AVG}	ℓ_{MAX}
`wiki` [41]	467K	15.5%	0.4%	84.1%	271	8676
`log` [18]	7,284K	17.4%	3.8%	78.8%	65	8196

Additionally, practical datasets often include strings with long shared prefixes followed by diverse dangling suffixes, which naturally introduce large numbers of unary paths in trie-based representations. For example, collections of hyperlinks frequently share common prefixes such as "en.wikipedia.org/wiki/", while differing only in their trailing titles. As a result, large portions of the trie structure degenerate into long chains of unary nodes.

Without effective compression, unary paths can consume a dominant fraction of succinct-trie storage, increasing memory footprint and disrupting cache locality. For example, as shown in Table 2, the majority of suffix regions in realistic datasets consist of long, redundant unary chains in the two largest tested inputs. In the `wiki` dataset, 84.1% of all branch edges correspond to compressible unary paths, with an average compressible length of 271 characters and a maximum compressible length of over 8K characters. Similarly, the `log` dataset contains 78.8% compressible unary paths.

Optimizing for locality in bitvector-based succinct tries. *We introduce C^2, a set of techniques that improve locality in both the topology and the data of succinct tries.* We target two operations: the ***existence query*** (checking whether a key exists) and the ***range query*** (returning a range of keys).

The first "C" of C^2 improves cache locality of the `select` index in succinct tries. We introduce the ***functional index***, a novel `select` index layout that maps straightforwardly to bit positions,

EDBT '27, 6-9 April 2027, Lille (France)

10.48786/edbt.2027.08

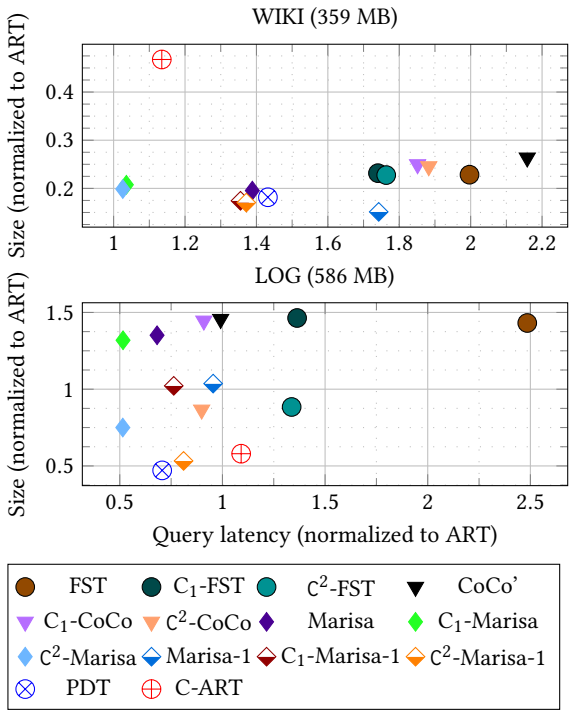

Figure 1: Space-time performance comparison on `wiki` [41] and `log` [18] (normalized to ART). We use FSST as the tail container in all C²-tries. The prefixes "C₁-"/"C²-" indicate different optimization strategies (bitvector redesign / bitvector redesign + unary path compression); the suffix "-1" means we enforce one recursion (see Section 5). Lower and to the left is better.

enabling both `rank` and `select` indexes to be inlined in the bit sequence. Using the functional index, we cut random accesses to two per child navigation in LOUDS-based succinct tries, which include many state-of-the-art tries such as the FST [44], CoCo-trie [11], and Marisa [42]. Table 1 and Section 5 show that this optimization reduces cache misses by 19% on average with at most 2% overall space overhead.

The second "C" of C² is an ***adaptive compression*** algorithm that gives *any* succinct trie access to compression in the "tail container," which stores the remaining string suffixes at the end of trie paths. Building on "re-pair" [27] and "recursion" [42], two orthogonal techniques for path compression, we enable all tries to use both recursion and the Fast Static Symbol Table (FSST) [14], a lightweight scheme that achieves good compression ratios with fast build times. Although recursion and compressed tail containers appear in prior work, this is the first generalization of recursion across tail containers, exposing space-time tradeoffs for any succinct trie.

Contributions. Our primary contributions are as follows:

- We redesign the succinct-trie topology with a *cache-conscious* functional index that inlines both `rank` and `select` indexes, minimizing cache misses during trie navigation.
- We propose an *adaptive compression scheme* that selects the best compression strategy per dataset to balance space and query performance. Our experiments indicate that the chosen strategy improves space efficiency by 1.3× on average with similar (within 1.05×) query performance compared to the original succinct trie versions. Furthermore, increasing compression aggressiveness exposes further space-time tradeoffs. For example, on Marisa, increasing "recursion" for more compression yields 1.17× space savings, but 1.30× slower queries.
- We integrate these techniques into three state-of-the-art succinct tries — FST [44], CoCo-trie [11, 12], and Marisa [42] — improving query performance on the whole (with both C_1 and C_2 optimizations) by 1.58×, 1.13×, and 1.42×, respectively.

Results summary. Overall, C²-optimized tries improve cache locality and performance over their original counterparts with negligible (<4%) additional space overhead. C²-FST achieves better space-time performance than FST on all datasets. C²-CoCo outperforms CoCo' by 1.13× in query latency and is 1.31× smaller in space, on average across the six datasets [1]. C²-Marisa improves Marisa's query performance by 1.42× with similar memory consumption, and is generally the strongest C² index.

Figure 1 shows the query latency and size of the different tries on the `wiki` and `log` datasets. C²-Marisa dominates all other succinct tries on query latency with the lowest (or within a few percent of the lowest) space usage. ART and C-ART, two non-succinct tries, achieve lower query latency at the cost of higher space usage.

These space-time improvements stem from the cache-locality gains of C², as demonstrated by the cache-miss counts in Table 1.

2 Preliminaries

We introduce the `rank`/`select` primitives on bitvectors and common succinct tree-encoding schemes, review four state-of-the-art succinct tries this paper builds upon, and summarize string compression schemes for the tail container.

2.1 Rank/Select Primitives

Bitvectors are a fundamental building block of many succinct data structures. Succinct tree navigation relies on the `rank` and `select` primitives of bitvectors, which are defined as follows:

- $\texttt{rank}_p\texttt{(i)}$: return the number of pattern p's in the first i bits of the bitvector.
- $\texttt{select}_p\texttt{(i)}$: return the end of the i-th occurrence of pattern p in the bitvector.

Different bitvectors support different patterns for `rank` and `select`. Common choices of p include 1, 0 and 00. For example, $\texttt{rank}_1\texttt{(i)}$ returns the number of occurrences of "1" in positions [0, i).

Index structures for `rank` and `select` partition the bitvector into fixed-sized (e.g., 512-bit) *basic blocks* and store partial results so far for each block from left-to-right [45]. They also often include *secondary indices* on smaller fixed-size blocks within each basic block for higher granularity and better performance. These auxiliary structures reduce the worst-case cost of `rank` and `select` to $O(1)$ with minimal (typically about 5%) space overhead [15, 20–22, 31, 34, 40, 44, 45].

For the `rank` operation, the ***rank index*** stores ***samples***, or exact accumulative ranks before each block (e.g., the number of 1s preceding each basic block), which enable fast exact responses to `rank` queries. Answering a $\texttt{rank}_p\texttt{(i)}$ query involves combining the accumulative rank before the target block with the (computed) rank in the remainder up to position `i`.

[1] The original CoCo-trie was designed for prefix-only datasets and fails to build on the original datasets. We implemented CoCo' by integrating CoCo-trie's topology with C²-CoCo. Section 5 provides the full details and shows that CoCo and CoCo' achieve almost identical performance and space usage on the prefix-only datasets.

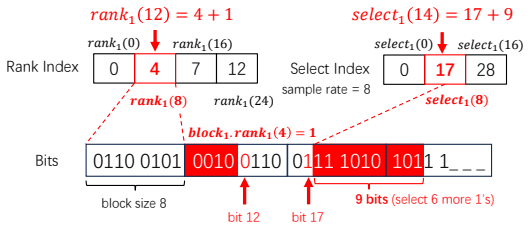

Figure 2: Examples of rank/select operations using the associated indices on top of a bitvector.

For example, Figure 2 illustrates a worked example of a $\texttt{rank}_1\texttt{(12)}$ query on a bitvector where the rank index is built with block size 8. First, we look up the result of $\texttt{rank}_1\texttt{(8) = 4}$ in $O(1)$ time in the rank index because position 8 is the start of the basic block containing 12. Second, we calculate the remaining instances of the pattern in the range 8 - 12 in the bitvector. The combined result of the first and second step is 4 + 1 = 5, which is $\texttt{rank}_1\texttt{(12)}$.

The `select` operation uses a ***select index*** similar to the aforementioned rank index, but `select` is more challenging than `rank` because a query may not know exactly which block is needed based on the input. For example, with 8-bit basic blocks, bit 12 is always in the second basic block, but the 12-th *one bit* might be in the 100-th basic block!

To support the `select` operation, the `select` index precomputes and stores samples, or exact results of `select` queries at regular intervals (e.g., $\texttt{select}_p\texttt{(0)}$, $\texttt{select}_p\texttt{(8)}$, $\texttt{select}_p\texttt{(16)}$, ...). Answering a `select` query involves finding the closest index entry to the left of the desired select query, then scanning sequentially using that as a starting point. For example, Figure 2 illustrates a worked example of a $\texttt{select}_1\texttt{(14)}$ query on a bitvector where the select index is built with block size 8. First, we look up the closest entry below, which is $\texttt{select}_1\texttt{(8) = 17}$, in $O(1)$ time. Therefore, we know that the result of $\texttt{select}_1\texttt{(14)}$ must be to the right of $\texttt{select}_1\texttt{(8)}$, so we scan left to right starting from position 17 in the bitvector until we have found 6 more bits (for 8 + 6 = 14 bits total).

Succinct trie bitvector structure. BP, LOUDS, and DFUDS are three popular bitvector-based succinct tree-encoding schemes [10, 24]. They all use the topology bitvector to map each tree element (i.e., nodes, edges, internal nodes, leaf nodes) to a unique integer ID in the range of $[0, n-1]$, where n is the number of such elements in the tree. Depending on the encoding scheme, the topology may support efficient tree operations (e.g., parent/child navigation) via bitvector operations (e.g., rank/select). Succinct trees encoded using these schemes can be converted to tries by storing data associated with each trie edge in a separate array, which is indexed using the edge IDs [12, 23, 42, 44].

Efficient bitvector operations require auxiliary data structures which separate the bitvector into two parts: the ***index***, which accelerates bitvector operations, and the ***bit sequence***, which stores the content of the bitvector (i.e. the trie topology). The index facilitates `rank`/`select` operations essential to succinct trie navigation.

2.2 LOUDS Encoding

Level-Order Unary Degree Sequence (LOUDS) is aclassical example of a bitvector-based succinct-tree encoding scheme [24]. Each tree node is encoded using the bit sequence 1^m0 (i.e., m 1's followed by a 0), where m is the degree (i.e., the number of children) of the node. The encoding bits of each node are concatenated in level order to form the encoding bitvector `bv` of the entire tree. With this encoding, each 1 bit corresponds to a parent-to-child edge, whereas each 0 bit marks the end of a node. We say a position i belongs to a node k if bit i is part of node k's encoding. The LOUDS encoding efficiently supports the following tree operations:

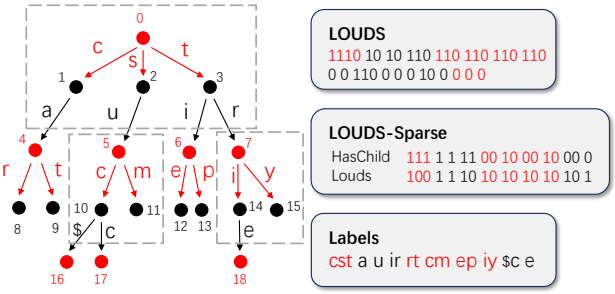

Figure 3: LOUDS/LOUDS-Sparse encoding example. Stored keys: car, cat, suc, succ, sum, tie, tip, trie, try.

- `NodeID(i)` = $\texttt{bv.rank}_0\texttt{(i)}$, the LOUDS ID of the node to which position i belongs.
- `EdgeID(i)` = $\texttt{bv.rank}_1\texttt{(i)}$, the LOUDS ID of the edge at position i (Requirement: `bv[i] == 1`).
- `LeafID(i)` = $\texttt{bv.rank}_{00}\texttt{(i)}$, the LOUDS ID of the leaf node to which position i belongs.
- `InnerID(i)` = `NodeID(i) - LeafID(i)`, the LOUDS ID of the inner node to which position i belongs.
- `Child(i)` = $\texttt{bv.select}_0\texttt{(bv.rank}_1\texttt{(i+1)) + 1}$, the position of the child node connected to the edge at position i.
- `Parent(i)` = $\texttt{bv.select}_1\texttt{(bv.rank}_0\texttt{(i))}$, the position of the parent of the node to which position i belongs.

As illustrated in Figure 3, encoding a trie using LOUDS involves the following steps: First, the *topology* of the trie is encoded using the LOUDS bitvector; Second, all the trie labels are concatenated in level order into a separate array. At query time, trie labels are uniquely indexed by edge IDs (because they are associated with trie edges), whereas keys are indexed by leaf IDs.

Next, we will present an example of how to traverse a LOUDS-encoded trie using the trie in Figure 3. Suppose we are navigating to the second child of the root (i.e., from node 0 to node 2). We need to follow the second edge of the root node, which is at position 1. From the aforementioned formula, $\texttt{Child(1)} = \texttt{bv.select}_0\texttt{(bv.rank}_1\texttt{(2))+1} = \texttt{bv.select}_0\texttt{(2)+1} = 6$. Indeed, if we look at the LOUDS bitvector in Figure 3, the second child of the root starts at position 6.

Balanced Parentheses (BP) [24] and Depth-First Unary Degree Sequence (DFUDS) [10] are two popular alternatives to LOUDS with the same two-bits-per-node overhead, but they traverse the tree in different orders. Compared with LOUDS, they support a broader range of tree operations, including computing subtree sizes or counting the number of leaf nodes to the left/right of a specific node. However, their functionality comes at the cost of slower parent/child navigation compared to LOUDS due to their greater complexity.

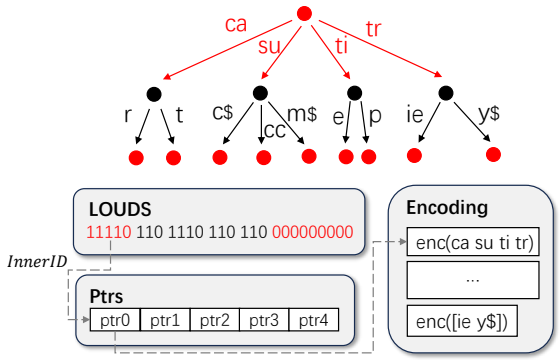

Figure 4: An example of a CoCo-trie obtained by collapsing sub-tries surrounded by dashed boxes in Figure 3.

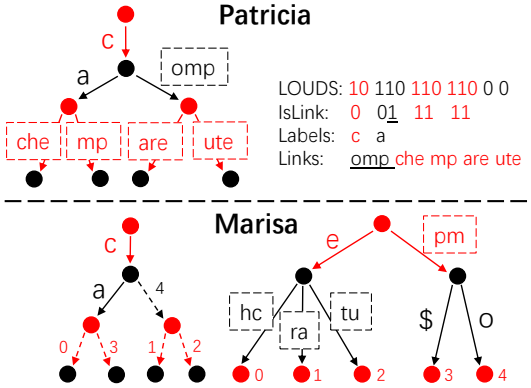

Figure 5: Examples of Patricia (top) and Marisa (bottom) with one recursion. Keys stored: cache, camp, compare, compute. Unary paths in the first Marisa trie are stored in the second Marisa trie in reversed form: ehc, era, etu, pm, pmo. Links in the first Marisa trie are indicated using dashed arrows, and the numbers alongside them indicate their values (which in this example are leaf node IDs in the second Marisa trie).

2.3 State-of-the-art Succinct Tries

Fast Succinct Trie. The Fast Succinct Trie (FST) [44] is a static succinct trie that combines the benefits of LOUDS-Sparse and LOUDS-Dense by leveraging both formats in different levels of the trie. Specifically, FST encodes the top levels of the trie, where nodes tend to have more branches and are more likely to be queried, using LOUDS-Dense, a representation that is fast and space efficient for nodes with dense branches. The rest of the trie, which tends to be colder and of lower average degree, is encoded with the more compact LOUDS-Sparse. Due to space limitations, we omit the details of LOUDS-Dense, but we refer the reader to the original FST paper [44] for the full details.

LOUDS-Sparse uses 2 bits to encode each trie edge. The `HasChild` bit of each edge indicates if this edge points to an internal node (1) or a leaf node (0). The `Louds` bit of each edge indicates if this edge points to the first child of current node (1) or not (0), which is used to determine node boundaries. These two sets of encoding bits are respectively concatenated in level order to form two bitvectors, `HasChild` and `Louds`. With LOUDS-Sparse, we have

- $\texttt{Child(i)} = \texttt{Louds.select}_1\texttt{(HasChild.rank}_1\texttt{(i+1)+1)}$.
- $\texttt{Parent(i)} = \texttt{HasChild.select}_1\texttt{(Louds.rank}_1\texttt{(i+1)-1)}$.
- $\texttt{LeafID(i)} = \texttt{HasChild.rank}_0\texttt{(i)}$.

Let us consider how to perform trie navigation in a LOUDS-sparse-encoded trie using the example in Figure 3. Suppose we are navigating again to the second child of the root (also at position 1). From above, $\texttt{Child(1)} = \texttt{Louds.select}_1\texttt{(HasChild.rank}_1\texttt{(2)+1)} = \texttt{Louds.select}_1\texttt{(3)} = 4$. Indeed, the second child of the root starts at position 4 in the `Louds` bitvector in the LOUDS-Sparse encoding.

CoCo-Trie. The CoCo-trie [11, 12] is a recent state-of-the-art static succinct trie that achieves both low memory consumption and fast query performance. A CoCo-trie is constructed by first creating a regular trie T and then adaptively selecting sub-tries of T to collapse into macro-nodes. Keys in each collapsed macro-node are transformed to an increasing sequence of integer codes which is then encoded using a select pool of succinct integer encoding algorithms. The optimal set of sub-tries to collapse is computed using a bottom-up dynamic-programming algorithm.

Figure 4 illustrates the CoCo-trie encoding of the example trie in Figure 3. Since the CoCo-trie uses LOUDS encoding, its trie navigation is done through `rank`/`select` operations on the bitvector as described in Section 2.2.

Marisa Trie. The Marisa trie [42] is a LOUDS-encoded Patricia trie [29] that contracts unary paths into single concatenated labels. Unlike Patricia, Marisa improves space efficiency by recursively storing unary paths in multiple tries.

The upper subgraph of Figure 5 shows an example of Patricia, where unary paths are contracted into a single node. To convert the trie to LOUDS succinct form, two bitvectors are needed: the topology bitvector `LOUDS` (Section 2.2), and an additional bitvector (`IsLink`) which indicates if each edge is a regular edge (0) or a contracted edge (1). Each contracted edge is uniquely indexed by a `LinkID` determined by a `rank` operation on the `IsLink` bitvector. The `Links` vector stores ***links***, or references, to the concatenated labels on the contracted edge. For example, bit 3 in `LOUDS` corresponds to bit 2 in `IsLink` and link 0 in `Links` ("omp").

Marisa improves on Patricia by recursively storing unary paths in secondary tries. The recursion continues until the number of tries reaches a preset limit, at which point the outstanding unary paths are sorted, deduplicated and moved to a simple "sorted" tail container (Section 2.4). The lower subgraph of Figure 5 shows a worked example of a Marisa trie with one recursion. For readability, links between tries are expressed as leaf node IDs in this example. Because the second trie only needs to support random access but not lookup operations, the keys are stored in reversed form so that they can be efficiently retrieved by reading the labels on a bottom-up path. For example, to retrieve link 4's corresponding key, we start from leaf 4 of the second trie, concatenate all the (reversed) labels on the leaf-to-root path ('o' and 'mp'), and obtain 'omp'. In practice, links are implemented as leaf node positions to avoid an additional `select` operation. Finally, to mitigate the cost of link tracing during lookups, Marisa uses a small piece of memory to cache frequently-taken paths.

The number of recursions in Marisa exposes a *space-time trade-off*, as more recursions further compress the data at the cost of degraded search performance and higher build times [4].

Path Decomposed Trie. The Path Decomposed Trie (PDT) [19, 23, 25] is a DFUDS-encoded succinct trie. A PDT is obtained by transforming a regular trie through path decomposition [38]: At each step, a root-to-leaf path is selected and contracted into

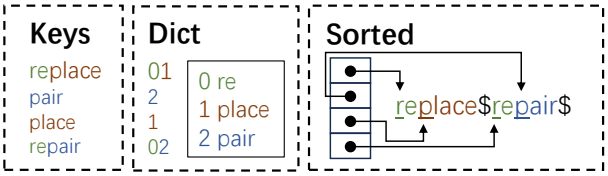

Figure 6: Examples of re-pair, FSST and sorted tail containers. Re-pair and FSST are both dictionary encoding methods (denoted with `Dict`) that replaces common text patterns with small integer codes. The sorted tail container (denoted with `Sorted`) overlaps suffix keys.

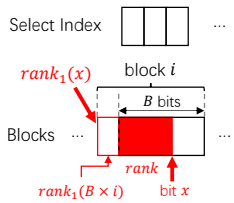

Figure 7: Interleaving.

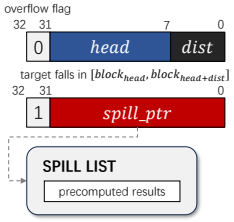

Figure 8: Select index overflow.

a single node, and the subtrees dangling on the original path are converted to its child nodes. The procedure recurses on the dangling subtrees.

The PDT achieves both significant memory reduction and competitive query performance by compressing the contracted paths with an approximate version of the ***re-pair*** text compression algorithm [16, 27], which we will summarize in the next subsection.

2.4 Text Compression Algorithms

Figure 6 presents a worked example of how to compress a set of strings with the sorted tail container, re-pair algorithm, and FSST. Next, we summarize these algorithms.

Sorted tail container. The sorted tail container is the string container for the last Marisa trie on a recursion chain. It overlaps two keys if one is a suffix of the other, and is particularly space efficient for datasets with short and repetitive keys. To efficiently detect possibilities of overlapping, it needs to reverse and sort all keys. Once the strings have been sorted, construction takes linear time. The link array which is used to retrieve keys from the container should also preserve the original key ordering.

Dictionary compression. Dictionary compression encodes common text patterns as small integer codes, exemplified by the famous LZW algorithm [26]. The mapping between patterns and integers is known as the ***dictionary***, and finding a good dictionary is known to be a key challenge to such algorithms [14]. In our work, we focus on two related algorithms that support fast random access: re-pair [27] and FSST [14].

Exact and approximate re-pair. At a high level, the original re-pair algorithm [27] replaces the most frequently-occurring character pair with a new special character, proceeding in rounds. For example, using special characters α, β, γ and δ, re-pair may compress the string aabbaabb as αbbαbb (α = aa) in the first pass, $\alpha\beta\alpha\beta$ (β = bb) in the second pass, $\gamma\gamma$ ($\gamma = \alpha\beta$) in the third pass and δ ($\delta = \gamma\gamma$) in the fourth pass. The dictionary is expressed as a set of recursive rules of pairing, which can be flattened to support constant-time decoding [23], with a small space overhead. Prior work shows that re-pair compresses a sequence T of length n over an alphabet of size σ and k-th order entropy H_k to $O(H_k)$ bits. [32, 33].

In practice, the PDT uses an *approximate* version of the re-pair algorithm [16] that accelerates build time by orders of magnitude compared to exact re-pair by identifying and replacing the top k most frequently-seen character pairs in each round, rather than one at a time. To our knowledge, there is no theoretical bound on how far the compression ratio of approximate re-pair is from exact re-pair's, as it depends on the dataset.

Fast Static Symbol Table. The Fast Static Symbol Table (FSST) [14] is similar to approximate re-pair, but optimizes for *lightweight* compression (with linear build times). Specifically, it restricts the size of the symbol table (i.e., the list of special characters for replacement) to 256 entries and expands the size of each replaced entry to substrings of 8 bytes (rather than two characters in the original approximate re-pair). FSST further optimizes for build time by first encoding a small sample (about 16KB) of the original dataset, and then using the resulting symbol table to encode the entire dataset. FSST implements hardware-specific optimizations, achieving extremely high compression and decompression throughput. In practice, FSST reduces the compression time by over an order of magnitude compared to approximate re-pair, while achieving similar compression ratios. However, no theoretical space bound has been proved for FSST.

3 The First "C": Cache-conscious Bitvector Design

In this section, we present the first "C" of C^2: Cache-conscious bitvector design (denoted with C_1). First, inspired by prior work [15, 21], we apply an array-of-structs reorganization to bitvectors to better exploit the access pattern of `rank` queries. Second, we eliminate cache misses from reading the `select` index using our novel *functional index*. Third, to handle both sparse and dense patterns, we further optimize the existing technique of overflowing `select` indexes and reduce cache misses by storing all data in-place. Finally, we apply the above techniques to three state-of-the-art LOUDS-based succinct tries [11, 42, 44].

We adopt two design principles: (1) optimization priority of cache > branch > arithmetic operations [45], and (2) trading space for performance. Since trie topologies are usually much smaller than trie labels, a small space overhead is worthwhile for increased query speed.

Locality issues in bitvectors. The main challenge to locality during trie navigation arises from storing the index and bitvector separately [20, 22, 31, 34, 40, 44, 45], since both must be accessed during `rank` and `select`. Packing the `rank` index with the bit sequence improves locality for `rank` [15, 21], but the same approach does not work for `select`. In most cases, the input to `rank` maps linearly to a position in the bit sequence; the input to `select`, however, is often an intermediate value (e.g. the output of `rank`) with no straightforward mapping to a bit position.

3.1 Array-of-Struct Reordering

Most traditional bitvectors are organized in a struct-of-array manner. During `rank` queries, each query accesses the target block immediately after its `rank` index. To better exploit this

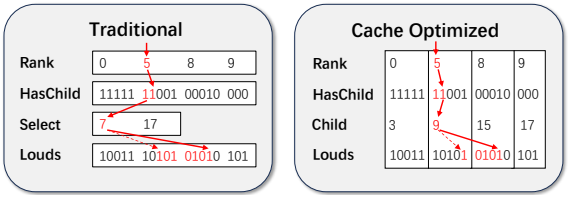

Figure 9: Functional index example for LOUDS-Sparse's `Child(x) = Louds.select`$_1$`(HasChild.rank`$_1$`(x + 1) + 1)`. The block size and the sampling rate for `select` (left) are both 5. The third row on the right (`Child`) caches the results of `Child(x)` at the start of block (i.e. `Child(0)`, `Child(5)`, `etc.`).

locality, `rank` indexes can be ***interleaved***, or stored in an array-of-structs manner with the index and bitvector in one contiguous memory allocation, as illustrated in Figure 7.

Similarly, for tries encoded with multiple bitvectors, we interleave blocks of different bitvectors that are often jointly accessed at nearby bit positions. For example, in LOUDS-Sparse, we pack the blocks of `HasChild` and `LOUDS`, as they are aligned with each other and are typically accessed together. Similarly, we can interleave secondary `rank` indexes [15, 20, 21, 40, 42].

We restrict the size of each `rank` index element to 32 bits for better space efficiency without loss of applicability to larger datasets. To support bitvectors larger than 2^{32} bits, previous work shows how to first *partition* the data into connected sub-tries that individually fit in the size limit [7].

3.2 Functional Index

To improve spatial locality for `select`, we introduce the ***functional index*** that samples results of the navigation function (e.g., `Child(x)`) rather than intermediate `select` values. For example, to support child navigation in LOUDS (where `Child(x) =` `select`$_0$`(rank`$_1$`(x+1))+1`), we sample `Child(x)`, instead of `select`$_0$`(x)`, at regular intervals of x. Since the sampled function is monotonically non-decreasing, the sampling is unambiguous. We then interleave these samples with blocks for spatial locality. Other succinct trie operations (e.g., parent navigation) can be optimized using similar approaches.

Figure 9 shows a worked example of computing `Child(6)` using traditional and functional indexes. An implementation using the traditional index first computes `HasChild.rank`$_1$`(6+1)=7` (by accessing `HasChild`'s rank index and the `HasChild` bitvector) and then obtains `Louds.select`$_1$`(7+1)=13` (by accessing `Louds`' select index and the `Louds` bitvector itself), which easily incurs 4 cache misses. In contrast, using the cache-optimized functional index, we first access block 1 (which bit 6 falls into) and obtain `HasChild.rank`$_1$`(6+1)` = 7 (using block 1's rank index). Then, we read block 1's functional index for `Child`, which stores the value of `Louds.select`$_1$`(5+1)` = 9. Since we need to compute `select`$_1$`(7+1)`, we select two more ones starting from bit 9 and end up at bit position 13.

Assuming the two accessed blocks belong to different cache lines, we incur only two cache misses in total. In practice, since trie navigation operations are often executed in a pipelined manner (*i.e.*, the output of one operation is immediately used as the input of the next operation), this optimization essentially cuts cache misses by 4× in long pipelines.

Applicability beyond LOUDS. Although we develop and evaluate the functional index in the context of LOUDS-based tries, the core idea applies to any succinct-tree encoding whose navigation functions are monotonically non-decreasing compositions of `rank` and `select`. The key insight behind the functional index is that `Child(x)` is a function of the position `x`, not of an intermediate bitvector value, so sampling it at regular intervals of `x` aligns the index with the bit sequence and enables interleaving. This property holds in BP and DFUDS as well, since the navigation function's input and output both increase monotonically with the traversal order of the encoding.

In BP, child navigation reduces to a `findclose` operation (i.e., locating the matching close parens for an open parens),which is implemented via `rank`/`select` on the parenthesis bitvector. The functional index directly applies because `findclose(x)` is monotonically non-decreasing in `x`, so it can be directly sampled at regular intervals of `x` and interleaved with the bit blocks. Similarly, DFUDS navigation starts with the `Child(x, 1)` operation, which returns the first child of the node at position `x`. This first-child operation admits a functional index in the standard way, with subsequent siblings located by scanning forward.

3.3 Overflow of Select Index

So far, we have assumed that the probe distance within the target interval is small. But this is not always the case, especially for the CoCo-trie, which sometimes contains nodes with very large fan-outs. For instance, when encoding the `trec-terms` [8] dataset in the original CoCo-trie paper, the first 600k bits of the CoCo-trie topology contain only 36 zero bits. With such extreme sparsity, regular searching algorithms would be too slow.

A classic solution to this problem is to allow the `select` index to overflow [34, 40]. That is, the `select` index should detect regions with extremely low pattern density, and precompute every `select` result in those regions. Since the precomputed results do not fit in regular sample space, they are overflowed to a separate list, and the sample stores a pointer to the overflow list instead.

Unfortunately, overflowing the select index disrupts locality because two adjacent samples are no longer guaranteed to form a bounding interval, as either of them can be overflowed. Therefore, after querying the sample, we must either use linear search or incur an additional cache miss to read the spill list.

We resolve this issue by storing *block indexes* instead of exact bit positions in regular samples, as this frees up additional bits to store the size of the bounding interval (in blocks). The `rank` indexes in each block provide enough information to restore any lost precision at no cost of additional cache misses. For example, in Figure 9, block 1's functional index for `Child` (i.e. `Child(5)`) points to block 1, which is equivalent to bit position 5 (instead of its original value 9 in Figure 9). Using the inlined rank index, we immediately restore `HasChild.rank`$_1$`(5) = 5`. Since `Child(5)` `= Louds.select`$_1$`(7)`, we can reach position `Child(5)` (i.e., 9) by selecting two more ones.

As shown in Figure 8, each sample occupies 32 bits, where bit 31 indicates whether the sample overflows. If bit 31 is false, then bits 7 to 30 (the 24-bit `head`, which means block size must be at least 256) point to the lower bounding block, whereas bits 0 to 6 (the 7-bit `dist`) store the length of the bounding interval (in blocks). If `dist` does not fit in 7 bits (*i.e.*, the bounding interval is at least 128 blocks long), then bit 31 is set to true, and the sample is overflowed. The rest of the sample (the 31-bit `spill_ptr`)

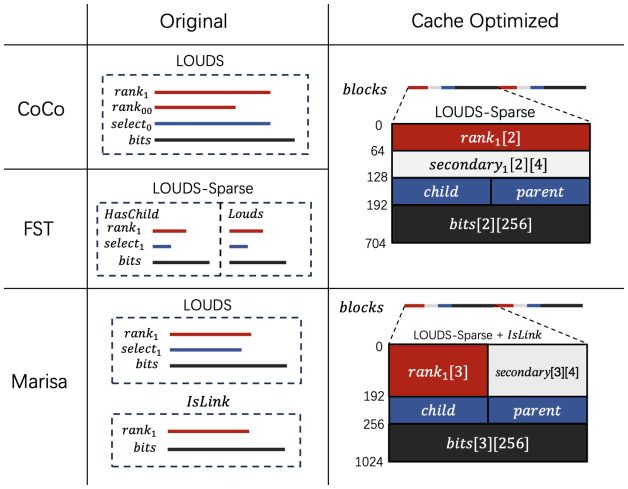

Figure 10: Optimized bitvector layout. `child`: functional index for `Child(x)`. `parent`: functional index for `Parent(x)`. $\texttt{secondary}_1$: secondary index for $\texttt{rank}_1$.

indexes into the centralized overflow list `spill_list`, where `spill_list[spill_ptr + i]` is the position of the i-th target pattern in the overflowing interval. We apply this optimization to both traditional `select` indexes and functional indexes with intermediate selects.

We sketch why `spill_ptr` always fits in 31 bits. With block size $B = 256$ and sample rate $S = 256$, each overflowing interval has target pattern density at most $S/128B = 1/128$ (for `select` indexes) or $B/128B = 1/128$ (for functional indexes). Since the bitvector size does not exceed 2^{32}, the size of `spill_list` would not exceed $2^{32}/128 = 2^{25}$ and hence always fits in 26 bits.

Finally, we address the space overhead incurred by the spill list. Because each spilled interval contains at most $B = 256$ bits, the density of ones is at least $256/(256\times 128) = 1/128$, and the associated spill list occupies at most $32/128 = 25\%$ of the bitvector size. For LOUDS-encoded tries, since only half of the bits are zeros, the bound can be reduced to $25\% \times 128/127/2 = 12.6\%$. For LOUDS-Sparse encoded tries, the functional index is sampled on only one of the two bitvectors, which also tightens the bound to 12.5%. Both upper bounds are reached only under extreme conditions where most trie edges are concentrated in a few large nodes, which is unlikely in practical datasets.

3.4 Applying the Rules

We apply the above optimizations to FST, CoCo-trie, and Marisa; Figure 10 shows the resulting bitvector layouts.

The FST is encoded using LOUDS-Sparse and only needs to support existence query. We pack the blocks of `HasChild` and `LOUDS` (Section 3.1) and interleave bits with the $\texttt{rank}_1$ (Section 3.1) and `child` (Sections 3.2 and 3.3) indexes.

The CoCo-trie is encoded using standard LOUDS and supports top-down lookup, and therefore an ideal approach is to inline (primary and secondary) $\texttt{rank}_1$ and $\texttt{rank}_{00}$ indexes (Section 3.1) and the `child` (Section 3.2, Section 3.3) functional index. However, this adds a 192-bit space overhead to every 256-bit block, which is too expensive. Therefore, we switch to LOUDS-Sparse encoding which incurs less index overhead.

The case with Marisa is more involved. First, we change the encoding scheme from LOUDS to LOUDS-Sparse, as this allows for better alignment. Then, we pack the blocks of `HasChild`, `LOUDS` and `IsLink`, as all three bitvectors are now edge-aligned and are typically accessed with high locality (Section 3.1). Finally, we interleave the bits with indexes for `child` and `parent` (Section 3.2, Section 3.3), and each bitvector's $\texttt{rank}_1$ (Section 3.1) indexes, as the Marisa trie needs to support both forward and reverse lookups.

Theoretical analysis. The C_1 optimizations preserve asymptotic bounds on space and query complexity. The rank and select indexes are lower-order terms relative to the bitvector size [44].

Lemma 3.1. *Given a bitvector of size S and a block size of B, the C_1 optimizations incur at most $S/2B$ additional space overhead.*

Proof. The size of the rank index in C_1 is unchanged and takes 32 bits per block. Since only half of the bits in a trie encoding are 1, the $\texttt{select}_1$ index contains about $S/2B$ elements, whereas the `Child` functional index contains S/B elements. □

Although functional indexes use more space compared to traditional `select` indexes, the overall space overhead is small relative to the index size and label data. For example, when $B = 256$, the functional index takes 12.5% extra space relative to the bitvector size compared to the `select` index.

Lemma 3.2. *A child navigation takes at most 4 cache misses in all of the C_1-optimized tries in this paper.*

Proof. Let C_1-CoCo, C_1-FST, and C_1-Marisa denote the CoCo-trie, FST, and Marisa trie with the C_1 optimization as illustrated in Figure 10. As shown in Figure 10, C_1-CoCo, C_1-FST, and C_1-Marisa have block sizes of 704, 704, and 1024 bits, respectively. Given a cache-line size of 64 bytes (512 bits), each block in the C_1-optimized succinct tries takes at most 2 cache lines. A child navigation with the functional index queries two blocks: the input block and the output block. □

Although the blocks in the C_1-optimized tries span 2 cache lines each, the second cache line in the block is prefetched and therefore incurs no additional cache misses. In contrast, the original succinct tries (e.g., CoCo, FST, and Marisa), are encoded using separate bitvectors, which take at least 3 random cache misses per block.

4 The Second "C": Adaptive Unary-Path Compression

In this section, we address the space inefficiency incurred by unary paths (especially suffix unary paths) and introduce the second "C" of C^2: the adaptive unary-path Compression scheme (denoted with C_2). The C_2 optimization enables all tries to access unary-path compression in the data. Specifically, it combines Marisa's recursion [42] with the Fast Static Symbol Table (FSST) [14] as the tail container. Furthermore, it *adaptively* chooses the number of levels of recursion for the Marisa trie based on per-dataset space savings.

We evaluate C^2 tries against baselines at the same recursion level; since CoCo-trie and FST originally lack recursion, C_2 changes only their tail container.

Locality issues in the data. Long unary paths increase topology size and waste space through redundant label sequences. Without proper compression, suffix paths can account for about 80% of succinct-trie space, as shown in Table 2.

Prior work uses two orthogonal compression methods (detailed in Section 2): PDT [23] uses re-pair [27], while Marisa [42]

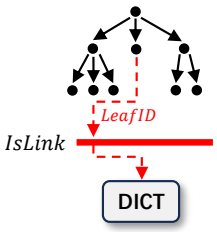

Figure 11: Suffix path compression for C²-FST and C²-CoCo.

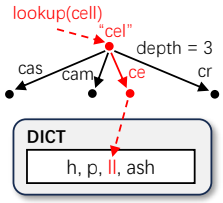

Figure 12: A C²-CoCo suffix path compression example. Stored keys: cash, camp, cell, crash.

uses recursion, as detailed in Section 2. Neither method dominates: on the `log` [18] dataset, recursion and re-pair achieve compression ratios of 16.4× and 7.5×, respectively; on `xml` [35], the ratios are 3.9× and 5.0×.

Choice of tail container. Although Marisa originally uses a sorted tail container, recursion is agnostic to the tail container; more advanced containers such as approximate re-pair [23] and FSST [14] can improve both space and query time.

We limit our C² tries to the FSST tail container because it captures almost all (within 1.12×) of the compression benefits of approximate re-pair with 12.8×–21.6× faster build times. As shown in Section 5, choosing FSST as the tail container always achieves a better compression ratio (up to 1.3×) with minimal impact on the query time (.96-1.13×) over the sorted container.

Integration with succinct tries. First, we enable C_1-FST, C_1-CoCo, and C_1-Marisa to access recursion and compressed tail containers. We focus the discussion on FST and CoCo, as these previously could not access unary-path compression.

Figure 11 illustrates how we containerize the storage of suffix paths in FST. Specifically, each leaf node has an `IsLink` bit indicating whether its suffix path is moved to the next container. A suffix link at position i can thus be uniquely identified by `LinkId(i) = IsLink.rank`$_1$`(LeafID(i))`. When a lookup operation reaches a leaf node, we follow the `IsLink` bitvector to check the remaining suffixes in the container (if any).

The integration with C_1-CoCo is similar to C_1-FST, but CoCo requires special care for correctness: its encoding schemes support only exact lookup. Figure 12 shows the issue. When looking up the string "cell," the CoCo-trie would search for "cel" in the root because the root node has depth 3. Since "cel" is not a key of the root, the lookup would fail, even if "cell" exists in the trie. Instead, we search for the lower bound of the target key rather than an exact match. If the lower bound is a prefix of the target key, we trace the link to check the remaining suffixes.

Adaptive recursion depth. We enable all of the succinct tries studied in this paper to access unary-path compression, but focus on the case of the Marisa trie and show how to achieve better space-time tradeoffs by combining recursion with FSST.

Using recursion in the other tries (FST and CoCo) trades query performance for space savings. Therefore, we limit non-Marisa tries to no recursion in our evaluation, but expose it as a user option for different space-time tradeoffs. We will include the data on the effect of recursion in non-Marisa tries in the full version.

The original Marisa-trie exposes the maximum number of recursions as a parameter to the user, but offers no per-dataset guidance on the space-query tradeoff. As the original Marisa trie documentation notes [4] and Section 5 confirms, recursion has a serious negative impact on query performance, so we only want to continue recursion when the space reduction is significant.

We *adaptively select* the number of recursion levels that achieve the best time-space tradeoff per dataset. To balance compression ratio against build/query time, we stop recursion when the space savings would be less than some $\epsilon = 0.1$ (relative to trie size). However, the user can set ϵ to suit their use case.

To efficiently determine the space usage at each level of recursion, we adopt FSST's fast estimation scheme, encoding a subset of the data to approximate the compression ratio. In practice, this estimate is within 10% of the true compression ratio.

Other locality optimizations. For the Marisa trie, compressed tail containers degrade lookup performance in nodes with many links. To address this issue, we store the branching label (i.e., the first label of each unary path) in the label vector, enabling in-place intra-node search accelerated by SIMD [23].

To further improve locality, we store unary paths smaller than a link in-place, avoiding the index overhead of moving them to the next level. The number of bits needed to store a link depends on the size of the trie (i.e., if a trie has n nodes, we need $\lg(n)$ bits to store each link). Short unary paths may be further compressed by moving them to the next level, but the space savings is limited by the short path length relative to the link cost.

5 Experimental Evaluation

We evaluate three succinct tries optimized using C² (C²-FST, C²-CoCo, and C²-Marisa) and compare them with eight state-of-the-art tries: FST [44], Marisa [42], CoCo-trie [11, 12], PDT [23], ART [28], C-ART [43], c-trie++ [39], and z-fast trie [9]. The first four are bitvector-based succinct tries introduced in Section 2; ART is a well-established pointer-based dynamic trie, C-ART is the compact and static version of ART, and c-trie++/z-fast trie are high-performance dynamic tries. We compare all data structures on query performance, space usage, and build time.

Summary. The C² tries significantly outperform the original versions in both performance and space efficiency on most datasets. On average, C²-FST, C²-CoCo, and C²-Marisa improve query performance/space usage by 1.58×/1.27×, 1.13×/1.31×, and 1.42×/1.30×, respectively (at equal recursion).

These improvements stem from improved locality and fewer cache misses in both the index and the data. Applying C_1 (the cache-conscious bitvector redesign in Section 3) reduces cache misses per query by 1.51× for FST and 1.26× for Marisa on large datasets. The bitvector redesign adds at most 4% space overhead. At equal recursion, C_2 (the unary-path compression scheme in Section 4) improves query performance, space usage, or both over C_1 alone.

5.1 Experimental Settings

Hardware. We ran the experiments on a server with an AMD EPYC 9555 64-Core Processor running at 3.2GHz. The server has 1007 GB of memory, a 3 MiB L1 data cache, a 2 MiB L1 instruction cache, a 64 MiB L2 cache, and a 256 MiB L3 cache. We ran the experiments with one thread.

Implementation. We implement the three C² tries (C²-FST, C²-CoCo and C²-Marisa) using C++17. We use the compressed string pool implementation of PDT [1] and the publicly-available implementation [3] of FSST [14] for the re-pair and FSST tail containers, respectively. We also use the sdsl [20, 37], ds2i [34] and sux [31] libraries for C²-CoCo as in the original CoCo-trie [6].

Table 3: Datasets used in evaluation. All dataset sizes are expressed in MB. LCP means length of longest common prefix. *Size is the size of the prefix-only version of each dataset.**

Name	*Size*	*Size**	*#Keys*	*Avg len*	*Avg LCP*	*Description*
`words` [17]	5	4	4.67E5	9	6	English words
`url` [13]	37	17	1.77E6	21	7	UK private domains
`dna` [35]	101	44	3.32E6	31	11	DNA 31-mers
`xml` [35]	117	74	2.15E6	56	33	dblp XML dump
`log` [18]	586	258	4.45E6	137	54	server access logs
`wiki` [41]	359	236	1.75E7	21	11	Wikipedia titles

For CoCo-trie [6], Marisa [4], PDT [1] and C-ART [2], we use their original implementations. We adopt the ART implementation in the C-ART codebase. For FST, we use an optimized third-party implementation [5] which compacts suffixes into a contiguous array, as this allows for fairer comparison with many unary suffix paths. Finally, we also evaluate c-trie++ [39] and z-fast trie [9], two state-of-the-art dynamic tries.

All tested systems support existence queries, but out of the tested C^2 tries, only FST supports an iterator for range queries. Therefore, we evaluate FST versus C^2-FST on range queries. In each experiment, we generate range queries with a start key and a length, or number of subsequent keys to return beginning from the start key. We vary the query lengths k between experiments and test $k = 1, 10, \ldots, 10000$.

The ART and C-ART implementations we use are designed as filters but not indexes and may return false positives. To support lossless queries, the original dataset must be replicated and may need to be checked when the filter returns positive results. We report space costs without the replicated dataset; query times include false-positive checks.

Finally, as detailed in Section 5.2, we introduce CoCo', a version of the CoCo-trie with an optimized build routine but the same bitvector design, because the open-source CoCo-trie does not build on large datasets.

All code is compiled with g++ 10.3.0 at -O3.

All implementations used in our experiments are publicly available at https://github.com/alexztc/C2.

Parameter settings. For FST, we use the default setting with $R = 64$. For CoCo-trie, CoCo', and C^2-CoCo, we set $\alpha = 5\%$ for a good space-time balance. We do not enable LOUDS-Dense for C^2-FST as it provides no benefit. We use Marisa's default cache size for both Marisa and C^2-Marisa, which is 1/512 of the key count. We set $\epsilon = .1$ for all C^2 tries.

Datasets. Table 3 details the six datasets in our evaluation. These datasets span diverse sources, key lengths, prefix depths, and alphabet sizes.

Furthermore, following the methodology from the CoCo-trie paper [12], we generate prefix-only versions of each dataset (denoted by `dataset*`) because CoCo originally used only prefix-only datasets. We compare the open-source CoCo implementation with our CoCo' implementation on the prefix-only datasets.

Experimental setup. We set up experiments as follows: First, each dataset is sorted and deduplicated. Second, we build the trie under test on the dataset and record the build time and memory usage. Finally, we query the trie using all keys in the dataset in random order and compute the average query latency. We omit negative-query latencies, as they are generally proportional to positive workloads [12]. We also measure the cache performance of queries in different tries on the `log` and `wiki` datasets by recording the number of cache misses using `perf`. All times are the average of three trials after one warm-up trial.

Table 4: Comparison of CoCo and CoCo′ (Query, in ns per query) and space usage (Size, in % of original dataset size) on the prefix-only datasets used in the original CoCo-trie evaluation.

	`words*`		`url*`		`dna*`		`xml*`	
Trie	*Query*	*Size*	*Query*	*Size*	*Query*	*Size*	*Query*	*Size*
CoCo	315	62.7%	742	42.4%	655	24.0%	1039	47.5%
CoCo′	380	54.8%	811	32.3%	743	40.1%	1011	29.6%

Notation. We use Marisa-*i* to denote Marisa with *i* levels of recursion. Furthermore, for a given trie X, we use C_1-X to denote X optimized with C_1 (Section 3). We use C^2-X to denote X with both C_1 and C_2. For example, C_1-FST is C_1-optimized FST with sorted tail container, whereas C^2-Marisa-1 is C_1-optimized Marisa with FSST tail container and one level of recursion.

5.2 Building C^2-CoCo on larger datasets

Since the original CoCo-trie implementation runs out of memory when building the larger prefix-only datasets, we implement an optimized build routine underneath C^2-CoCo to support all datasets. The original CoCo is built from a pointer-based uncompacted trie, which not only consumes excessive memory but also incurs significant cache misses when traversing each node's descendants at different levels, a key operation of CoCo-trie's optimization process. We optimize the builder by representing the uncompacted trie as C^2-FST, which significantly reduces cache misses because in LOUDS-Sparse representation each node's descendants at the same level fall into contiguous memory. Our implementation reduces the build time of C^2-CoCo by up to 14× and uses orders of magnitude less memory during build than the original CoCo-trie.

We introduce ***CoCo'***: C^2-CoCo's build routine with the original CoCo-trie bitvector, to isolate bitvector differences.

We use CoCo' as the CoCo-trie baseline for larger datasets. As shown in Table 4, on the prefix-only datasets that the original CoCo-trie can build on, CoCo-trie and CoCo' have similar query latencies (within 100 ns). However, the original CoCo-trie suffers from poor performance and space usage on full datasets because it does not handle unary suffix paths. For example, on `url`, a relatively small dataset, the original CoCo-trie implementation requires more than 10 minutes and 60 GB of memory to build. Furthermore, it takes over 2000 ns/query and occupies nearly half as much memory as the original dataset, both the worst among all tested tries.

5.3 Ablation Study and Sensitivity Analysis

Figure 1 shows ablation results of C_1 and C^2 across configurations of FST, CoCo, and Marisa. Table 6 contains the full data for the different C^2 configurations discussed in Sections 5.3 and 5.4.

Effect of hybrid bitvector. Table 5 evaluates the effect of switching to LOUDS-Sparse in FST [44]. As mentioned in Section 2, the original FST introduced a hybrid bitvector with LOUDS-Sparse and LOUDS-Dense. We find that the hybrid bitvector improves the performance of both build and query in the baseline FST (with

Table 5: Normalized build and query time (relative to C²-FST; lower is better) for different LOUDS-Sparse/Dense configurations of FST. FST-Sparse uses LOUDS-Sparse/LOUDS-Dense, while FST-Hybrid uses LOUDS-Sparse, both with a sorted tail container. C²-FST uses the same bitvectors with the FSST tail container.

	words (0.47M keys)		*log* (4.45M keys)	
	Build	*Query*	*Build*	*Query*
FST-Sparse	7.73×	3.50×	2.30×	2.73×
FST-Hybrid	7.02×	2.85×	2.24×	2.57×
C²-FST-Sparse	5.31×	1.52×	1.53×	1.82×
C²-FST-Hybrid	5.29×	1.45×	1.55×	1.85×

no special tail container). However, the effect is much smaller on C²-FST with the FSST tail container, so we switch to the fully sparse version in the final C² configuration.

Effect of cache-optimized bitvector design. First, we isolate the effect of C_1, the cache-optimized bitvector design. We perform the ablation study on the datasets in Table 3 (1) with the sorted tail container and no recursion to avoid confounding factors from path compression, and (2) with all branching labels stored out of place to suppress intra-node search effects, so that the remaining query-time difference is attributable to the bitvector layout alone.

The ablation study in Table 6 shows that the C_1 optimization improves query performance by 1.58×, 1.12×, and 1.42×, respectively, compared to the original FST, CoCo-trie, and Marisa. These trie-level gains compound directly from the per-operator speedups in Table 7, since every full lookup issues $O(depth)$ rank queries on the topology bitvector and a small constant number of select queries, each of which benefits from the co-located block metadata in our layout.

The bitvector redesign has the least impact in the CoCo-trie because the query time in CoCo is dominated by the cost to read the encodings, and each CoCo macro-node already amortises several traversal steps over one bitvector access. However, FST and Marisa improve significantly because bitvector cache misses dominate query time: each rank query in their baselines incurs two cache misses — one for the block summary and one for the bit field — which our co-located layout collapses to a single line.

Table 7 reports the effect of C_1 on the fine-grained bitvector operations that make up succinct-trie navigation. The results demonstrate that C_1 improves the performance of key rank-based operations such as `leaf_id` and `internal_id` by $1.2 - 7.1\times$. We observe similar trends for trie-specific operations, such as parent navigation in Marisa (1.76× on `parent_pos`) and child navigation across all three tries ($1.3 - 2.8\times$ on `child_pos`). The only operators where the baseline is faster are `get` and `degree`, which perform linear scans through bits and do not use rank and select; interleaving block metadata with the bit field slightly reduces the effective bit density per cache line on such scans. However, neither of these lie on the critical path of a full lookup or range query, and both are invoked $O(1)$ times per query at sub-3-ns cost each, so the few nanoseconds lost are dwarfed by the tens of nanoseconds saved on every rank query.

Effect of unary-path compression. Next, we evaluate the effect of the C_2 compression scheme. Specifically, for all tries, we measure the effect of replacing the sorted tail container with FSST. Furthermore, we test different recursion levels to evaluate the space-time tradeoff exposed by more recursions.

Replacing the sorted tail container with FSST improves space usage by 1.27×, 1.31×, and 1.30× over C_1-FST, C_1-CoCo, and C_1-Marisa, respectively. The effect on query performance is minimal (1-1.05× on average).

As shown in Figure 13, recursion levels 0 and 1 expose Pareto-optimal space-time trade-offs. For example, on C²-Marisa, on the `log` dataset, no recursion takes 829 ns/query, while one level of recursion takes 1308 ns/query. On the other hand, the space usage improves from 20.7% to 14.7%. Similarly, on the `wiki` dataset, no recursion takes 732 ns/query compared to 979 ns/query with one level of recursion. For `wiki`, one level of recursion improves the space usage from 32.9% to 28.3%.

The same pattern holds for C²-FST and C²-CoCo. On the `log` dataset, C²-FST without recursion takes 2187 ns/query at 24.4% space, while one level of recursion takes 2502 ns/query at 20.1% space; on the same dataset, C²-CoCo without recursion takes 1463 ns/query at 23.9% space, while one level of recursion takes 1745 ns/query at 19.6% space. The same trade also holds on `wiki`, `xml`, `url`, and `dna`: across our six datasets, the absolute space savings from recursion levels 0 to 1 range from 0 percentage points (e.g., `words`, whose avg. longest common prefix of 6 leaves no recursive sub-prefix structure to extract) to roughly 6 percentage points (`log` across all three variants), while the corresponding latency degradation stays below 1.5× in every cell.

Beyond one level of recursion, further recursion's space savings rarely justify the performance penalty. For example, if we continue to recurse C²-Marisa-1 on `log`, the relative space usage improves by 1.6× (which corresponds to a 5.3 percentage point improvement relative to the original data size, from 14.7% to 9.4%), but the query latency degrades by 1.25× (from 1308 to 1593 ns/query); on `xml`, the same step yields no measurable further space improvement. In fact, `log` is the only dataset on which ρ=2 strictly Pareto-dominates ρ=1 across all three tries, owing to its 113-character average key length harbouring nested common prefixes that a second recursion round can still extract. We observe one further anomaly: on `dna`, C²-Marisa is fully insensitive to recursion — its size remains at 41.1% of the original data and its latency varies by less than 1% across $\rho \in \{0, 1, 2\}$ — because its unary-path encoder already absorbs DNA's unique-suffix structure at ρ=0, leaving no inter-tail prefix redundancy for the adaptive cost-based recursive string-pool builder to exploit.

Going forward, we limit the recursion level of C²-FST and C²-CoCo to 0 to match the original designs. For C²-Marisa, we use the recursion level prescribed by the adaptive compression scheme. We expose recursion as an option for space-constrained settings: dialling C²-Marisa up to ρ=2 on long-key datasets like `log` trades a further 5 percentage points of space for a 1.25× query latency penalty, while on every other dataset the same dial yields at most one percentage point of additional space and is therefore Pareto-dominated by ρ=1.

5.4 Comparison to State-of-the-art Tries

We compare C² tries with all baselines. Table 6 reports the full data and the configurations prescribed by the C_2, the adaptive compression algorithm.

Existence queries. Among all tries, ART always has the best query performance at the cost of high space usage. Child navigation in ART simply requires following a pointer, while succinct tries must perform complex bitvector operations. Furthermore,

Table 6: Build time, query latency, and space usage on the datasets in Table 3. The prefixes C_1/C^2 indicate that the C_1/C_1 +C_2 optimizations are applied. We fix the tail container to be FSST with the C_2 optimization, but we report results with the sorted tail container for the ablation study on the C_1 optimization. Marisa-1 means the Marisa trie with 1 recursion level. For Marisa, we show all data points before C_2 stops recursion. For all other tries, the recursion level was set to 0 for a fair comparison. For each column, green cells highlight the best value and blue cells highlight the second-best value.

	Build (ns/key)						*Query (ns/key)*						*Size (% of original size)*					
Trie	words	url	dna	xml	wiki	log	words	url	dna	xml	wiki	log	words	url	dna	xml	wiki	log
FST	309	691	802	1157	716	2139	403	610	844	1541	1424	4003	45.5%	35.5%	74.3%	39.2%	37.9%	39.5%
C_1-FST	90	1043	1148	897	762	2173	228	404	562	835	1242	2197	40.4%	39.8%	75.9%	39.7%	38.4%	40.4%
C^2-FST	101	1068	1114	894	729	1821	227	404	561	842	1258	2154	40.4%	36.0%	43.8%	28.8%	37.7%	24.4%
CoCo'	790	1639	1875	1913	2114	7258	389	626	749	1027	1539	1597	51.5%	41.7%	73.2%	40.6%	43.7%	40.2%
C_1-CoCo	758	1595	1919	1949	2145	7562	355	562	695	907	1320	1465	45.8%	39.5%	70.7%	39.8%	41.4%	39.9%
C^2-CoCo	771	1637	1852	1900	2074	7237	358	560	613	898	1342	1448	45.8%	35.7%	38.5%	28.9%	40.7%	23.9%
Marisa	234	470	676	712	521	1270	272	460	663	669	991	1098	33.3%	35.3%	76.4%	40.3%	32.4%	37.3%
C_1-Marisa	131	1046	1175	991	865	2031	178	310	443	483	739	831	37.5%	36.4%	75.4%	39.5%	34.4%	36.4%
C^2-Marisa	168	1064	1139	933	799	1613	182	308	439	500	732	829	37.5%	30.9%	41.0%	27.6%	32.9%	20.7%
Marisa-1	239	558	1002	945	599	1405	301	583	968	960	1244	1538	29.5%	22.0%	34.3%	25.2%	25.1%	28.6%
C_1-Marisa-1	141	1229	2275	1476	1025	2209	178	425	686	746	967	1229	37.5%	26.5%	33.8%	26.3%	28.9%	28.2%
C^2-Marisa-1	169	1294	2369	1473	997	1938	196	424	436	764	979	1306	36.2%	26.3%	41.0%	21.5%	28.3%	14.7%
PDT	624	1582	2947	4593	2040	3912	356	560	795	841	1022	1138	33.7%	26.2%	28.4%	19.7%	30.1%	13.0%
ART	73	83	76	115	92	289	132	347	533	728	713	1611	346.8%	156.2%	115.0%	59.1%	165.7%	27.6%
C-ART	119	150	179	178	220	404	148	401	674	750	810	1758	155.0%	68.6%	53.8%	27.1%	77.5%	16.0%
c-trie++	401	654	365	344	644	315	217	495	683	915	716	1507	1540.1%	747.0%	513.5%	290.6%	788.5%	157.1%
z-fast trie	447	779	989	1587	1168	2531	541	1158	1394	2016	1192	2838	1084.5%	586.6%	400.2%	223.9%	546.6%	92.7%

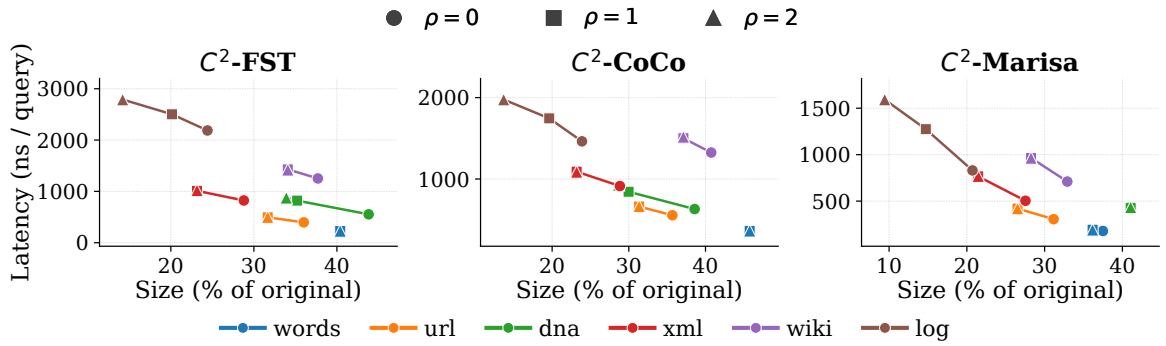

Figure 13: Space-latency Pareto frontier under recursion depths $\rho \in \{0, 1, 2\}$ for the three trie variants across all six datasets.

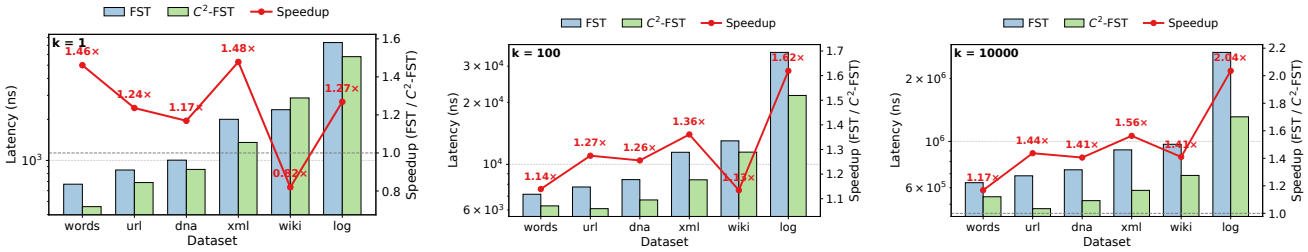

Figure 14: Range-query latency comparison between FST-baseline and C^2-FST across six datasets under different range widths k. Each subplot corresponds to a fixed range width. Bars report absolute latency (left axis), while the red line indicates the speedup of C^2-FST over the baseline (right axis).

C-ART [43] uses an internal-node organization that trades search performance for space savings over ART. Both c-trie++ and z-fast trie are slower than ART for queries while consuming significantly more space.

Table 6 demonstrates that C_2 does not affect the query-time gains from C_1 across tries and datasets. Among the succinct tries, C^2-Marisa achieves the lowest query latency with a space

Table 7: Bitvector Operation Latency (ns) on XML Dataset (2.1M keys). *Speedup* = Baseline / C_1. Bold entries indicate where C_1 outperforms the baseline.

Trie	*Operation*	*Baseline (ns)*	*C_1 (ns)*	*Speedup*
FST	`get`	1.48	1.52	0.98×
	`leaf_id`	10.32	**1.45**	**7.13×**
	`degree`	1.09	2.75	0.40×
	`child_pos`	48.62	**17.09**	**2.84×**
CoCo	`get`	0.64	1.06	0.60×
	`leaf_id`	2.10	**1.72**	**1.23×**
	`internal_id`	4.18	**2.82**	**1.48×**
	`degree`	1.83	2.34	0.78×
	`child_pos`	22.60	**17.87**	**1.26×**
Marisa	`get`	0.98	1.48	0.66×
	`link_id`	8.20	**1.23**	**6.66×**
	`leaf_id`	9.00	**1.38**	**6.53×**
	`child_pos`	30.31	**16.62**	**1.82×**
	`parent_pos`	28.68	**16.29**	**1.76×**

usage within 1.2× of the best. On average, C²-Marisa improves the query latency of the original Marisa by 1.42×.

On `words` and `url`, C²-Marisa-1's space consumption is relatively high compared to Marisa-1 because in this case, the C_2 scheme stores short unary paths in place rather than in the tail container (see Section 4). For example, on the `words` dataset, C²-Marisa contains only about .3× as many links as Marisa.

C²-FST improves query latency/space usage by 1.58×/1.27× on average compared with FST. C²-CoCo does not benefit as much from C_1 as it incurs fewer bitvector operations, but still improves both space usage and query latency compared to CoCo'.

Time-space tradeoff. We next examine query time and space usage. C_2 improves space usage by 1.34× on average compared to their original versions.

On average, with no recursion, switching from the sorted tail container to FSST reduces the space usage by 1.3× at the cost of a 1.05× query performance penalty and 1.1× higher build time across all tested succinct tries.

Additionally, for Marisa, on the datasets where C_2 prescribes one level of recursion, the recursion improves the space efficiency by 1.29× on average, but also incurs a query performance penalty of 1.43× compared to no recursions. Specifically, recursion turns out to be extremely effective on the highly redundant `log` dataset. For C²-Marisa, one level of recursion reduces the space cost by 1.74×, but increases the query time by 1.7× compared to no recursions due to the presence of many recursive paths.

On `words`, `url` and `wiki`, C²-Marisa-1 has higher space usage (by at most 1.19×) than Marisa-1 because of the space overhead of storing the branching labels in place (see Section 4).

On the larger datasets (e.g., `log`), C²-FST and C²-CoCo are generally less competitive than C²-Marisa in both query performance and compression ratio, due to their handling of internal unary paths. C²-FST does not handle such paths at all, and C²-CoCo does not address the data redundancy in such paths. Furthermore, unary paths that do not fit the machine word size will be fragmented in C²-CoCo, impairing data locality.

Range queries. Figure 14 shows that the C² optimizations speed up range queries by $1.2 - 1.5\times$ in FST, which was the only tested trie that natively supports successor queries. The locality improvements in C²-FST carry over into range queries, where each `next()` step can be resolved from data in the packed 256-bit blocks, which are already in cache. In contrast, the original FST fetches the equivalent information from three separately heap-allocated arrays, incurring up to three cache-line misses per step.

The relative advantage of C²-FST increases monotonically with range width, with speedups on all but one configuration (`wiki` with $k = 1$). The `wiki` contains many diverse suffixes, so the FSST tail container in C²-FST finds almost nothing to compress, as shown in Table 6. However, C²-FST must incur additional indirections during traversal to check the `is_link` array and fetch the compressed suffix, while the original FST can do a direct comparison.

Build time. While modern static succinct tries excel at space efficiency and query latency, their build times are significantly worse (between 3.2-24.7×) than the build times of pointer-based tries such as ART and C-ART, especially on large datasets. Depending on the dataset, the C² optimizations can increase the build time by about 2× due to the time it takes to compress the tail container. We chose FSST as the tail container to try to minimize build times compared to approximate re-pair, which can take another 2× longer, as shown in PDT's build times. Furthermore, building is a one-time cost for succinct tries and can be amortized over many queries. Future work will parallelize the build phase of succinct tries.

6 Conclusion

This paper presents C², guidelines for transforming bitvector-based succinct tries to more cache-friendly and compact versions. C² includes two optimizations: C_1, the Cache-conscious bitvector design, and C_2, the adaptive unary-path Compression. Following C², we redesign three state-of-the-art succinct tries: FST, CoCo-trie, and Marisa. On average, the three C²-optimized tries improve query time by 1.58×/1.13×/1.42×, respectively, while using 1.27×/1.31×/1.30× less space.

Future work will 1) apply similar bitvector optimizations to DFUDS/BP tries, like PDT or the DFUDS variant of CoCo-trie, 2) theoretically bound the space usage of practically-efficient compression schemes such as approximate re-pair [23] and FSST [14], and 3) integrate other powerful text-compression algorithms, such as LZW [26], with succinct tries.

Artifacts

All code and experiment scripts used in this paper are publicly available at https://github.com/alexztc/C2. Experiments were run single-threaded on an AMD EPYC 9555 64-core server (3.2GHz, 1007GB RAM), with all code compiled using g++ 10.3.0 at `-O3`. The repository ships a small sample dataset for a quick build check; instructions for obtaining the six full datasets used in our evaluation are given in `DATASETS.md`. After building the project with CMake, the provided scripts reproduce the main performance, recursion-depth, and bitvector-operation results reported in Section 5.

References

[1] 2011. Path Decomposed Tries. https://github.com/ot/path_decomposed_tries.
[2] 2017. C-ART. https://github.com/efficient/fast-succinct-trie/tree/master/third-party/art.
[3] 2020. FSST. https://github.com/cwida/fsst.
[4] 2020. Marisa trie. https://github.com/s-yata/marisa-trie.
[5] 2021. Fast Succinct Trie. https://github.com/kampersanda/fast_succinct_trie.
[6] 2024. CoCo-trie. https://github.com/aboffa/CoCo-trie.
[7] Christoph Anneser, Andreas Kipf, Harald Lang, Thomas Neumann, and Alfons Kemper. 2020. The Case for Hybrid Succinct Data Structures. In *International*

Conference on Extending Database Technology. https://api.semanticscholar.org/CorpusID:214613429

[8] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. 2008. Theory and practice of monotone minimal perfect hashing. *ACM J. Exp. Algorithmics* 16, Article 3.2 (Nov. 2008), 26 pages. doi:10.1145/1963190.2025378

[9] Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. 2010. Dynamic z-fast tries. In *International Symposium on String Processing and Information Retrieval*. Springer, 159–172.

[10] David Benoit, Erik D Demaine, J Ian Munro, Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. 2005. Representing trees of higher degree. *Algorithmica* 43 (2005), 275–292.

[11] Antonio Boffa, Paolo Ferragina, Francesco Tosoni, and Giorgio Vinciguerra. 2022. Compressed String Dictionaries via Data-Aware Subtrie Compaction. In *String Processing and Information Retrieval*, Diego Arroyuelo and Barbara Poblete (Eds.). Springer International Publishing, Cham, 233–249.

[12] Antonio Boffa, Paolo Ferragina, Francesco Tosoni, and Giorgio Vinciguerra. 2024. CoCo-trie: Data-aware compression and indexing of strings. *Information Systems* 120 (2024), 102316. doi:10.1016/j.is.2023.102316

[13] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice & Experience* 34, 8 (2004), 711–726.

[14] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2649–2661.

[15] ByteDance. 2019. Terark Bit Vector. https://github.com/bytedance/terark-zip/tree/dev.1.1/src/terark/succinct.

[16] Francisco Claude and Gonzalo Navarro. 2010. Fast and Compact Web Graph Representations. *ACM Trans. Web* 4, 4, Article 16 (Sept. 2010), 31 pages. doi:10.1145/1841909.1841913

[17] dwyl. 2024. English Words. https://github.com/dwyl/english-words.

[18] Elias Dabbas. 2024. Web Server Access Logs. https://www.kaggle.com/datasets/eliasdabbas/web-server-access-logs.

[19] Paolo Ferragina, Roberto Grossi, Ankur Gupta, Rahul Shah, and Jeffrey Scott Vitter. 2008. On searching compressed string collections cache-obliviously. In *PODS*. Association for Computing Machinery, 181–190. doi:10.1145/1376916.1376943

[20] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. From Theory to Practice: Plug and Play with Succinct Data Structures. In *Experimental Algorithms*, Joachim Gudmundsson and Jyrki Katajainen (Eds.). Springer International Publishing, Cham, 326–337.

[21] Simon Gog and Matthias Petri. 2014. Optimized succinct data structures for massive data. *Software: Practice and Experience* 44, 11 (2014), 1287–1314. doi:10.1002/spe.2198 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2198

[22] Rodrigo González and Veli Mäkinen. 2005. Practical implementation of rank and select queries. In *WEA*. https://api.semanticscholar.org/CorpusID:10701791

[23] Roberto Grossi and Giuseppe Ottaviano. 2015. Fast Compressed Tries through Path Decompositions. *ACM J. Exp. Algorithmics* 19, Article 3.4 (Jan. 2015), 20 pages. doi:10.1145/2656332

[24] Guy Jacobson. 1989. Space-efficient static trees and graphs. In *30th annual symposium on foundations of computer science*. IEEE Computer Society, 549–554.

[25] Shunsuke Kanda, Dominik Köppl, Yasuo Tabei, Kazuhiro Morita, and Masao Fuketa. 2020. Dynamic Path-decomposed Tries. *ACM J. Exp. Algorithmics* 25, Article 1.13 (Sept. 2020), 28 pages. doi:10.1145/3418033

[26] W. Kinsner and R.H. Greenfield. 1991. The Lempel-Ziv-Welch (LZW) data compression algorithm for packet radio. In *WESCANEX*. 225–229. doi:10.1109/WESCAN.1991.160551

[27] N Jesper Larsson and Alistair Moffat. 2000. Off-line dictionary-based compression. *Proc. IEEE* 88, 11 (2000), 1722–1732.

[28] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*. 38–49. doi:10.1109/ICDE.2013.6544812

[29] Donald R. Morrison. 1968. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM* 15, 4 (Oct. 1968), 514–534. doi:10.1145/321479.321481

[30] J Ian Munro and S Srinivasa Rao. 2018. Succinct representation of data structures. In *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 595–610.

[31] Gonzalo Navarro and Eliana Providel. 2012. Fast, small, simple rank/select on bitmaps. In *Proceedings of the 11th International Conference on Experimental Algorithms* (Bordeaux, France) *(SEA'12)*. Springer-Verlag, Berlin, Heidelberg, 295–306. doi:10.1007/978-3-642-30850-5_26

[32] Gonzalo Navarro and Luís MS Russo. 2008. Re-pair Achieves High-Order Entropy.. In *DCC*. 537.

[33] Carlos Ochoa and Gonzalo Navarro. 2018. RePair and all irreducible grammars are upper bounded by high-order empirical entropy. *IEEE Transactions on Information Theory* 65, 5 (2018), 3160–3164.

[34] Giuseppe Ottaviano, Nicola Tonellotto, and Rossano Venturini. 2015. Optimal Space-time Tradeoffs for Inverted Indexes. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining* (Shanghai, China) *(WSDM '15)*. Association for Computing Machinery, New York, NY, USA, 47–56. doi:10.1145/2684822.2685297

[35] Pizza&Chili Corpus. 2024. The Text Collection. https://pizzachili.dcc.uchile.cl/texts.html.

[36] Tetsuo Shibuya. 2019. Application-Oriented Succinct Data Structures for Big Data. *The Review of Socionetwork Strategies* 13 (10 2019), 1–10. doi:10.1007/s12626-019-00045-1

[37] simongog. 2013. SDSL-Lite. https/github.com/simongog/sdsl-lite.

[38] Daniel D. Sleator and Robert Endre Tarjan. 1983. A data structure for dynamic trees. *J. Comput. System Sci.* 26, 3 (1983), 362–391. doi:10.1016/0022-0000(83)90006-5

[39] Kazuya Tsuruta, Dominik Köppl, Shunsuke Kanda, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. 2022. c-trie++: A dynamic trie tailored for fast prefix searches. *Information and Computation* 285 (2022), 104794.

[40] Sebastiano Vigna. 2008. Broadword Implementation of Rank/Select Queries. *In the Proceedings of the 7th International Workshop on Experimental Algorithms*, 154–168. doi:10.1007/978-3-540-68552-4_12

[41] Wikipedia. 2024. Wikipedia titles. https://dumps.wikimedia.org/enwiki/20241120/.

[42] Susumu Yata. 2011. Dictionary compression by nesting prefix/patricia tries. In *Proc. 17th Meeting of the Association for Natural Language*.

[43] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *SIGMOD*. Association for Computing Machinery, 1567–1581. doi:10.1145/2882903.2915222

[44] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *SIGMOD*. 323–336. doi:10.1145/3183713.3196931

[45] Dong Zhou, David Andersen, and Michael Kaminsky. 2013. Space-Efficient, High-Performance Rank and Select Structures on Uncompressed Bit Sequences. In *ESA*. 151–163. doi:10.1007/978-3-642-38527-8_15