Dynamic Grammar-Compressed Self-Index in δ-Optimal Space

Takaaki Nishimoto
RIKEN Center for Advanced Intelligence Project, Tokyo, Japan

Yasuo Tabei
RIKEN Center for Advanced Intelligence Project, Tokyo, Japan

Abstract

A compressed self-index stores a string in compressed form while supporting locate queries without decompression. For highly repetitive strings – arising in web crawls, versioned documents, and genomic collections – static self-indexes can match the δ-optimal lower bound of $\Omega(\delta \log(n \log \sigma/(\delta \log n)) \log n)$ bits up to constant factors, where n is the string length, σ is the alphabet size, and δ is the substring complexity. Their dynamic counterparts, however, remain scarce: every existing dynamic self-index either fails to attain δ-optimal space, pays $\Omega(\log n)$ time per reported occurrence during locate, or has an update time that grows with the maximum value in the longest common prefix (LCP) array of the text. We present the *dynamic RR-index*, a dynamic grammar-compressed self-index built on the restricted recompression run-length straight-line program (RLSLP). To our knowledge, it is the first dynamic self-index to attain δ-optimal space. The index occupies expected $\mathcal{O}(\delta \log(n \log \sigma/(\delta \log n)) \log n)$ bits, answers locate queries in expected $\mathcal{O}(m + \log m \log^2 n + \mathsf{occ}(\log n/\log\log n))$ time – where m is the pattern length and occ is the number of occurrences – and supports insertion of a length-m' string and deletion of a length-m' substring in expected amortized $\mathcal{O}(m' \log^2 n + \log^3 n)$ time, with no dependence on the maximum LCP value. On eleven highly repetitive corpora, including a 37 GB Wikipedia dump and a 59 GB human-chromosome collection, the dynamic RR-index is up to 77× faster than the dynamic r-index on updates and up to 11× faster than other dynamic indexes on locate.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Compressed string indexes, grammar-compression, dynamic data structures

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.6

Related Version *Full Version*: `https://arxiv.org/abs/2604.24080` [41]

Supplementary Material *Software*: `https://github.com/TNishimoto/dynamic-rlslp-index` [38] archived at `swh:1:dir:e7a517c0fd034955d844beea282fceefedac7b63`

Funding This work was supported by JST, ACT-X Grant Number JPMJAX24CL, Japan.

1 Introduction

Datasets composed of *highly repetitive* strings – those with many repeated substrings – have grown rapidly in recent years. Typical examples include web pages collected by crawlers [15], version-controlled documents such as Wikipedia with its complete edit history [49], and, perhaps most notably, biological sequences such as collections of human genomes [45, 48]. These datasets are not static: edits, deletions, and new releases arrive continuously. Yet the methods practitioners rely on most heavily, such as the r-index [16], a compressed index supporting locate queries, are *static*: reflecting even a single edit requires rebuilding the index from scratch. There is therefore a strong and growing need to develop *dynamic* compressed self-indexes for highly repetitive strings that can be updated efficiently without rebuilding.

A practical solution must simultaneously (i) use space proportional to the repetitiveness of the string, (ii) answer locate queries quickly, and (iii) support string insertions and deletions without rebuilding. The third requirement has proven difficult: while compressed *self-indexes* – data structures that represent the string in compressed form while answering locate queries directly – are well understood in the static case, their dynamic counterparts remain challenging to develop.

For highly repetitive strings, compressed self-indexes aim to approach the *δ-optimal space* of $\mathcal{O}(\delta \log(n \log \sigma/(\delta \log n)) \log n)$ bits established by Kociumaka et al. [29], where n is the string length, σ is the alphabet size, and δ is the *substring complexity* of the string – a repetitiveness measure satisfying $\delta \ll n$ on genomic and versioned corpora. A line of specialized self-indexes – grammar-based [8, 29, 28], LZ-based [7], block-tree–based [3, 31, 29], and run-length BWT (RLBWT)-based indexes such as the r-index [16] and OptBWTR [39, 4] – has closed much of this gap, with several attaining δ-optimal space up to constant factors. None, however, supports updates.

Only a handful of *dynamic* self-indexes have been proposed, and each sacrifices at least one of the three requirements (Table 1 in Appendix A gives the full comparison). The dynamic FM-index [47, 33] builds on the Burrows–Wheeler Transform (BWT) [6] but uses $\mathcal{O}(n \log \sigma)$ bits, oblivious to repetitiveness, with an $\mathcal{O}(\log^{2+\epsilon} n)$ per-occurrence locate factor. Grammar-based dynamic indexes – the SE-index [37], the TST-index-d [43], and the index of Gawrychowski et al. [17] – either pay $(\log n \log^* n)^2$-type factors in locate/update or require $\Omega(n \log n)$ bits. A recent dynamic r-index [42] uses $\mathcal{O}(r \log n)$ bits (r the number of BWT runs), but its update time $\mathcal{O}((m' + L_{\max}) \log n)$ depends on the maximum value $L_{\max} \le n$ of the longest common prefix (LCP) array – the length of the longest repeated substring – which can grow independently of δ and r as versioned copies accumulate. In short, every existing dynamic self-index either fails to compress to δ-optimal space, pays $\Omega(\log n)$ time per reported occurrence during locate, or has an update time that grows with the maximum LCP value. We address this gap.

Our contributions. We present the *dynamic RR-index*, a dynamic grammar-compressed self-index built on the *restricted recompression* RLSLP of Jeż [21] and Kociumaka et al. [30, 29]. Our starting observation is that the derivation tree of a restricted recompression RLSLP admits a compact *DAG* representation whose explicit-node count is tied to δ rather than to r or g. Moreover, insertions and deletions perturb this DAG only along two short popped sequences [37, 20], independently of the maximum LCP value. Combined with two-dimensional range reporting [5, 35] and with a path-cached ancestor field that accelerates locate (Section 5.6), this design simultaneously achieves δ-optimal space and a sub-logarithmic per-occurrence locate bound. To our knowledge, the dynamic RR-index is the first dynamic self-index to attain δ-optimal space. Concretely, it occupies expected δ-optimal space – $\mathcal{O}(\delta \log(n \log \sigma/(\delta \log n)) \log n)$ bits – and answers locate queries in expected $\mathcal{O}(m + \log m \log^2 n + \mathsf{occ}(\log n/\log\log n))$ time, where m is the pattern length and occ is the number of occurrences. Insertion of a length-m' string and deletion of a length-m' substring are supported in expected amortized $\mathcal{O}(m' \log^2 n + \log^3 n)$ time. We implement and evaluate the dynamic RR-index on eleven highly repetitive corpora, including the 37 GB enwiki Wikipedia dump and the 59 GB chr19 genomic collection, and we show that it is up to 77× faster than the dynamic r-index on updates and up to 11× faster than other dynamic indexes on locate, while using working memory that empirically scales with δ.

Organization. Sections 2–3 fix notation and review the structural properties of restricted recompression. Sections 4–5 introduce the dynamic RR-index, its DAG representation, the locate algorithm, and the ancestor-path caching of Section 5.6. Section 6 gives the

insertion and deletion algorithms, and Section 7 reports the experiments. Omitted proofs and implementation details are deferred to the full version of this paper [41], available at `https://arxiv.org/abs/2604.24080`; Appendices A and B provide a detailed comparison with existing self-indexes and additional experimental results.

2 Preliminaries

Basic notation. Let T be a string of length $|T| = n \geq 2$ over a totally ordered alphabet Σ of size $\sigma = n^{\mathcal{O}(1)}$. We denote the empty string by ε, the i-th character of T by $T[i]$, the substring of T from position i to position j by $T[i..j]$, and the reverse of T by $\mathsf{reverse}(T) = T[n]T[n-1]\cdots T[1]$. A prefix (resp. suffix) of T is a substring beginning at position 1 (resp. ending at position n); for a string $P \in \Sigma^+$, $T \cdot P$ denotes concatenation. We use the standard lexicographic order on Σ^*. The set of occurrence positions of P in T is $Occ(T,P) = \{i \in \{1,2,\ldots,n-|P|+1\} \mid T[i..i+|P|-1] = P\}$. For a nonempty set $\mathcal{S}$ of integers, $\min \mathcal{S}$ and $\max \mathcal{S}$ denote its minimum and maximum. For a path $\mathbb{P}$ in a graph, let $|\mathbb{P}|$ denote the number of edges on $\mathbb{P}$.

Model of computation. We use base-2 logarithms throughout this paper unless otherwise indicated. Our computation model is a unit-cost word RAM [19] with multiplication, randomization, and machine word size $B = \Theta(\log n)$ bits. We analyze our randomized algorithms against an *oblivious adversary* (e.g., [2]): the sequence of updates to T is chosen in advance, without access to the random bits used by our algorithms.

Substring complexity. Substring complexity [46, 8] measures the repetitiveness of a string. For each $d \geq 1$, let $\mathcal{S}_d = \{T[i..i+d-1] \mid 1 \leq i \leq n-d+1\}$ be the set of length-d substrings of T. The *substring complexity* of T is $\delta = \max_{1 \leq d \leq n} |\mathcal{S}_d|/d$.

The *δ-optimal space* is $\Theta(\delta \log \frac{n \log \sigma}{\delta \log n} \log n)$ bits: every string can be encoded within this space bound, and this bound is tight in n, σ, and δ [29].

Grammar-based compression. A grammar-based compression of T is a context-free grammar $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{D}, E)$ that generates exactly T, where $\mathcal{V} = \{X_1, X_2, \ldots, X_g\}$ is a set of *nonterminals*, Σ is the alphabet of T, and $\mathcal{D}$ is a set of *production rules* $X_i \to \mathsf{expr}_i$ with $\mathsf{expr}_i \in (\mathcal{V} \cup \Sigma)^*$. A rule $X_i \to \mathsf{expr}_i$ expands X_i to the sequence expr_i. The *start symbol* $E \in \mathcal{V}$ does not appear on any right-hand side, while every other nonterminal appears on at least one right-hand side. To avoid cycles, we require $j < i$ whenever X_j appears in expr_i. In addition, $\mathsf{expr}_i \neq \mathsf{expr}_j$ for any two distinct integers $i, j \in \{1, 2, \ldots, g\}$.

Each nonterminal $X \in \mathcal{V}$ derives a substring of T; in particular, the start symbol E derives T itself. The tree whose root is labeled E and whose leaves spell out T is the *derivation tree* of $\mathcal{G}$. Let $\mathsf{val}(X)$ denote the substring of T derived by $X \in \mathcal{V}$, and let $\mathsf{val}(X_1, X_2, \ldots, X_d) = \mathsf{val}(X_1) \cdot \mathsf{val}(X_2) \cdot \ldots \cdot \mathsf{val}(X_d)$ for nonterminals $X_1, X_2, \ldots, X_d \in \mathcal{V}$.

A *straight-line program* (SLP) [22] is a grammar-based compression in which each nonterminal produces either a character or a pair of nonterminals. A *run-length SLP (RLSLP)* [36] is an SLP whose nonterminals can additionally produce a single nonterminal or a repetition of a single nonterminal. Specifically, $\mathcal{D} \subseteq \mathcal{V} \times (\Sigma \cup \mathcal{V}^+)$, and each production rule in $\mathcal{D}$ takes one of the following forms: (i) $X_i \to c$ for $c \in \Sigma$; (ii) $X_i \to X_j X_k$ $(j, k < i)$ for distinct nonterminals $X_j, X_k \in \mathcal{V}$; (iii) $X_i \to X_j$ $(j < i)$ for $X_j \in \mathcal{V}$; (iv) $X_i \to (X_j)^d$ $(j < i)$, where $(X_j)^d$ denotes $d \geq 2$ successive copies of $X_j \in \mathcal{V}$. Several algorithms build an RLSLP for a given string, e.g., *signature encoding* [36], *recompression* [21], *signature grammar* [7], and *restricted block compression* [28].

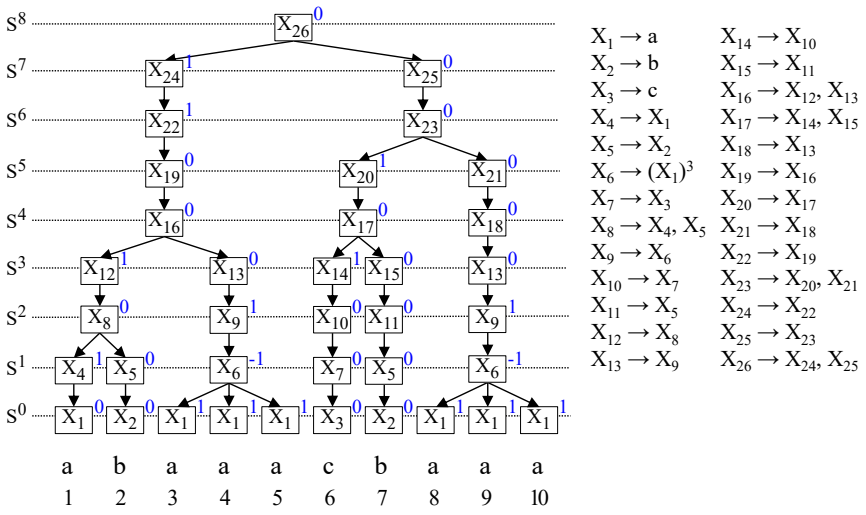

Figure 1 An illustration of the derivation tree of a restricted recompression RLSLP (left), and the corresponding production rules (right). Each node is depicted as a rectangle enclosing its corresponding nonterminal X_i. The blue integer shown at the upper right of each node is the value of $\mathsf{assign}(X_i)$.

3 Restricted Recompression

Restricted recompression [21, 29, 30] is a randomized algorithm that constructs an RLSLP $\mathcal{G}^R$ whose derivation tree is *height-balanced* (i.e., all children of every node have the same height). Let H denote the height of the derivation tree of $\mathcal{G}^R$. For each $h \in \{0, 1, \ldots, H\}$, the nonterminals labeling the nodes at height h, read left to right, form a sequence S^h. In this paper, $\mathcal{G}^R$ is called a *restricted recompression RLSLP*.

Restricted recompression builds the derivation tree bottom-up, processing $S^0, S^1, \ldots, S^H$ in order. While processing S^h, each newly introduced nonterminal X_i in S^h is assigned a value $\mathsf{assign}(X_i) \in \{-1, 0, 1\}$ as follows. Let $\mu(h) = (8/7)^{\lceil (h+1)/2 \rceil - 1}$. If $|\mathsf{val}(X_i)| \leq \mu(h)$, then $\mathsf{assign}(X_i)$ is drawn uniformly at random from $\{0, 1\}$; otherwise, $\mathsf{assign}(X_i) = -1$. The assign values partition S^h into segments as follows. For odd h, every two consecutive nonterminals X_i, X_j in S^h with $\mathsf{assign}(X_i) = 1$ and $\mathsf{assign}(X_j) = 0$ form a segment; for even h, every maximal run of a nonterminal X_i with $\mathsf{assign}(X_i) \in \{0, 1\}$ forms a segment. In either case, every remaining nonterminal forms a segment by itself. Unless $|S^h| = 1$, S^{h+1} is constructed by replacing each segment with a nonterminal. See [41] for a detailed description of the construction.

Figure 1 shows an example of the derivation tree for $T = \mathsf{abaaacbaaa}$. Since the actual values of $\mu(h)$ would make the tree too large to display, we use $\lfloor \mu(0) \rfloor = \lfloor \mu(1) \rfloor = 1$ and $\lfloor \mu(2) \rfloor = \lfloor \mu(3) \rfloor = \cdots = \lfloor \mu(8) \rfloor = 10$ in the figure.

The following lemma shows that $H = \mathcal{O}(\log n)$ with high probability.

▶ **Lemma 1.** *$H \leq \lceil 2(w+1) \log_{8/7}(4n) + 2 \rceil$ holds with probability at least $1 - 1/n^w$ for every integer $w \geq 1$.*

Proof. See the full version [41]. ◀

For a user-defined constant $w \geq 2$, we assume $H \leq \lceil 2(w+1) \log_{8/7}(4n) + 2 \rceil$ throughout this paper. By Lemma 1, this bound holds with high probability.

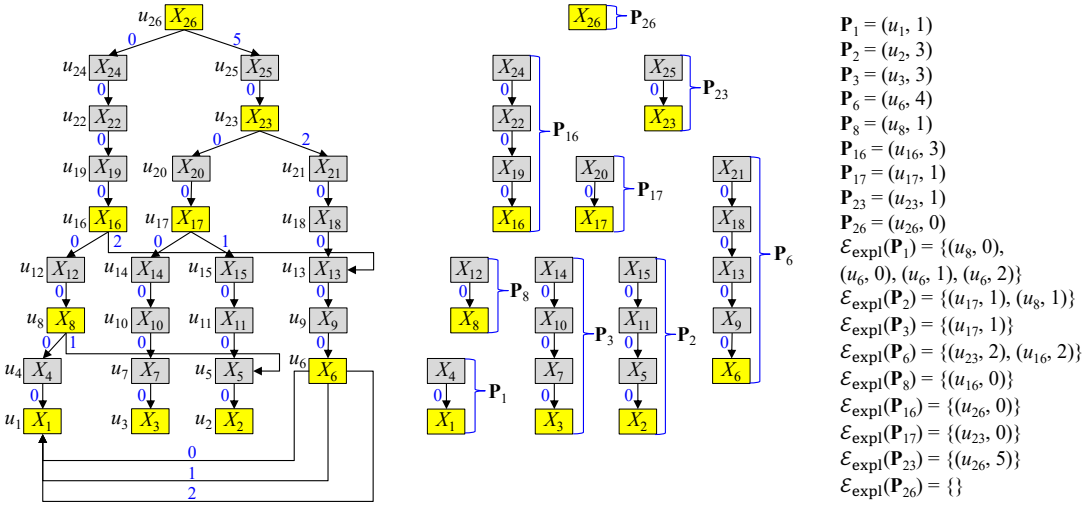

Figure 2 (Left) DAG for the derivation tree of Figure 1: each rectangle is a node u_i with nonterminal X_i (yellow = explicit, gray = implicit), and the blue number above each edge is its label. (Center) The nine paths obtained by removing edges whose source is explicit. (Right) Each path together with its associated set of inter-path edges. For simplicity, each edge is represented as a pair consisting of its source and edge label.

Let M denote the number of nonterminals in $\mathcal{V}$ excluding those that produce a single nonterminal. By Proposition V.19 of [29], $\mathbb{E}[M] = \mathcal{O}(\delta \log \frac{n \log \sigma}{\delta \log n})$. Using this bound on M, Section 4 represents the restricted recompression RLSLP as a DAG that achieves expected δ-optimal space.

4 Dynamic RR-index

The *dynamic RR-index* is built on the restricted recompression RLSLP $\mathcal{G}^R = (\mathcal{V}, \Sigma, \mathcal{D}, E)$ introduced in the previous section. It consists of two components: (i) a directed acyclic graph (DAG) representing the derivation tree, and (ii) a set of points in two-dimensional space corresponding to explicit nodes in the DAG. The details of these two components are described in the following subsections.

4.1 DAG Representation of Derivation Tree and Path Decomposition

DAG representation. The derivation tree of $\mathcal{G}^R$ is represented as a directed acyclic multigraph [36, 37] $\mathfrak{D} = (\mathcal{U}, \mathcal{E}, \mathrm{src}, \mathrm{dst}, \mathcal{L}_U, \mathcal{L}_E)$ obtained by merging nodes labeled with the same nonterminal: $\mathcal{L}_U : \mathcal{U} \to \mathcal{V}$ is a bijection between nodes and nonterminals, $\mathcal{E}$ is the set of directed edges; for $e \in \mathcal{E}$, $\mathrm{src}(e)$ is its source (corresponding to a parent node in the derivation tree) and $\mathrm{dst}(e)$ its destination (a child node), and the outgoing edges of u_i are in one-to-one correspondence with those of any node labeled $\mathcal{L}_U(u_i)$ in the derivation tree. The label $\mathcal{L}_E(e) \in \{0, 1, \ldots, n-1\}$ is $j - i$, where i and j are the starting positions of the substrings derived from $\mathrm{src}(e)$ and $\mathrm{dst}(e)$. Let $d^+(u)$ denote the outdegree of u. Figure 2 (left) illustrates this for Figure 1.

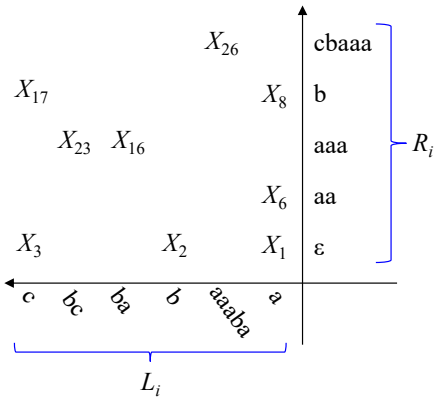

Figure 3 Set $\mathcal{P}$: each nonterminal represents the corresponding point in two-dimensional space.

Path decomposition. We partition $\mathcal{U}$ into $\mathcal{U}_{\mathsf{expl}} = \{u \in \mathcal{U} \mid d^+(u) \neq 1\}$ and $\mathcal{U}_{\mathsf{impl}} = \mathcal{U} \setminus \mathcal{U}_{\mathsf{expl}}$. Nodes in $\mathcal{U}_{\mathsf{expl}}$ are called *explicit*, and nodes in $\mathcal{U}_{\mathsf{impl}}$ are called *implicit*. After removing every edge whose source is explicit, the resulting subgraph $\mathfrak{D}'$ consists of M disjoint directed paths, each ending at an explicit node (see Figure 2, center).

For each explicit node $u \in \mathcal{U}_{\mathsf{expl}}$, let $\mathbb{P}_u$ denote the directed path in $\mathfrak{D}'$ that ends at u. Writing $\langle u, k\rangle$ for the implicit node k edges before u on this path, $\mathbb{P}_u$ has the form

$$\mathbb{P}_u : \langle u, |\mathbb{P}_u|\rangle \to \langle u, |\mathbb{P}_u| - 1\rangle \to \cdots \to \langle u, 1\rangle \to u.$$

Therefore, $\mathbb{P}_u$ can be encoded as the pair $(u, |\mathbb{P}_u|)$.

The *path decomposition* of $\mathfrak{D}$ is the set $\Pi = \{\mathbb{P}_u \mid u \in \mathcal{U}_{\mathsf{expl}}\}$, so $|\Pi| = |\mathcal{U}_{\mathsf{expl}}| = M$ (see Section 3). The directed edges of $\mathfrak{D}$ partition into two kinds: *intra-path edges*, which are the edges of $\mathfrak{D}'$ (equivalently, edges between consecutive nodes on the same path in Π), and *inter-path edges*, whose source and destination lie on different paths in Π.

4.2 Point Representation of Explicit Nodes

To support efficient locate queries, we map each explicit node $u_i \in \mathcal{U}_{\mathsf{expl}}$ to a point p_i in two-dimensional space with x-coordinate L_i and y-coordinate R_i, defined from the production rule of $X_i = \mathcal{L}_U(u_i)$ as follows: if $X_i \to X_j X_k$ then $L_i = \mathsf{reverse}(\mathsf{val}(X_j))$ and $R_i = \mathsf{val}(X_k)$; if $X_i \to c$ for $c \in \Sigma$ then $L_i = c$ and $R_i = \varepsilon$; if $X_i \to (X_j)^d$ with $d \geq 2$ then $L_i = \mathsf{reverse}(\mathsf{val}(X_j))$ and $R_i = \mathsf{val}((X_j)^{d-1})$. Let $\mathcal{P}$ be the resulting point set, and let $\mathcal{X}$ and $\mathcal{Y}$ be $\mathcal{U}_{\mathsf{expl}}$ sorted lexicographically by L_i and R_i, with ties broken by u_i. This point representation will later allow us to find the relevant explicit nodes for locate queries by two-dimensional range reporting, i.e., by queries that return all points in a given axis-aligned rectangle. Figure 3 illustrates $\mathcal{P}$, $\mathcal{X}$, and $\mathcal{Y}$ for the DAG in Figure 2.

4.3 Dynamic Data Structures for the DAG and Point Set

The DAG $\mathfrak{D}$ is indexed by $\mathcal{D}_{\mathsf{DAG}}(\Pi)$ and the point set $\mathcal{P}$ by $\mathcal{D}_{\mathsf{grid}}(\mathcal{P})$.

Data structure for path decomposition. $\mathcal{D}_{\mathsf{DAG}}(\Pi)$ consists of a doubly linked list and two types of hash tables [12]. Intra-path edges are implicit in the list structure; inter-path edges are stored in per-path hash tables. For each explicit node $u \in \mathcal{U}_{\mathsf{expl}}$ with $X_i := \mathcal{L}_U(u)$, the list

stores one element associated with the endpoint u that records: a pair $(u, |\mathbb{P}_u|)$ for recovering $\mathbb{P}_u$, the production rule $X_i \to \mathsf{expr}_i$, the height of u, $|\mathsf{val}(X_i)|$, and $\mathsf{assign}(\mathcal{L}_U(u_j))$ for each node u_j on $\mathbb{P}_u$ (occupying $\mathcal{O}(H)$ bits since $|\mathbb{P}_u| \leq H+1$ and each label is 2 bits). For each path $\mathbb{P}_u \in \Pi$, the set $\mathcal{E}_{\mathsf{expl}}(\mathbb{P}_u) = \{e \in \mathcal{E} \mid \mathrm{src}(e) \in \mathcal{U}_{\mathsf{expl}},\ \mathrm{dst}(e) \text{ lies on } \mathbb{P}_u\}$ is indexed by a per-path hash table whose entry for e has key $\mathrm{src}(e)$ and value $\mathrm{dst}(e)$ (only distinct key-value pairs are stored); the keys are also linked in a doubly linked list sorted by the height of the corresponding nodes and indexed by a *list indexing data structure* [10]. It requires $\mathcal{O}(kB)$ additional bits and supports random access queries on the doubly linked list in $\mathcal{O}(\log k)$ time, where k is the number of elements in the list; each insertion or deletion of an element takes amortized $\mathcal{O}(\log k)$ time.

A global hash table indexes the M production rules with key expr_i (encoded in $\mathcal{O}(B)$ bits) and value X_i.

Data structure for point set. We index $\mathcal{P} \subseteq \mathcal{X} \times \mathcal{Y}$ by Blelloch's dynamic data structure $\mathcal{D}_{\mathsf{grid}}(\mathcal{P})$ [5] for range reporting and point insertion/deletion, which uses $\mathcal{O}(MB)$ bits of space and requires $\mathcal{O}(1)$-time comparison of elements of $\mathcal{X}$ and $\mathcal{Y}$. We maintain *order maintenance data structures* [11] on $\mathcal{X}$ and $\mathcal{Y}$ to provide $\mathcal{O}(1)$-time comparison and $\mathcal{O}(\log M)$-time insertion/deletion, and list indexing data structures for $\mathcal{O}(\log M)$-time access; each coordinate is a pointer to the corresponding list element of Π, so each point uses $\mathcal{O}(B)$ bits. Each list element of $\mathcal{D}_{\mathsf{DAG}}(\Pi)$ in turn stores pointers into the list indexing and order maintenance structures for $\mathcal{X}$ and $\mathcal{Y}$. Range reporting takes $\mathcal{O}(\log M + \mathit{rocc}(\log M/\log\log M))$, where rocc is the number of reported points, and a point insertion or deletion takes amortized $\mathcal{O}(\log M)$ time.

4.4 Space Complexity

The doubly linked list contributes $\mathcal{O}(M(H+B))$ bits; the global hash table, the per-path hash tables (which hold $\mathcal{O}(M)$ distinct key-value pairs in total since every DAG node has at most two distinct children) together with their list indexing structures, and $\mathcal{D}_{\mathsf{grid}}(\mathcal{P})$ with its auxiliary structures each contribute $\mathcal{O}(MB)$ bits. Summing, the dynamic RR-index occupies $\mathcal{O}(M(H+B))$ bits. Since $B = \Theta(\log n)$ and $\mathbb{E}[M] = \mathcal{O}(\delta \log \frac{n \log \sigma}{\delta \log n})$, the total space is δ-optimal in expectation under the assumption that $H = \mathcal{O}(\log n)$.

5 Locate Query Algorithm

This section presents the algorithm answering a locate query for a pattern P of length m, establishing the baseline complexity bound (Theorem 5); Section 5.6 then improves the per-occurrence factor to $\log n/\log\log n$ via ancestor-path caching (Theorem 6). For $m \geq 2$, the algorithm proceeds in four steps, each detailed in a subsection below: from the derivation tree we extract a specific *core* [34] of P as the *popped sequence* [13, 20], whose run-length encoding has ρ runs, build ρ axis-aligned rectangles in $\mathcal{P}$, apply range reporting to obtain the *primary occurrences* [37], and convert these into $\mathit{Occ}(T,P)$ by traversing the DAG to the root. The case $m = 1$ is handled separately in Section 5.5.

5.1 Step (i): Computing the Popped Sequence

This step computes the popped sequence of P, a specific core of P whose run-length encoding has $\mathcal{O}(\min\{m, H\})$ runs. We first introduce what a core is, and then introduce the popped sequence as a specific construction.

Let $Q = X_1, X_2, \ldots, X_d$ be a sequence of nonterminals (each $X_i \in \mathcal{V}$) such that $\mathsf{val}(Q) = P$. We say that Q is *embedded* in the derivation tree of $\mathcal{G}^R$ at position $s \in \{1, 2, \ldots, n\}$ if, for every $i \in \{1, 2, \ldots, d\}$, some node labeled X_i in the derivation tree derives the substring $T[s_i..s_i + |\mathsf{val}(X_i)| - 1]$ of T, where $s_i := s + \sum_{b=1}^{i-1} |\mathsf{val}(X_b)|$. Such an embedding witnesses an occurrence of P at position s (i.e., $s \in Occ(T, P)$). We call Q a core of P if Q is embedded at every position $s \in Occ(T, P)$.

Since P can have multiple cores, we use a specific one, called the *popped sequence* of P, defined as follows. Restricted recompression is applied to P itself, where production rules are shared with $\mathcal{G}^R$ whenever the right-hand sides agree. At each height, the leftmost and rightmost segments of the current sequence are *popped* (i.e., removed and recorded) unless they consist of two distinct nonterminals, and the other segments are replaced by their parent nonterminals as in Section 3. The popped sequence Q of P is the concatenation of the popped left segments in bottom-up order, followed by the popped right segments in top-down order; $\mathsf{val}(Q) = P$ holds, and Q is a core of P. Its run-length encoding consists of ρ runs, where $\rho = \mathcal{O}(\min\{m, H\})$ and $\mathbb{E}[\rho] = \mathcal{O}(\log m)$. The popped sequence is computed from the derivation tree of $\mathcal{G}^R$ by the algorithm of [13], invoked in phase (i) of Section 5.5.2. See [41] for the precise definition and properties of the popped sequence. The next step uses these ρ runs to construct ρ axis-aligned rectangles; Lemma 2 ensures this suffices for finding all primary occurrences.

5.2 Step (ii): Computing the Rectangles from the Popped Sequence

Primary occurrences and their rectangles. A primary occurrence [37] of P for a split position $\ell \in \{1, 2, \ldots, m-1\}$ is a pair (u_i, ℓ) with $u_i \in \mathcal{U}_{\mathsf{expl}}$ such that (i) $\mathsf{reverse}(P[1..\ell])$ is a prefix of L_i and (ii) $P[\ell+1..m]$ is a prefix of R_i; that is, u_i is an explicit node where an occurrence of P crosses the first boundary between substrings derived from u_i's production rule. We write $pOcc(P, \ell) \subseteq \mathcal{U}_{\mathsf{expl}}$ for the set of such u_i at split position ℓ. Recall from Section 4.2 that u_i is mapped to the point p_i with x-coordinate L_i and y-coordinate R_i. The two prefix conditions constrain L_i and R_i, respectively, so $pOcc(P, \ell)$ coincides with the set of points of $\mathcal{P}$ in the axis-aligned rectangle $\mathsf{rect}_\ell = [x, x'] \times [y, y']$, where $x, x' \in \mathcal{X}$ are the smallest and largest elements whose x-coordinate has $\mathsf{reverse}(P[1..\ell])$ as a prefix, and $y, y' \in \mathcal{Y}$ are the analogous bounds for $P[\ell+1..m]$ (if any of the four does not exist, rect_ℓ is empty).

Reducing to ρ rectangles. Because the popped sequence is a core of P, only split positions at its run boundaries or at the end of its first nonterminal can yield a nonempty $pOcc(P, \ell)$, as formalized below.

▶ **Lemma 2** (Generalization of Lemma 7 in [37]). *Let Q be a core of a pattern P of length $m \geq 2$, and let $Q[1]$ be the first nonterminal of Q. Also, let ρ be the number of runs in the run-length encoding of Q. For each $i \in \{1, \ldots, \rho - 1\}$, let $\phi(i)$ denote the length of the string derived from the first i runs. Then, for any integer $\ell \in \{1, 2, \ldots, m-1\}$, if $\ell \notin \{|\mathsf{val}(Q[1])|\} \cup \{\phi(i) \mid i = 1, 2, \ldots, \rho - 1\}$, then $pOcc(P, \ell) = \emptyset$.*

Proof. See the full version [41]. ◀

Applying Lemma 2 to the popped sequence Q, this step constructs only the ρ rectangles $\mathsf{rect}_{|\mathsf{val}(Q[1])|}, \mathsf{rect}_{\phi(1)}, \mathsf{rect}_{\phi(2)}, \ldots, \mathsf{rect}_{\phi(\rho-1)}$.

5.3 Step (iii): Range Reporting for Primary Occurrences

We apply a range reporting query on Blelloch's data structure for $\mathcal{P}$ (Section 4.3) to each of the ρ rectangles; since $pOcc(P,\ell)$ coincides with the points in rect_ℓ, taking the union yields the complete set $\bigcup_{\ell=1}^{m-1} pOcc(P,\ell)$ of primary occurrences.

5.4 Step (iv): Converting Primary Occurrences to $Occ(T,P)$

For $u_i \in \mathcal{U}$, let $\mathsf{vOcc}(u_i) \subseteq \{1,2,\ldots,n\}$ be the starting positions in T of the occurrences of $\mathcal{L}_U(u_i)$ in the derivation tree (i.e., $j \in \mathsf{vOcc}(u_i)$ if and only if $\mathcal{L}_U(u_i)$ is embedded at position j). The following lemma converts primary occurrences into $Occ(T,P)$ using vOcc, distinguishing two cases by the production rule of u_i.

▶ **Lemma 3** ([37]). *Recall that L_i is the x-coordinate of u_i (defined in Section 4.2). For each $\ell \in \{1,2,\ldots,m-1\}$ and each $u_i \in pOcc(P,\ell)$, define $\mathcal{Z}_\ell(u_i)$ according to the production rule of u_i:*

- *if the rule has the form $X_i \to X_j X_k$ with two distinct nonterminals,*

$$\mathcal{Z}_\ell(u_i) = \{q + |L_i| - \ell \mid q \in \mathsf{vOcc}(u_i)\};$$

- *if the rule has the form $X_i \to (X_k)^d$,*

$$\mathcal{Z}_\ell(u_i) = \left\{ q + j \cdot |L_i| - \ell \;\middle|\; q \in \mathsf{vOcc}(u_i),\ j \in \{1,2,\ldots,\lfloor \tfrac{|\mathsf{val}(X_i)|-(m-\ell)}{|L_i|} \rfloor\} \right\}.$$

Then,

$$Occ(T,P) = \bigcup_{\ell=1}^{m-1} \bigcup_{u_i \in pOcc(P,\ell)} \mathcal{Z}_\ell(u_i).$$

Moreover, the sets $\mathcal{Z}_\ell(u_i)$ are pairwise disjoint over all pairs (u_i,ℓ) with $u_i \in pOcc(P,\ell)$.

By Lemma 3, taking the union $\bigcup_{\ell=1}^{m-1} \bigcup_{u_i \in pOcc(P,\ell)} \mathcal{Z}_\ell(u_i)$ yields $Occ(T,P)$, completing the locate query.

5.5 Implementation and Complexity Analysis

This subsection gives the full locate algorithm for $m \geq 1$, after first setting up the auxiliary queries it relies on.

5.5.1 Auxiliary queries on the DAG

For the locate algorithm we use two ingredients: Lemma 4 for traversing nodes and edges of the DAG, and random access, LCE, and reversed LCE queries on T for computing rectangle coordinates in phase (ii) of Section 5.5.2. Although inter-path edges are stored without labels (Section 4.3), each label can be reconstructed in $\mathcal{O}(1)$ time from the source's production rule (see [41] for details); each value in $\mathsf{vOcc}(u_i)$ then equals 1 plus the sum of edge labels along the corresponding root-to-u_i path. The following lemmas summarize the resulting queries.

▶ **Lemma 4.** *Using the doubly linked list for Π, we can support the following four operations on a given node $u_i \in \mathcal{U}$ lying on a path $\mathbb{P} \in \Pi$:*

(i) *return the following information in $\mathcal{O}(1)$ time: (A) the production rule $X_i \to \mathsf{expr}_i$ corresponding to u_i, (B) $|\mathsf{val}(X_i)|$, (C) the height of u_i, (D) $\mathsf{assign}(X_i)$, and (E) the children of u_i;*

(ii) *return the outgoing edges of u_i with edge labels in $\mathcal{O}(1)$ time per edge;*
(iii) *return the directed edges with edge labels in $\mathcal{E}_{\mathsf{expl}}(\mathbb{P})$ in $\mathcal{O}(1)$ time per edge;*
(iv) *return $\mathsf{vOcc}(u_i)$ in $\mathcal{O}(H|\mathsf{vOcc}(u_i)|)$ time if u_i is explicit.*

Proof. See the full version [41]. ◀

Random access, LCE, and reversed LCE queries. A random access query returns $T[i]$; an LCE (resp. reversed LCE) query returns the length of the longest common prefix (resp. suffix) of $T[i..n]$ and $T[j..n]$ (resp. $T[1..i]$ and $T[1..j]$); one argument may instead be a suffix (resp. prefix) of P once the popped sequence of P has been computed. Kempa and Kociumaka's algorithms (Theorems 5.24 and 5.25 in [25]) traverse the derivation tree of the restricted recompression RLSLP to answer these in $\mathcal{O}(H)$ time; Lemma 4 enables the same traversal in our representation, and a modification computes a substring of length ℓ starting at position i in $\mathcal{O}(H+\ell)$ time. Phase (ii) of Section 5.5.2 uses them to compare, via binary search, a suffix of P with the coordinate R_s of a candidate node u_s (or the reverse of a prefix of P with L_s) in $\mathcal{O}(H+\log M)$ time per comparison. See [41] for details of the three queries.

5.5.2 Full locate algorithm and complexity bound

Combining Steps (i)–(iv) with the auxiliary queries of Section 5.5.1 yields the following.

▶ **Theorem 5.** *Given a pattern P of length $m \geq 1$, a locate query returning $Occ(T,P)$ can be answered in expected*

$$\mathcal{O}\big(m + \log m(H+\log M)\log M + \mathsf{occ}(H+\log M/\log\log M)\big)$$

time using the dynamic RR-index.

Proof. If $m=1$, write $P=c$ with $c\in\Sigma$. In $\mathcal{O}(1)$ time, the global hash table tells us whether some $X_i\in\mathcal{V}$ satisfies $X_i\to c$ (and returns it if so); otherwise $Occ(T,P)=\emptyset$. The corresponding DAG node u_i is explicit, so Lemma 4(iv) enumerates $Occ(T,P)=\mathsf{vOcc}(u_i)$ in $\mathcal{O}(\mathsf{occ}\cdot H)$ time, which is absorbed into the stated bound.

For $m\geq 2$, we execute the four phases corresponding to Steps (i)–(iv).

Phase (i). Compute the popped sequence Q of P, whose run-length encoding has ρ runs, by the algorithm of [13] (see [41]) in expected $\mathcal{O}(m)$ time. If it fails, $Occ(T,P)=\emptyset$ and we return $\emptyset$.

Phase (ii). Compute the ρ rectangles $\mathsf{rect}_{|\mathsf{val}(Q[1])|}, \mathsf{rect}_{\phi(1)}, \ldots, \mathsf{rect}_{\phi(\rho-1)}$ by binary search on $\mathcal{X}$ and $\mathcal{Y}$ for each of the four coordinates. Each of the $\mathcal{O}(\log M)$ comparisons takes $\mathcal{O}(H+\log M)$ time (Section 5.5.1), so this phase runs in $\mathcal{O}(\rho(H+\log M)\log M)$ time with $\mathbb{E}[\rho]=\mathcal{O}(\log m)$.

Phase (iii). Issue ρ range reporting queries on $\mathcal{P}$ to obtain $\bigcup_{\ell=1}^{m-1} pOcc(P,\ell)$ in $\mathcal{O}(\rho\log M + \mathsf{occ}_{\mathsf{prim}}(\log M/\log\log M))$ time, where $\mathsf{occ}_{\mathsf{prim}}\leq\mathsf{occ}$ is the number of primary occurrences.

Phase (iv). For each (u_i,ℓ) with $u_i\in pOcc(P,\ell)$, compute $\mathsf{vOcc}(u_i)$ in $\mathcal{O}(H|\mathsf{vOcc}(u_i)|)$ time (Lemma 4(iv)) and return $\mathcal{Z}_\ell(u_i)$ as defined in Lemma 3. Since $\sum_\ell\sum_{u_i\in pOcc(P,\ell)}|\mathcal{Z}_\ell(u_i)|=\mathsf{occ}$, phase (iv) takes $\mathcal{O}(\mathsf{occ}\cdot H)$ time.

Summing the four phase bounds and absorbing the $\mathsf{occ}_{\mathsf{prim}}(\log M/\log\log M)$ term of phase (iii) into phase (iv) via $\mathsf{occ}_{\mathsf{prim}}\leq\mathsf{occ}$ gives the stated complexity. ◀

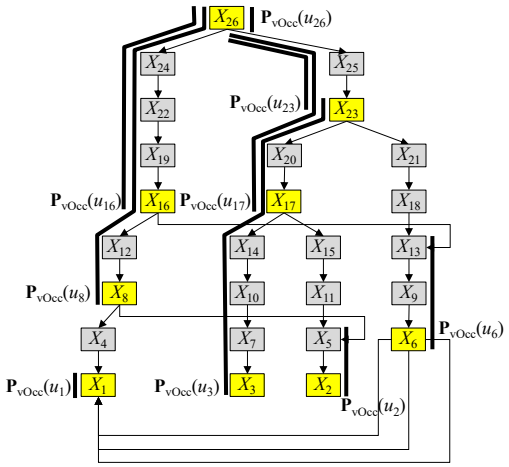

Figure 4 Each path $\mathbb{P}_{\mathsf{vOcc}}(u_i)$ on the DAG of Figure 2 with $\lceil \alpha + \log B \rceil = 6$: the bold line starting at X_i is $\mathbb{P}_{\mathsf{vOcc}}(u_i)$, and u_i is the explicit node for X_i.

5.6 Faster Locate via Ancestor-Path Caching

In the locate algorithm of Theorem 5, phase (iv) spends $\mathcal{O}(H)$ time per reported occurrence, so the term $\mathsf{occ} \cdot H$ dominates the running time whenever occ is large. We reduce this factor to $H/\log B$ by caching, at each explicit node, a short ancestor path in the DAG. This yields the main result of this section.

▶ **Theorem 6.** *Given a pattern P of length $m \geq 1$, a locate query returning $Occ(T, P)$ can be answered in expected*

$$\mathcal{O}\big(m + \log m \log^2 n + \mathsf{occ}(\log n / \log\log n)\big)$$

time using the dynamic RR-index augmented with the additional data structures described below. This bound holds under the assumption that $H = \mathcal{O}(\log n)$.

Section 5.6.1 proves Theorem 6 by replacing the algorithm for computing $\mathsf{vOcc}(u_i)$ (i.e., Lemma 4(iv)) with an improved algorithm that runs in $\mathcal{O}(|\mathsf{vOcc}(u_i)|\lceil H/\log B \rceil)$ time.

We also propose a complementary heuristic for phase (ii); it does not change the asymptotic complexity but is expected to reduce the running time in practice. See [41] for details.

Additional data structures. Fix tunable constants $\alpha, \beta = \mathcal{O}(1)$ controlling the length of a cached ancestor path and a stored coordinate prefix, respectively. For each path $\mathbb{P} \in \Pi$ with explicit endpoint $u_i \in \mathcal{U}_{\mathsf{expl}}$, we augment the corresponding list element with two $\mathcal{O}(B)$-bit fields: (i) the pair (u_j, W) encoding the longest path $\mathbb{P}_{\mathsf{vOcc}}(u_i)$ ending at u_i with $|\mathbb{P}_{\mathsf{vOcc}}(u_i)| \leq \lceil \alpha + \log B \rceil$ such that every node except its first node u_j has exactly one parent in $\mathfrak{D}$, where W is the sum of edge labels along the path (if u_i does not have exactly one parent, $\mathbb{P}_{\mathsf{vOcc}}(u_i) = (u_i)$ with $u_j = u_i$ and $W = 0$); this field is used in Section 5.6.1. (ii) the first $\lceil \beta B / \log \sigma \rceil$ characters of L_i and of R_i; this field is used in the phase-(ii) heuristic. Figure 4 illustrates $\mathbb{P}_{\mathsf{vOcc}}(u_i)$. The M list elements together add $\mathcal{O}(MB)$ bits, so the doubly linked list still fits in $\mathcal{O}(M(H + B))$ bits.

5.6.1 Speeding up computation of $\mathsf{vOcc}(u_i)$

The cached paths $\mathbb{P}_{\mathsf{vOcc}}(u_i)$ let us compute $\mathsf{vOcc}(u_i)$ via the following recurrence.

▶ **Lemma 7.** *For an explicit node $u_i \in \mathcal{U}_{\mathsf{expl}}$, let (u_j, W) be the encoding of $\mathbb{P}_{\mathsf{vOcc}}(u_i)$ and let $\mathbb{P}_j \in \Pi$ be the path containing u_j. Then*

$$\mathsf{vOcc}(u_i) = \begin{cases} \bigcup_{e \in \mathcal{E}_{\mathsf{expl}}(\mathbb{P}_j)} \{W + \mathcal{L}_E(e) + q \mid q \in \mathsf{vOcc}(\mathrm{src}(e))\} & \text{if } \mathcal{E}_{\mathsf{expl}}(\mathbb{P}_j) \neq \emptyset, \\ \{W+1\} & \text{otherwise.} \end{cases}$$

Proof. See the full version [41]. ◀

Guided by Lemma 7, the algorithm reads (u_j, W) from the list element, follows the pointer at u_j to the element representing $\mathbb{P}_j$, recovers $\mathcal{E}_{\mathsf{expl}}(\mathbb{P}_j)$ with its edge labels via Lemma 4(iii), and either returns $\{W+1\}$ (if empty) or recursively computes $\mathsf{vOcc}(\mathrm{src}(e))$ and returns the union in the recurrence. The total number Z of recursive calls satisfies $Z = \mathcal{O}(|\mathsf{vOcc}(u_i)|\lceil H/\log B\rceil)$, since they decompose every root-to-u_i path in the DAG into cached paths, inter-path edges, and zero-label paths. See [41] for details. Substituting into Theorem 5 with $\log M = \mathcal{O}(\log n)$ and $B = \mathcal{O}(\log n)$ yields Theorem 6 under $H = \mathcal{O}(\log n)$.

6 Update Operations

6.1 Insertion Operation

Given a string $P' \in \Sigma^+$ of length $m' = \mathcal{O}(n)$ and a position $s \in \{1, 2, \ldots, n+1\}$, the insertion operation produces T' by inserting P' into T at position s and updates the dynamic RR-index so that its RLSLP derives T'. For simplicity, we focus on $2 \le s \le n$, so that T' is the concatenation of $T[1..s-1]$, P', and $T[s..n]$; the boundary cases $s = 1$ and $s = n+1$ are analogous. Following the standard strategy of dynamic grammar-compressed data structures [43, 37, 40, 36], the update proceeds in three phases (see the full version [41] for details).

Phase 1: constructing the derivation tree of $\mathcal{G}^{R'}$. The first phase constructs the derivation tree of an RLSLP $\mathcal{G}^{R'} = (\mathcal{V}', \Sigma', \mathcal{D}', E')$ deriving T' by restricted recompression, computing $(H'+1)$ sequences $S'^0, S'^1, \ldots, S'^{H'}$ bottom-up; production rules are shared with $\mathcal{G}^R$ whenever the right-hand sides agree. For each nonterminal in $\mathcal{V}' \setminus \mathcal{V}$, we insert the corresponding node into the DAG and update the dynamic data structures. A naive construction takes $\Omega(n + m')$ time since the tree has $n + m'$ leaves. We reduce this cost using the popped sequences Q_L and Q_R of $T[1..s-1]$ and $T[s..n]$: since each popped sequence is a core of its source, Q_L and Q_R are embedded in the derivation tree of $\mathcal{G}^{R'}$ at positions 1 and $s + m'$ as subtrees that need not be reconstructed. Concretely, (i) compute Q_L and Q_R by the algorithm of [13] in $\mathcal{O}(H)$ time; (ii) build $S'^0, \ldots, S'^{H'}$ in $\mathcal{O}(|\mathcal{T}_{\mathsf{diff}}|)$ time, where $\mathcal{T}_{\mathsf{diff}}$ collects (A) the ancestors of the embedded subtree roots, (B) the m' leaves covering $T'[s..s+m'-1]$, and (C) their ancestors. Figure 5 illustrates the embeddings. The phase runs in expected amortized $\mathcal{O}(|\mathcal{T}_{\mathsf{diff}}| + |\mathcal{I}_{\mathsf{expl}}| \max\{H, H', \log M\} \log M + |\mathcal{I}_{\mathsf{impl}}|)$ time, where $\mathcal{I}_{\mathsf{expl}}$ (resp. $\mathcal{I}_{\mathsf{impl}}$) is the set of explicit (resp. implicit) nodes inserted into the DAG.

Phase 2: removing obsolete nonterminals. For each nonterminal in $\mathcal{V} \setminus \mathcal{V}'$, we remove the corresponding node from the DAG and update the dynamic data structures. This phase takes expected amortized $\mathcal{O}(|\mathcal{R}_{\mathsf{expl}}| \log M + |\mathcal{R}_{\mathsf{impl}}|)$ time. Here, $\mathcal{R}_{\mathsf{expl}}$ (resp. $\mathcal{R}_{\mathsf{impl}}$) is the set of explicit (resp. implicit) nodes whose nonterminals do not appear in the derivation tree of $\mathcal{G}^{R'}$.

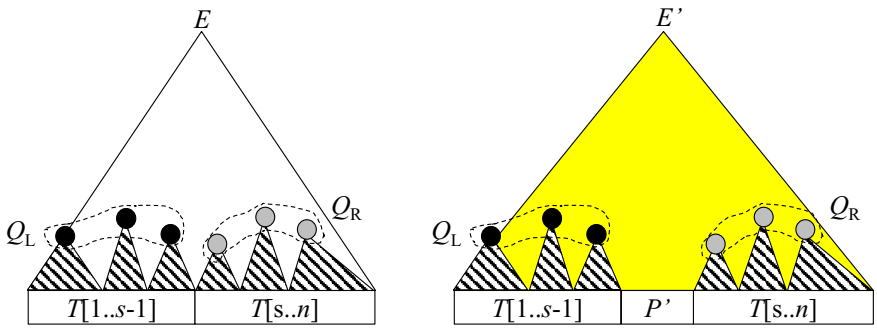

Figure 5 Embeddings of Q_L and Q_R in the derivation trees of $\mathcal{G}^R$ and $\mathcal{G}^{R'}$ (start symbols E and E'). The black/gray circles mark the nodes for Q_L/Q_R, hatched regions are the subtrees rooted there, and the yellow region is the part of the derivation tree of $\mathcal{G}^{R'}$ to be constructed.

Phase 3: updating ancestor-path caches and coordinate prefixes. For each explicit node in $\mathcal{I}_{\mathsf{expl}} \cup \mathcal{U}_{\mathsf{change}}$, we update the additional information stored in the corresponding element of the doubly linked list for Π. Here, $\mathcal{U}_{\mathsf{change}} \subseteq \mathcal{U}_{\mathsf{expl}}$ is the set of explicit nodes that have an ancestor in $\mathcal{I}_{\mathsf{expl}} \cup \mathcal{I}_{\mathsf{impl}} \cup \mathcal{R}_{\mathsf{expl}} \cup \mathcal{R}_{\mathsf{impl}}$ within distance $\lceil \alpha + \log B \rceil$ in the DAG. This phase takes $\mathcal{O}(W_{\max} + (|\mathcal{U}_{\mathsf{change}}| + |\mathcal{I}_{\mathsf{expl}}|)(H' + B))$ time, where $W_{\max} = \max\{|\mathcal{I}_{\mathsf{expl}}|, |\mathcal{I}_{\mathsf{impl}}|, |\mathcal{R}_{\mathsf{expl}}|, |\mathcal{R}_{\mathsf{impl}}|, |\mathcal{U}_{\mathsf{change}}|\}$.

Correctness and running time. To ensure that $H' = \mathcal{O}(\log n)$, we declare the update to fail if $H' > \lceil 2(w+1)\log_{8/7}(4(n+m')) + 2 \rceil$. By Lemma 1, the update succeeds with high probability for a sufficiently large constant w. Since $H, H', B, \log M = \mathcal{O}(\log(n+m'))$ and $m' = \mathcal{O}(n)$, summing the three phases yields

$$\mathcal{O}(|\mathcal{T}_{\mathsf{diff}}| + (|\mathcal{I}_{\mathsf{expl}}| + |\mathcal{R}_{\mathsf{expl}}|)\log^2 n + W_{\max} + |\mathcal{U}_{\mathsf{change}}| \log n).$$

The following lemma bounds $|\mathcal{T}_{\mathsf{diff}}|$, $|\mathcal{I}_{\mathsf{expl}}|$, $|\mathcal{I}_{\mathsf{impl}}|$, $|\mathcal{R}_{\mathsf{expl}}|$, $|\mathcal{R}_{\mathsf{impl}}|$, and $|\mathcal{U}_{\mathsf{change}}|$.

▶ **Lemma 8.** *(i)* $|\mathcal{T}_{\mathsf{diff}}| = \mathcal{O}((m'+H)H')$, *(ii)* $|\mathcal{I}_{\mathsf{impl}}| = \mathcal{O}((m'+H)H')$, *(iii)* $|\mathcal{R}_{\mathsf{impl}}| = \mathcal{O}(H^2)$, *(iv)* $|\mathcal{I}_{\mathsf{expl}}| = \mathcal{O}(m'+H)$, *(v)* $|\mathcal{R}_{\mathsf{expl}}| = \mathcal{O}(H)$, *and (vi)* $|\mathcal{U}_{\mathsf{change}}| = \mathcal{O}(HB)$.

Proof. See the full version [41]. ◀

Therefore, the total expected amortized running time is $\mathcal{O}(m' \log^2 n + \log^3 n)$.

6.2 Deletion Operation

Given a position $s \in \{1, 2, \ldots, n\}$ and a length m' with $s + m' - 1 \leq n$, the deletion operation produces T' by removing the substring $T[s..s+m'-1]$ from T and updates the dynamic RR-index so that its RLSLP derives T'. It runs in expected amortized $\mathcal{O}(m' \log^2 n + \log^3 n)$ time by a similar approach. See [41] for details.

7 Experiments

Setup. We evaluated the dynamic RR-index on locate queries and update operations over eleven highly repetitive strings: (i) nine strings from the Pizza&Chili repetitive corpus [44]; (ii) a 37 GB string (enwiki) of English Wikipedia articles with complete edit history [49]; and (iii) a 59 GB string (chr19) obtained by concatenating chromosome 19 from 1,000 human genomes in the 1000 Genomes Project [48]. Per-dataset statistics (σ, n, δ, the explicit-node count M,

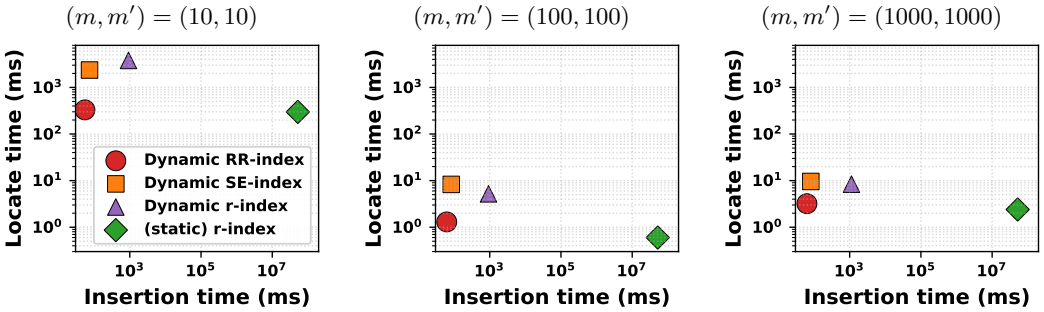

Figure 6 Insertion–locate trade-off on chr19 for $(m, m') \in \{(10, 10), (100, 100), (1000, 1000)\}$. Each panel plots, for each method, the mean insertion time (x-axis) against the mean locate time (y-axis), both on a log scale; lower-left is better.

the height H of the derivation tree, and the average occurrence counts $\mathsf{occ}_{10}, \mathsf{occ}_{100}, \mathsf{occ}_{1000}$ for 1,000 patterns of lengths 10, 100, and 1000) appear in Appendix B.1; for chr19, $\sigma = 4$, $n \approx 5.9 \times 10^{10}$, $\delta \approx 2.7 \times 10^6$, $M \approx 2.7 \times 10^7$, $H = 372$, $\mathsf{occ}_{10} = 881{,}774$, $\mathsf{occ}_{100} = 940$, and $\mathsf{occ}_{1000} = 504$.

We compared the dynamic RR-index with both dynamic and static compressed indexes: the dynamic SE-index [37], the dynamic r-index [42], and the static r-index [16]. We implemented the dynamic RR-index and the dynamic SE-index from scratch in C++ (`https://github.com/TNishimoto/dynamic-rlslp-index`), and used the public implementations of the r-index (`https://github.com/nicolaprezza/r-index`) and the dynamic r-index (`https://github.com/TNishimoto/dynamic_r_index`). All experiments ran on a single core of a 48-core Intel Xeon Gold 6126 CPU (2.6 GHz) with 2 TB of RAM, under 64-bit CentOS 7.9.

Implementation details of the dynamic RR-index. The dynamic RR-index has three parameters: w (Section 3) and α, β (Section 5.6). For performance reasons, we fix the integer quantity that each parameter controls, as follows. We store node heights in 16-bit integers, i.e., we require the height bound to satisfy $\lceil 2(w+1)\log_{8/7}(4n) + 2 \rceil \leq 65535\ (= 2^{16} - 1)$, which fixes w. We set the cached ancestor-path length $\lceil \alpha + \log B \rceil$ to 6 when $n \leq 10^9$, and to 8 otherwise. We set the stored coordinate-prefix length $\lceil \beta B / \log \sigma \rceil$ to $\lfloor 64 / \log \sigma \rfloor$, so that the first $\lfloor 64 / \log \sigma \rfloor$ characters fit in a single 64-bit integer.

Since no public implementation of Blelloch's data structure (Section 4.3) was available, we used a dynamic wavelet-matrix-based data structure [9] for two-dimensional range reporting. It uses $\mathcal{O}(MB)$ bits of space, supports range reporting queries in $\mathcal{O}((1 + rocc)\log^2 M)$ time, where $rocc$ is the number of reported points, and supports point insertions and deletions in amortized $\mathcal{O}(\log^2 M)$ time. Therefore, this substitution increases the time complexities of locate queries and update operations by logarithmic factors.

Results for chr19. For chr19 we measured update time, locate time, working space, and construction time. We inserted, then deleted, 1,000 random substrings of length $m' \in \{1, 10, 100, 1000\}$ at random positions in T, and ran 1,000 locate queries with random patterns of length $m \in \{10, 100, 1000\}$. In the trade-off figures below, the (static) r-index does not support updates, so its x-coordinate is the full reconstruction time ($\approx 5.2 \times 10^7$ ms ≈ 14.5 h on chr19; $\approx 3.9 \times 10^7$ ms ≈ 10.8 h on enwiki).

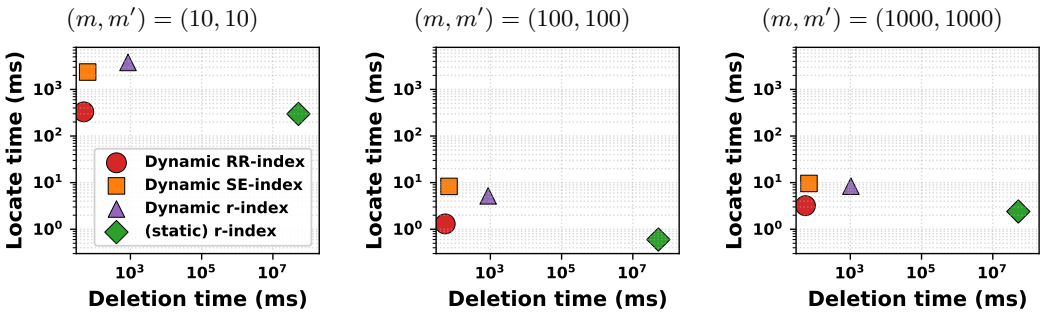

Figure 7 Deletion–locate trade-off on chr19. Axes and conventions are as in Figure 6.

The dynamic RR-index appears in the lower-left region of every panel of Figures 6 and 7: it dominates the dynamic SE-index and dynamic r-index on both axes for every (m, m'), running roughly 1.3× faster than the dynamic SE-index and 15–19× faster than the dynamic r-index on updates, and at least 2.6× faster than both on locate (reaching ≈ 7× vs. the dynamic SE-index and ≈ 11× vs. the dynamic r-index at $m = 10$). On locate, it is competitive with the (static) r-index – within 1.1–1.3× at $m \in \{10, 1000\}$ and within 2.2× at $m = 100$ – while additionally supporting updates that the r-index cannot. On the 59 GB chr19 ($\delta \approx 2.7 \times 10^6$), the dynamic RR-index used ≈ 3.7 GB of working space (Figure 10, center) – about 1/16 of the raw text size – and finished construction in ≈ 9 h (Figure 10, right), within commodity-server budgets. Full numbers appear in Appendix B.2.

Results for enwiki. The same trade-off pattern holds on enwiki (Figures 8 and 9): the dynamic RR-index dominates the other two dynamic indexes, running roughly 2.1× faster than the dynamic SE-index and 16× faster than the dynamic r-index on updates, and is competitive with the (static) r-index on locate – matching it for short patterns and within ≈ 2.6× for medium patterns. On the 37 GB enwiki ($\delta \approx 7.3 \times 10^6$), the dynamic RR-index used ≈ 7.3 GB of working space (Figure 10, center) and finished construction in ≈ 8 h (Figure 10, right), again within commodity-server budgets; the ≈ 2× memory usage for a δ value approximately 2.7× larger than that of chr19 is consistent with linear-in-δ scaling. Full numbers appear in Appendix B.2.

Results for the Pizza&Chili corpus. Across most settings in the Pizza&Chili corpus, the dynamic RR-index was the fastest or competitive with the fastest dynamic index on update operations, reaching a speedup of ≈ 77× over the dynamic r-index on coreutils at $m' = 1000$ when insertions and deletions are considered together. For locate queries, the dynamic RR-index was usually the fastest among the dynamic indexes, although there were a few exceptions. Compared with the static r-index, it was within 2.5× for long patterns, but up to about 8.6× slower for short patterns on world leaders. Its working space scaled with δ, from 19 MiB ($\delta \approx 1.6 \times 10^4$, einstein.de.txt) to 886 MiB ($\delta \approx 1.4 \times 10^6$, para); construction finished within 7 minutes on every Pizza&Chili string.

Summary. The experiments show that the dynamic RR-index (i) scales with δ in memory, (ii) outperforms the other dynamic compressed indexes on both updates and locate in most settings, and (iii) is competitive with the (static) r-index on locate for short and long patterns on large datasets.

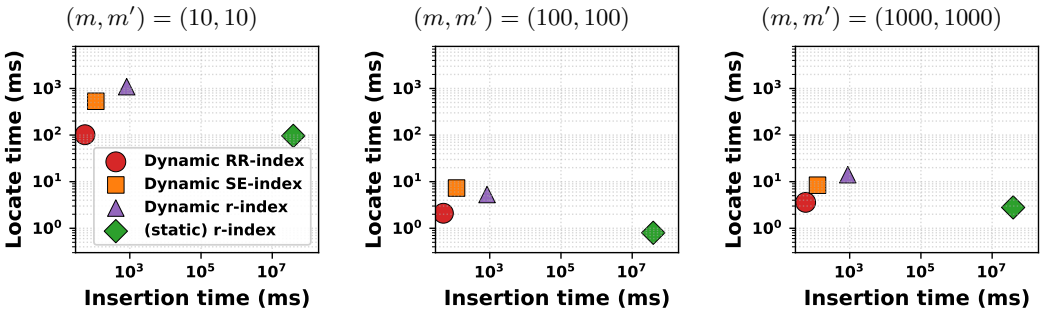

Figure 8 Insertion–locate trade-off on enwiki. Axes and conventions are as in Figure 6.

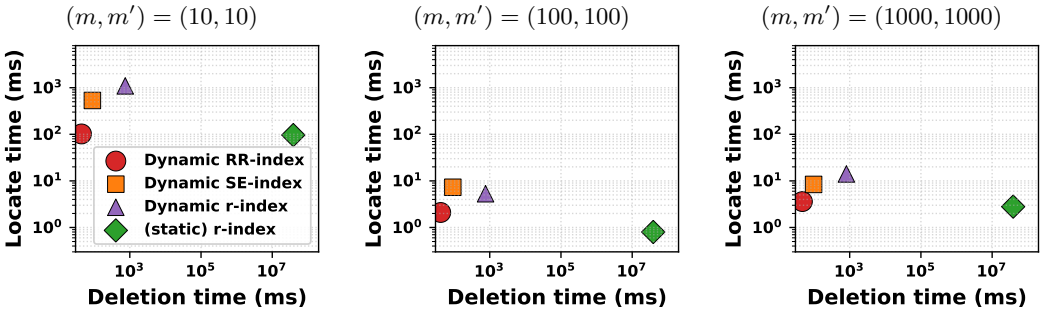

Figure 9 Deletion–locate trade-off on enwiki. Axes and conventions are as in Figure 6.

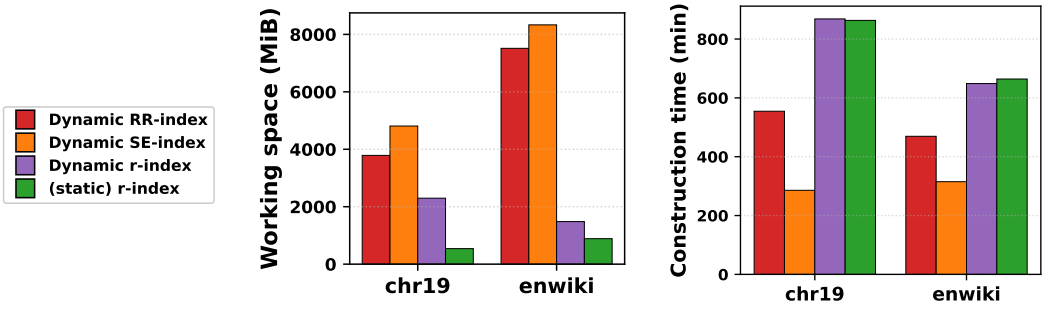

Figure 10 Working space during locate queries (center) and construction time (right) on chr19 and enwiki.

References

1 Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 819–828, 2000. URL: https://dl.acm.org/doi/10.5555/338219.338645.

2 Amos Beimel, Haim Kaplan, Yishay Mansour, Kobbi Nissim, Thatchaphol Saranurak, and Uri Stemmer. Dynamic algorithms against an adaptive adversary: generic constructions and lower bounds. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 1671–1684, 2022. doi:10.1145/3519935.3520064.

3 Djamal Belazzougui, Manuel Cáceres, Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Gonzalo Navarro, Alberto Ordóñez Pereira, Simon J. Puglisi, and Yasuo Tabei. Block trees. *Journal of Computer and System Sciences*, 117:1–22, 2021. doi:10.1016/j.jcss.2020.11.002.

4 Nico Bertram, Johannes Fischer, and Lukas Nalbach. Move-r: Optimizing the r-index. In *Proceedings of 22nd International Symposium on Experimental Algorithms (SEA)*, pages 1:1–1:19, 2024. doi:10.4230/LIPIcs.SEA.2024.1.

5 Guy E. Blelloch. Space-efficient dynamic orthogonal point location, segment intersection, and range reporting. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2008. URL: https://dl.acm.org/doi/10.5555/1347082.1347180.

6 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report SRC-RR-124, Digital Equipment Corporation, Systems Research Center, 1994.

7 Anders Roy Christiansen and Mikko Berggren Ettienne. Compressed indexing with signature grammars. In *Proceedings of the 13th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 331–345, 2018. doi:10.1007/978-3-319-77404-6_25.

8 Anders Roy Christiansen, Mikko Berggren Ettienne, Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Optimal-time dictionary-compressed indexes. *ACM Transactions on Algorithms*, 17:8:1–8:39, 2021. doi:10.1145/3426473.

9 Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez Pereira. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015. doi:10.1016/J.IS.2014.06.002.

10 Paul F. Dietz. Optimal algorithms for list indexing and subset rank. In *Proceedings of the Workshop on Algorithms and Data Structures (WADS)*, pages 39–46, 1989. doi:10.1007/3-540-51542-9_5.

11 Paul F. Dietz and Daniel Dominic Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC)*, pages 365–372, 1987. doi:10.1145/28395.28434.

12 Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23:738–761, 1994. doi:10.1137/S0097539791194094.

13 Anouk Duyster and Tomasz Kociumaka. Logarithmic-time internal pattern matching queries in compressed and dynamic texts. *Theory of Computing Systems*, 70(1):11, 2026. doi:10.1007/s00224-026-10266-x.

14 Andrzej Ehrenfeucht, Ross M. McConnell, and Sung-Whan Woo. Contracted suffix trees: A simple and dynamic text indexing data structure. In *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 41–53, 2009. doi:10.1007/978-3-642-02441-2_5.

15 Paolo Ferragina and Giovanni Manzini. On compressing the textual web. In *Proceedings of the 3rd Web Search and Data Mining (WSDM)*, pages 391–400, 2010. doi:10.1145/1718487.1718536.

16 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, 67, 2020. doi:10.1145/3375890.

17 Pawel Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Lacki, and Piotr Sankowski. Optimal dynamic strings. *CoRR*, abs/1511.02612, 2015. doi:10.48550/arXiv.1511.02612.

18 Pawel Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Lacki, and Piotr Sankowski. Optimal dynamic strings. In *Proceedings of the 2018 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1509–1528, 2018. doi:10.1137/1.9781611975031.99.

19 Torben Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 366–398, 1998. doi:10.1007/BFb0028575.

20 Tomohiro I. Longest common extensions with recompression. In *Proceedings of the 28th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 18:1–18:15, 2017. doi:10.4230/LIPIcs.CPM.2017.18.

21 Artur Jez. Faster fully compressed pattern matching by recompression. *ACM Transactions on Algorithms*, 11:20:1–20:43, 2015. doi:10.1145/2631920.

22 Marek Karpinski, Wojciech Rytter, and Ayumi Shinohara. An efficient pattern-matching algorithm for strings with short descriptions. *Nordic Journal of Computing*, 4:172–186, 1997.

23 Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows-Wheeler transform conjecture. In *Proceedings of the 61st IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1002–1013, 2020. doi:10.1109/FOCS46700.2020.00097.

24 Dominik Kempa and Tomasz Kociumaka. Dynamic suffix array with polylogarithmic queries and updates. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 1657–1670, 2022. doi:10.1145/3519935.3520061.

25 Dominik Kempa and Tomasz Kociumaka. Collapsing the hierarchy of compressed data structures: Suffix arrays in optimal compressed space. *CoRR*, abs/2308.03635, 2023. doi:10.48550/arXiv.2308.03635.

26 Dominik Kempa and Tomasz Kociumaka. Collapsing the hierarchy of compressed data structures: Suffix arrays in optimal compressed space. In *Proceedings of the 64th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1877–1886, 2023. doi:10.1109/FOCS57990.2023.00114.

27 Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: String attractors. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 827–840, 2018. doi:10.1145/3188745.3188814.

28 Tomasz Kociumaka, Gonzalo Navarro, and Francisco Olivares. Near-optimal search time in δ-optimal space, and vice versa. *Algorithmica*, 86(4):1031–1056, 2024. doi:10.1007/s00453-023-01186-0.

29 Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Toward a definitive compressibility measure for repetitive sequences. *IEEE Transactions on Information Theory*, 69:2074–2092, 2023. doi:10.1109/TIT.2022.3224382.

30 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Optimal data structure for internal pattern matching queries in a text and applications. *CoRR*, abs/1311.6235, 2013. doi:10.48550/arXiv.1311.6235.

31 Dmitry Kosolobov. Compressed index with construction in compressed space. *CoRR*, abs/2602.13735, 2026. doi:10.48550/arXiv.2602.13735.

32 Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976. doi:10.1109/TIT.1976.1055501.

33 Martine Léonard, Laurent Mouchard, and Mikaël Salson. On the number of elements to reorder when updating a suffix array. *Journal of Discrete Algorithms*, 11:87–99, 2012. doi:10.1016/j.jda.2011.01.002.

34 Shirou Maruyama, Masaya Nakahara, Naoya Kishiue, and Hiroshi Sakamoto. ESP-index: A compressed index based on edit-sensitive parsing. *Journal of Discrete Algorithms*, 18:100–112, 2013. doi:10.1016/j.jda.2012.07.009.

35 Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014. doi:10.1016/j.jda.2013.07.004.

36 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. In *Proceedings of the 41st International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 72:1–72:14, 2016. doi:10.4230/LIPIcs.MFCS.2016.72.

37 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Dynamic index and LZ factorization in compressed space. *Discrete Applied Mathematics*, 274:116–129, 2020. doi:10.1016/j.dam.2019.01.014.

38 Takaaki Nishimoto and Yasuo Tabei. Dynamic RLSLP Index. Software, swhId: swh:1:dir:e7a517c0fd034955d844beea282fceefedac7b63 (visited on 2026-08-13). URL: https://github.com/TNishimoto/dynamic-rlslp-index, doi:10.4230/artifacts.27657.

39 Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on BWT-runs compressed indexes. In *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 101:1–101:15, 2021. doi:10.4230/LIPIcs.ICALP.2021.101.

40 Takaaki Nishimoto and Yasuo Tabei. Dynamic suffix array in optimal compressed space. *CoRR*, abs/2404.07510, 2024. doi:10.48550/arXiv.2404.07510.

41 Takaaki Nishimoto and Yasuo Tabei. Dynamic grammar-compressed self-index in δ-optimal space. *CoRR*, abs/2604.24080, 2026. doi:10.48550/arXiv.2604.24080.

42 Takaaki Nishimoto and Yasuo Tabei. Dynamic r-index: An updatable self-index for highly repetitive strings. In *Proceedings of Data Compression Conference (DCC)*, pages 401–410, 2026. doi:10.1109/DCC66757.2026.00048.

43 Takaaki Nishimoto, Yoshimasa Takabatake, and Yasuo Tabei. A compressed dynamic self-index for highly repetitive text collections. *Information and Computation*, 273:104518, 2020. doi:10.1016/j.ic.2020.104518.

44 Pizza&Chili repetitive corpus. http://pizzachili.dcc.uchile.cl/repcorpus.html.

45 Molly Przeworski, Richard R. Hudson, and Anna Di Rienzo. Adjusting the focus on human variation. *Trends in Genetics*, 16:296–302, 2000.

46 Sofya Raskhodnikova, Dana Ron, Ronitt Rubinfeld, and Adam D. Smith. Sublinear algorithms for approximating string compressibility. *Algorithmica*, 65:685–709, 2013. doi:10.1007/s00453-012-9618-6.

47 Mikaël Salson, Thierry Lecroq, Martine Léonard, and Laurent Mouchard. Dynamic extended suffix arrays. *Journal of Discrete Algorithms*, 8:241–257, 2010. doi:10.1016/j.jda.2009.02.007.

48 The 1000 Genomes Project Consortium. An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491:56–65, 2012.

49 Enwiki dump progress on 20241201: All pages with complete edit history (xml-p1p812). https://dumps.wikimedia.org/enwiki/.

A Summary Comparison of Dynamic and Static Self-Indexes

Table 1 summarizes state-of-the-art dynamic and static self-indexes supporting locate queries.

Table 1 Summary of state-of-the-art dynamic and static self-indexes supporting locate queries. Here, n is the length of the input string T, m is the length of the pattern P, δ is the substring complexity of T, occ is the number of occurrences of P in T, and m' is the length of an inserted string or a deleted substring. Moreover, $\mathsf{occ}_c \geq \mathsf{occ}$ is the number of candidate occurrences returned by the ESP-index [34], $\sigma = n^{\mathcal{O}(1)}$ is the alphabet size, $\epsilon > 0$ is an arbitrary constant, and $L_{\max} \leq n$ is the maximum value in the LCP array of T. Let r be the number of runs in the BWT [6] of T, where $r = \mathcal{O}(\delta \log \delta \max\{1, \log(n/(\delta \log \delta))\})$ [23]. Let z be the number of factors in the LZ77 factorization [32] of T, where $z = \mathcal{O}(\delta \log \frac{n \log \sigma}{\delta \log n})$ [29]. Let g be the size of a compressed grammar deriving T, where $g = \mathcal{O}(z \log n \log^* n)$ [37]. Let γ be the size of the smallest string attractor [27] for T, where $\gamma = \mathcal{O}(\delta \log \frac{n \log \sigma}{\delta \log n})$ [29]. Also, $q \geq 1$ is a user-defined parameter, and g' is the total size of the q-truncated suffix tree of T and a compressed grammar deriving T, where $g' = \mathcal{O}(z(q^2 + \log n \log^* n))$ [43]. We assume a machine word size $B = \Theta(\log n)$ and $m' = \mathcal{O}(n)$. We exclude dynamic self-indexes that use $\Omega(n \log n)$ bits, such as [1, 14], with the exception of [17]. Furthermore, we exclude data structures that do not explicitly support locate queries, even when locate queries can be answered by combining other supported queries, because the resulting locate-query time is slower than that of standard self-indexes (e.g., [24, 26, 40]).

Method	Format	Type	Space (bits)
Dynamic FM-index [47, 33]	BWT	Dynamic	$\mathcal{O}(n \log \sigma)$
Dynamic SE-index [37]	Grammar	Dynamic	$\mathcal{O}(g \log n)$
TST-index-d [43]	Grammar	Dynamic	$\mathcal{O}(g' \log n)$
Gawrychowski et al. [18, 17]	Grammar	Dynamic	$\Omega(n \log n)$
Dynamic r-index [42]	RLBWT	Dynamic	$\mathcal{O}(r \log n)$
r-index [16]	RLBWT	Static	$\mathcal{O}(r \log n)$
OptBWTR [39]	RLBWT	Static	$\mathcal{O}(r \log n)$
Christiansen et al. [8]	Grammar	Static	$\mathcal{O}(\gamma \log(n/\gamma) \log n)$
Kociumaka et al. [28]	Grammar	Static	$\mathcal{O}(\delta \log \frac{n \log \sigma}{\delta \log n} \log n)$
Dynamic RR-index (this paper)	Grammar	Dynamic	expected $\mathcal{O}(\delta \log \frac{n \log \sigma}{\delta \log n} \log n)$

Method	Locate time	Update time
Dynamic FM-index [47, 33]	$\mathcal{O}((m + \mathsf{occ}) \log^{2+\epsilon} n)$	$\mathcal{O}((m' + L_{\max}) \log n)$
Dynamic SE-index [37]	$\mathcal{O}(m(\log \log n)^2 + \log m(\log n \log^* n)^2 + \mathsf{occ} \log n)$	amortized $\mathcal{O}(m'(\log n \log^* n)^2 + \log n(\log n \log^* n)^2)$
TST-index-d [43]	$\mathcal{O}(m(\log \log \sigma)^2 + \mathsf{occ} \log n)$ $(m \leq q)$	$\mathcal{O}((\log \log n)^2(m'q + q^2 + \log n \log^* n))$
	$\mathcal{O}(m(\log \log n)^2 + \mathsf{occ}_c \log m \log n \log^* n)$ $(m > q)$	
Gawrychowski et al. [17]	$\mathcal{O}(m + \log^2 n + \mathsf{occ} \log n)$	$\mathcal{O}(m' \log n + \log^2 n)$
Dynamic r-index [42]	$\mathcal{O}((m + \mathsf{occ}) \log n)$	$\mathcal{O}((m' + L_{\max}) \log n)$
r-index [16]	$\mathcal{O}(m \log \log \sigma + \mathsf{occ} \log \log n)$	Unsupported
OptBWTR [39]	$\mathcal{O}(m \log \log \sigma + \mathsf{occ})$	Unsupported
Christiansen et al. [8]	$\mathcal{O}(m + (1 + \mathsf{occ}) \log^{\epsilon} n)$	Unsupported
Kociumaka et al. [28]	$\mathcal{O}(m + (1 + \mathsf{occ}) \log^{\epsilon} n)$	Unsupported
Dynamic RR-index (this paper)	expected $\mathcal{O}(m + \log m \log^2 n + \mathsf{occ}(\log n / \log \log n))$	expected amortized $\mathcal{O}(m' \log^2 n + \log^3 n)$

B Details of Experiments

B.1 Dataset Statistics

Table 2 reports the relevant statistics for every dataset.

Table 2 Per-dataset statistics, summarized in Section 7 for chr19. σ is the alphabet size of the string T; n is the length of T; δ is the substring complexity of T; M is the number of explicit nodes in the DAG representing the derivation tree built by restricted recompression; H is the height of the derivation tree. occ_{10}, occ_{100}, and occ_{1000} denote the average number of occurrences in T of 1,000 patterns of lengths 10, 100, and 1000, respectively, used for the locate queries.

Data	σ	n [10^3]	δ [10^3]	M [10^3]	H	occ_{10}	occ_{100}	occ_{1000}
cere	5	461,287	1,003	4,874	304	1,046	29	6
coreutils	236	205,282	636	3,828	292	7,308	154	16
einstein.de.txt	117	92,758	16	122	272	5,983	615	130
einstein.en.txt	139	467,627	42	320	300	21,097	2,547	604
Escherichia Coli	15	112,690	1,338	4,897	282	214	7	2
influenza	15	154,809	282	2,576	286	2,993	307	14
kernel	160	257,962	406	2,069	302	2,857	53	26
para	5	429,266	1,369	6,333	298	916	20	10
world leaders	89	46,968	69	580	260	24,326	1,217	10
enwiki	206	37,227,587	7,306	52,806	366	525,289	1,672	427
chr19	4	59,125,169	2,715	27,336	372	881,774	940	504

B.2 Results for Other Datasets

Results for update operations. Table 3 reports the average time and standard deviation of substring insertions on all datasets. See the full version [41] for the corresponding results for substring deletions. For update operations, the dynamic RR-index was the fastest or competitive with the fastest dynamic index in most settings. On enwiki with $m' = 1000$, it was approximately 2.1× faster than the dynamic SE-index and 16× faster than the dynamic r-index. On the Pizza&Chili repetitive corpus with $m' = 1000$, the best-case speedups were approximately 2.3× over the dynamic SE-index (para) and 77× over the dynamic r-index (coreutils), with insertions and deletions considered together.

Results for locate queries. Table 4 reports the average time and standard deviation for 1,000 locate queries with short, medium, and long patterns on all datasets. On enwiki, the dynamic RR-index matched the r-index for short patterns and was only about 2.6× slower for medium patterns; against the other dynamic indexes, it was at least 5× faster for short patterns.

On most datasets and pattern lengths, the r-index was the fastest, and the dynamic RR-index was usually the fastest among the dynamic indexes, with a few exceptions: the dynamic SE-index was faster on world leaders for short patterns, and the dynamic r-index was faster on influenza and world leaders for medium patterns. For long patterns, however, it was at most about 2.5× slower than the r-index. In the best case, the dynamic RR-index was at least 3× faster than both other dynamic indexes for short patterns, at least 2× for medium patterns, and at least 2.6× for long patterns.

Results for memory consumption. Table 5 reports the working space used during locate queries on all datasets. Among the three dynamic indexes, the dynamic r-index was the most space-efficient, followed by the dynamic RR-index. On enwiki, the dynamic RR-index used approximately 5× the memory of the dynamic r-index. On the Pizza&Chili repetitive corpus, this ratio ranged from 1.4× (einstein.en.txt) to 4.6× (influenza).

Table 5 Working space during locate queries for all datasets. The smallest working space among the dynamic methods in each row is shown in bold.

	Working Space (MiB)			
Data	Dynamic RR-index	Dynamic SE-index	Dynamic r-index	r-index
cere	685	826	**260**	110
coreutils	522	602	**216**	48
einstein.de.txt	19	22	**12**	2
einstein.en.txt	53	54	**38**	4
Escherichia Coli	691	781	**448**	127
influenza	354	420	**77**	29
kernel	283	324	**137**	29
para	886	1,040	**473**	145
world leaders	84	102	**42**	6
enwiki	7,515	8,332	**1,482**	887
chr19	3,787	4,810	**2,297**	541

Results for construction time. Table 6 reports the construction times of the dynamic RR-index and the other three methods on all datasets. On enwiki, all four methods finished within about 11 hours; the dynamic SE-index was the fastest (about 5 hours), followed by the dynamic RR-index (about 8 hours). On the Pizza&Chili repetitive corpus, all methods finished within 7 minutes for every dataset, with no significant difference among the four.

Table 6 Construction time for all datasets. The fastest construction time among the dynamic methods in each row is shown in bold.

	Construction Time (min)			
Data	Dynamic RR-index	Dynamic SE-index	Dynamic r-index	r-index
cere	**4.9**	5.3	6.3	5.7
coreutils	2.8	3.2	**2.4**	2.2
einstein.de.txt	0.9	**0.4**	0.7	0.6
einstein.en.txt	4.6	**2.1**	3.5	3.1
Escherichia Coli	**1.9**	3.5	2.5	1.9
influenza	1.9	2.2	**1.8**	1.6
kernel	2.9	**2.4**	2.6	2.5
para	**5.1**	6.2	6.8	5.9
world leaders	**0.4**	**0.4**	0.5	0.4
enwiki	469.4	**315.1**	648.7	664.1
chr19	554.6	**285.8**	868.4	863.4

Table 3 Average time and standard deviation of substring insertions for every dataset. For the (static) r-index, each row shows the construction time, since it does not support updates. The fastest dynamic method in each row is shown in bold.

	Insertion Time for $m' = 1$ (ms)			
Data	Dynamic RR-index	Dynamic SE-index	Dynamic r-index	r-index
cere	**24 ± 7**	36 ± 6	62 ± 115	378,650
coreutils	**27 ± 7**	31 ± 7	2,051 ± 4,641	143,300
einstein.de.txt	**13 ± 2**	16 ± 2	221 ± 177	41,880
einstein.en.txt	**14 ± 2**	24 ± 3	497 ± 436	207,580
Escherichia Coli	**22 ± 8**	31 ± 6	158 ± 565	152,760
influenza	21 ± 4	33 ± 6	**6 ± 8**	109,180
kernel	43 ± 6	**30 ± 6**	2,365 ± 3,702	154,020
para	**25 ± 9**	64 ± 8	29 ± 39	409,420
world leaders	**15 ± 3**	20 ± 3	65 ± 331	27,660
enwiki	**51 ± 56**	117 ± 46	883 ± 829	38,921,100
chr19	**55 ± 29**	73 ± 25	824 ± 2,317	52,106,240

	Insertion Time for $m' = 10$ (ms)			
Data	Dynamic RR-index	Dynamic SE-index	Dynamic r-index	r-index
cere	**24 ± 7**	37 ± 7	60 ± 118	378,650
coreutils	**27 ± 8**	30 ± 6	2,099 ± 4,414	143,300
einstein.de.txt	**13 ± 2**	17 ± 2	219 ± 172	41,880
einstein.en.txt	**15 ± 3**	24 ± 3	503 ± 445	207,580
Escherichia Coli	**21 ± 7**	35 ± 8	133 ± 516	152,760
influenza	21 ± 4	32 ± 5	**7 ± 8**	109,180
kernel	44 ± 6	**27 ± 4**	2,150 ± 3,417	154,020
para	**28 ± 10**	66 ± 10	32 ± 45	409,420
world leaders	**15 ± 3**	20 ± 3	87 ± 426	27,660
enwiki	**55 ± 58**	111 ± 44	818 ± 770	38,921,100
chr19	**55 ± 29**	74 ± 25	919 ± 2,781	52,106,240

	Insertion Time for $m' = 100$ (ms)			
Data	Dynamic RR-index	Dynamic SE-index	Dynamic r-index	r-index
cere	**24 ± 7**	39 ± 8	56 ± 114	378,650
coreutils	**29 ± 8**	34 ± 7	2,053 ± 4,573	143,300
einstein.de.txt	**14 ± 2**	18 ± 2	226 ± 180	41,880
einstein.en.txt	**15 ± 2**	25 ± 3	521 ± 448	207,580
Escherichia Coli	**21 ± 7**	33 ± 7	131 ± 484	152,760
influenza	21 ± 4	37 ± 7	**7 ± 8**	109,180
kernel	47 ± 6	**28 ± 4**	2,156 ± 3,491	154,020
para	**29 ± 10**	71 ± 11	31 ± 36	409,420
world leaders	**14 ± 2**	21 ± 3	72 ± 382	27,660
enwiki	**50 ± 55**	117 ± 43	841 ± 849	38,921,100
chr19	**62 ± 31**	83 ± 29	925 ± 2,713	52,106,240

	Insertion Time for $m' = 1000$ (ms)			
Data	Dynamic RR-index	Dynamic SE-index	Dynamic r-index	r-index
cere	**26 ± 7**	44 ± 9	60 ± 104	378,650
coreutils	**28 ± 6**	40 ± 8	2,025 ± 4,282	143,300
einstein.de.txt	**15 ± 2**	19 ± 2	223 ± 173	41,880
einstein.en.txt	**15 ± 2**	27 ± 3	520 ± 498	207,580
Escherichia Coli	**23 ± 7**	37 ± 7	148 ± 482	152,760
influenza	23 ± 4	36 ± 5	**11 ± 8**	109,180
kernel	53 ± 9	**33 ± 5**	1,997 ± 3,061	154,020
para	**32 ± 11**	75 ± 9	35 ± 37	409,420
world leaders	**16 ± 3**	22 ± 3	86 ± 435	27,660
enwiki	**58 ± 56**	125 ± 43	885 ± 824	38,921,100
chr19	**63 ± 30**	81 ± 25	1,122 ± 3,625	52,106,240

Table 4 Average locate time and standard deviation for patterns of length $m \in \{10, 100, 1000\}$ for every dataset. The fastest dynamic method in each row is shown in bold.

	Locate Time for $m = 10$ (ms)			
Data	Dynamic RR-index	Dynamic SE-index	Dynamic r-index	r-index
cere	**1.8 ± 2.9**	3.5 ± 6.5	3.2 ± 6.9	0.4 ± 0.9
coreutils	**8.3 ± 51.1**	13.6 ± 71.5	11.4 ± 48.8	1.5 ± 6.0
einstein.de.txt	**1.1 ± 2.9**	2.5 ± 9.7	6.6 ± 26.5	1.0 ± 4.0
einstein.en.txt	**2.3 ± 6.2**	7.7 ± 27.5	22.0 ± 81.0	2.5 ± 9.0
Escherichia Coli	**1.1 ± 0.8**	1.9 ± 1.1	1.3 ± 1.3	0.2 ± 0.1
influenza	**4.2 ± 4.5**	7.7 ± 6.3	5.2 ± 4.3	0.7 ± 0.6
kernel	**1.3 ± 5.3**	2.8 ± 11.1	4.7 ± 21.4	0.7 ± 3.2
para	**2.0 ± 4.4**	3.7 ± 4.2	3.2 ± 3.7	0.4 ± 0.5
world leaders	27.4 ± 78.5	**25.4 ± 61.4**	31.1 ± 84.4	3.2 ± 8.9
enwiki	**101.8 ± 483.7**	532.1 ± 2,979.3	1,093.3 ± 6,018.5	96.3 ± 546.9
chr19	**330.0 ± 781.4**	2,362.9 ± 6,722.4	3,793.2 ± 10,835.2	298.1 ± 845.1

	Locate Time for $m = 100$ (ms)			
Data	Dynamic RR-index	Dynamic SE-index	Dynamic r-index	r-index
cere	**0.6 ± 2.1**	1.7 ± 1.5	**0.6 ± 0.6**	0.2 ± 0.1
coreutils	**1.0 ± 1.5**	2.0 ± 1.6	**1.0 ± 0.9**	0.3 ± 0.2
einstein.de.txt	**0.7 ± 0.3**	1.4 ± 0.4	1.2 ± 0.7	0.3 ± 0.1
einstein.en.txt	**1.3 ± 0.9**	2.6 ± 1.2	3.4 ± 2.3	0.6 ± 0.3
Escherichia Coli	**0.4 ± 0.1**	1.5 ± 0.2	0.6 ± 0.1	0.1 ± 0.0
influenza	1.9 ± 2.5	3.4 ± 3.3	**0.9 ± 0.9**	0.2 ± 0.1
kernel	**0.6 ± 0.2**	1.4 ± 0.3	1.0 ± 0.5	0.2 ± 0.1
para	**0.5 ± 0.3**	1.6 ± 0.2	0.6 ± 0.1	0.2 ± 0.0
world leaders	2.5 ± 6.5	3.5 ± 6.6	**2.2 ± 5.0**	0.4 ± 0.7
enwiki	**2.1 ± 1.5**	7.3 ± 4.9	5.3 ± 4.3	0.8 ± 0.5
chr19	**1.3 ± 0.5**	8.3 ± 2.9	5.2 ± 1.7	0.6 ± 0.2

	Locate Time for $m = 1000$ (ms)			
Data	Dynamic RR-index	Dynamic SE-index	Dynamic r-index	r-index
cere	**1.5 ± 0.3**	4.2 ± 0.7	3.8 ± 0.3	1.1 ± 0.2
coreutils	**2.1 ± 0.4**	4.1 ± 0.5	6.7 ± 0.3	1.5 ± 0.4
einstein.de.txt	**1.6 ± 0.5**	3.2 ± 0.9	3.6 ± 0.7	1.3 ± 0.4
einstein.en.txt	**2.1 ± 0.8**	3.9 ± 0.9	5.3 ± 0.9	1.1 ± 0.2
Escherichia Coli	**1.5 ± 0.2**	3.8 ± 0.3	4.8 ± 0.2	1.0 ± 0.2
influenza	**2.4 ± 0.8**	4.4 ± 1.1	3.9 ± 0.2	1.0 ± 0.2
kernel	**1.8 ± 0.3**	3.8 ± 0.6	6.3 ± 0.2	1.4 ± 0.4
para	**1.5 ± 0.3**	4.2 ± 0.5	4.0 ± 0.3	0.9 ± 0.1
world leaders	**1.6 ± 0.4**	2.9 ± 0.7	4.1 ± 0.5	1.2 ± 0.4
enwiki	**3.6 ± 0.9**	8.4 ± 1.7	14.1 ± 3.4	2.8 ± 2.0
chr19	**3.2 ± 1.0**	9.6 ± 1.9	8.4 ± 1.8	2.4 ± 2.0