Dynamic Suffix Array with Polylogarithmic Queries and Updates*

Dominik Kempa
Stony Brook University
Stony Brook, New York, USA
kempa@cs.stonybrook.edu

Tomasz Kociumaka†
University of California
Berkeley, California, USA
kociumaka@mimuw.edu.pl

ABSTRACT

The suffix array SA[1..n] of a text T of length n is a permutation of $\{1, \ldots, n\}$ describing the lexicographical ordering of suffixes of T and is considered to be one of the most important data structures for string processing, with dozens of applications in data compression, bioinformatics, and information retrieval. One of the biggest drawbacks of the suffix array is that it is very difficult to maintain under text updates: even a single character substitution can completely change the contents of the suffix array. Thus, the suffix array of a dynamic text is modelled using *suffix array queries*, which return the value SA[i] given any $i \in [1..n]$.

Prior to this work, the fastest dynamic suffix array implementations were by Amir and Boneh, who showed how to answer suffix array queries in $\widetilde{O}(k)$ time, where $k \in [1..n]$ is a trade-off parameter, with $\widetilde{O}(n/k)$-time text updates [ISAAC 2020]. In a very recent preprint, they also provided a solution with $O(\log^5 n)$-time queries and $\widetilde{O}(n^{2/3})$-time updates [arXiv 2021].

We propose the first data structure that supports both suffix array queries and text updates in $O(\text{polylog}\, n)$ time (achieving $O(\log^4 n)$ and $O(\log^{3+o(1)} n)$ time, respectively). Our data structure is deterministic and the running times for all operations are worst-case. In addition to the standard single-character edits (character insertions, deletions, and substitutions), we support (also in $O(\log^{3+o(1)} n)$ time) the "cut-paste" operation that moves any (arbitrarily long) substring of T to any place in T. To achieve our result, we develop a number of new techniques which are of independent interest. This includes a new flavor of dynamic locally consistent parsing, as well as a dynamic construction of string synchronizing sets with an extra *local sparsity* property; this significantly generalizes the sampling technique introduced at STOC 2019. We complement our structure by a hardness result: unless the Online Matrix-Vector Multiplication (OMv) Conjecture fails, no data structure with $O(\text{polylog}\, n)$-time suffix array queries can support the "copy-paste" operation in $O(n^{1-\varepsilon})$ time for any $\varepsilon > 0$.

CCS CONCEPTS

• **Theory of computation → Pattern Matching**; *Data structures design and analysis*; Cell probe models and lower bounds; Problems, reductions and completeness.

KEYWORDS

Suffix array, text indexing, pattern matching, string synchronizing sets, dynamic data structures

ACM Reference Format:
Dominik Kempa and Tomasz Kociumaka. 2022. Dynamic Suffix Array with Polylogarithmic Queries and Updates. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing (STOC '22), June 20–24, 2022, Rome, Italy*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3519935.3520061

1 INTRODUCTION

For a text T of length n, the suffix array SA[1..n] stores the permutation of $\{1, \ldots, n\}$ such that T[SA[i]..n] is the ith lexicographically smallest suffix of T. The original application of SA [49] was to solve the *text indexing* problem: construct a data structure such that, given a pattern $P[1..m]$ (typically with $m \ll n$), we can quickly count (and optionally list) all occurrences of P in T. Since the sought set of positions occupies a contiguous block SA[$b..e$) (for some $b, e \in [1..n+1]$) and since, given $j \in [1..n]$, we can in $O(m)$ time check if the value SA[j] is before, inside, or after this block, the indices b and e can be computed in $O(m \log n)$ time with binary search. If $b < e$, we can then report all occurrences of P in T at the extra cost of $O(e - b)$. Soon after the discovery of SA it was realized that a very large set of problems on strings is essentially solved (or at least becomes much easier) once we have a suffix array (often augmented with the *LCP array* [39, 49]). This includes, for example, the following problems (see also the textbook of Gusfield [34]):

- finding repeats (e.g., MaximalRepeats, LongestRepeatedFactor, TandemRepeats);
- computing special subwords (e.g., MinimalAbsentWord, ShortestUniqueSubstring);
- sequence comparisons (e.g., LongestCommonSubstring, MaximalUniqueMatches); and
- data compression (e.g., LZ77Factorization, BWTCompression).

This trend continues in more recent textbooks [47, 53, 59], with the latest suffix array representations (such as *FM-index* [24], *compressed suffix array* [32], and *r-index* [28]) as central data structures. There are even textbooks such as [1] solely dedicated to the applications of suffix arrays and the closely related Burrows–Wheeler transform (BWT) [12].

The power of suffix array comes with one caveat: It is very difficult to maintain it for a text undergoing updates. For example,

*A full version of this paper is available at arxiv.org/abs/2201.01285 [43]. Proofs of the claims marked with ♠ are presented only in the full version.

†Partially supported by NSF 1652303, 1909046, and HDR TRIPODS 1934846 grants, and an Alfred P. Sloan Fellowship.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
STOC '22, June 20–24, 2022, Rome, Italy
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9264-8/22/06…$15.00
https://doi.org/10.1145/3519935.3520061

for $T = \mathsf{b}^n$ (symbol b repeated n times) we have $\mathrm{SA}_T = [n, \ldots, 2, 1]$, whereas for $T' = \mathsf{b}^{n-1}\mathsf{c}$ (obtained from T with a single substitution), it holds $\mathrm{SA}_{T'} = [1, 2, \ldots, n]$, i.e., the complete reversal. Even worse, if $n = 2m + 1$ and $T'' = \mathsf{b}^m\mathsf{a}\mathsf{b}^m$ (again, a single substitution), then $\mathrm{SA}_{T''} = [m{+}1, n, m, n-1, m-1, \ldots, m+2, 1]$, i.e., the near-perfect interleaving of two halves of SA_T. In general, even a single character substitution may permute SA in a very complex manner. Thus, if one wishes to maintain the suffix array of a dynamic text, its entries cannot be stored in plain form but must be obtained by querying a data structure. The quest for such *dynamic suffix array* is open since the birth of suffix array over three decades ago. We thus pose our problem:

Problem 1.1. Can we support efficient SA queries for a dynamically changing text?

Previous Work. One of the first attempts to tackle the above problem is due to Salson, Lecroq, Léonard, and Mouchard [63]. Their dynamic suffix array achieves good practical performance on real-world texts (including English text and DNA), but its update takes $\Theta(n)$ time in the worst-case. A decade later, Amir and Boneh [5] proposed a structure that, for any parameter $k \in [1..n]$, supports SA and SA^{-1} queries in $\widetilde{O}(k)$ time and character substitutions in $\widetilde{O}(n/k)$ time. This implies the first nontrivial trade-off for the dynamic suffix array, e.g., $\widetilde{O}(\sqrt{n})$-time operations. Very recently, Amir and Boneh [6] described a dynamic suffix array that supports updates (insertions and deletions) in the text in $\widetilde{O}(n^{2/3})$ time and SA queries in $O(\log^5 n)$ time. They also gave a structure that supports substitutions in $\widetilde{O}(n^{1/2})$ time and SA^{-1} queries in $O(\log^4 n)$ time.

A separate line of research focused on the related problem of *dynamic text indexing* introduced by Gu, Farach, and Beigel [33]. This problem aims to design a data structure that permits updates to the text T and pattern searches (asking for all occurrences of a given pattern P in T). As noted in [5], the solution in [33] achieves $\widetilde{O}(\sqrt{n})$-time updates to text and $\widetilde{O}(m\sqrt{n} + \mathrm{occ}\log n)$ pattern search query (where occ is the number of occurrences of P in T). Sahinalp and Vishkin [62] then proposed a solution based on the idea of *locally consistent parsing* that achieves $O(\log^3 n)$-time update and $O(m + \mathrm{occ})$ pattern searching time. The update time was later improved by Alstrup, Brodal, and Rauhe [3] to $O(\log^2 n \log\log n \log^* n)$ at the expense of the slightly slower query $O(m + \mathrm{occ} + \log n \log\log n)$. This last result was achieved by building on techniques for the *dynamic string equality* problem proposed by Mehlhorn, Sundar, and Uhrig [51]. This was improved in [29] to $O(\log^2 n)$-time update and $O(m{+}\mathrm{occ})$-time search. A slightly different approach to dynamic text indexing, based on *dynamic position heaps* was proposed in [23]. It achieves $O(m \log n + \mathrm{occ})$ amortized search, but the updates take $\Theta(n)$ time in the worst case. There is also work on *dynamic compressed text indexing*. Chan, Hon, Lam, and Sadakane [13] proposed an index that uses $O(\frac{1}{\varepsilon}(nH_0(T) + n))$ bits of space (where $H_0(T)$ is the zeroth order empirical entropy of T), searches in $O(m \log^2 n(\log^\varepsilon n{+}\log\sigma){+}\mathrm{occ}\log^{1+\varepsilon} n)$ time (where σ is the alphabet size), and performs updates in $O(\sqrt{n}\log^{2+\varepsilon} n)$ amortized time, where $0 < \varepsilon \le 1$. Recently, Nishimoto, I, Inenaga, Bannai, and Takeda [58] proposed a faster index for a text T with LZ77 factorization of size z. Assuming for simplicity $z \log n \log^* n = O(n)$ and $z = \Omega(\log n \log^* n)$, their index occupies $O(z \log n \log^* n)$ space, performs updates in amortized $O((\log n \log^* n)^2 \log z)$ time (in addition to edits, they also support the "cut-paste" operation that moves a substring of T from one place to another), and searches in time $O(m \cdot \min\{\log\log n \log\log z/\log\log\log n, \sqrt{\log z/\log\log z}\} + \log z \log m \log^* n(\log n + \log m \log^* n) + \mathrm{occ}\log n)$.

We point out that although the (compressed) dynamic text indexing problem [33, 58] discussed above is related to dynamic suffix array, it is not the same. Assuming one accepts additional log factors, the dynamic suffix array problem is strictly harder: it solves dynamic indexing (by simply adding binary search on top), but no converse reduction is known; such a reduction would compute values of SA in $O(\mathrm{polylog}\, n)$ time using searches for short patterns. Thus, the many applications of SA listed above cannot be solved with these indexes. Unfortunately, due to lack of techniques, the dynamic suffix array problem has seen very little progress; as noted by Amir and Boneh [5], "(…) although a dynamic suffix array algorithm would be extremely useful to automatically adapt many static pattern matching algorithms to a dynamic setting, other techniques had to be sought". They remark, however, that "Throughout all this time, an algorithm for maintaining the suffix tree or suffix array of a dynamically changing text had been sought". To sum up, until now, the best dynamic suffix arrays have been those of [5], taking $\widetilde{O}(k)$ time to answer SA and SA^{-1} queries and $\widetilde{O}(n/k)$ time for substitutions, and [6], taking $\widetilde{O}(n^{2/3})$ time for insertions/deletions and $O(\log^5 n)$ time for SA queries, or $\widetilde{O}(n^{1/2})$ time for substitutions and $O(\log^4 n)$ time for SA^{-1} queries. No solution with polylog n-time queries and updates (even amortized or expected) was known, not even for character substitutions only.

Our Results. We propose the first dynamic suffix array with all operations (queries *and* updates) taking only $O(\mathrm{polylog}\, n)$ time. Our data structure is deterministic and the complexities of both queries and updates are worst-case. Thus, we leap directly to a solution satisfying the commonly desired properties on the query and update complexity for this over thirty-years old open problem. In addition to single-character edits, our structure supports the powerful "cut-paste" operation, matching the functionality of state-of-the-art indexes [3, 58]. More precisely, our structure supports the following operations (the bounds below are simplified overestimates; see Theorem 6.7):

- We support SA queries (given $i \in [1..n]$, return $\mathrm{SA}[i]$) in $O(\log^4 N)$ time.
- We support SA^{-1} queries (given $j \in [1..n]$, return $\mathrm{SA}^{-1}[j]$) in $O(\log^5 N)$ time.
- We support updates (insertion and deletion of a single symbol in T as well as the "cut-paste" operation, moving any block of T to any other place in T) in $O(\log^{3+o(1)} N)$ time.

Here, $N = n\sigma$ is the product of the current text length and the size of the alphabet $\Sigma = [0..\sigma)$.

The above result may leave a sense of incompleteness regarding further updates such as "copy-paste". We show that, despite its similarity with "cut-paste", supporting this operation in the dynamic setting is most likely very costly. More precisely, we prove that, unless the Online Matrix-Vector Multiplication (OMv) Conjecture [37] fails, no data structure that supports SA or SA^{-1} queries in $O(\mathrm{polylog}\, n)$ time can support the "copy-paste" operation in $O(n^{1-\varepsilon})$ time for any $\varepsilon > 0$. In fact, we prove the following more general result (which can be easily extended to SA queries; see [43]):

Theorem 1.2 (♠). *For all constants $\alpha, \beta > 0$ with $\alpha + \beta < 1$, the OMv Conjecture implies that there is no dynamic algorithm that preprocesses a text T in time polynomial in $|T|$, supports copy-pastes in $O(|T|^{\alpha})$ time, and inverse suffix array queries in $O(|T|^{\beta})$ time, with each answer correct with probability at least $\frac{2}{3}$.*

Thus, the trade-off similar to the one achieved by Amir and Boneh [5] ($\widetilde{O}(k)$-time query and $\widetilde{O}(n/k)$-time update) may still be possible for a dynamic suffix array with copy-pastes; we leave proving such upper bound as a possible direction for future work.

Technical Contributions. To achieve our result, we develop new techniques and significantly generalize several existing ones. Our first novel technique is the notion of *locally sparse* synchronizing sets. String synchronizing sets [40] have recently been introduced in the context of efficient construction of BWT and LCE queries for "packed strings" (where a single machine word stores multiple characters). Since then, they have found many other applications [2, 4, 14, 41, 42]. Loosely speaking (a formal definition follows in Section 2), for any fixed $\tau \geq 1$, a set $\mathsf{S} \subseteq [1..n]$ is a τ-synchronizing set of a string $T \in \Sigma^n$ if S samples positions of T *consistently* (i.e., whether $i \in [1..n]$ is sampled depends only on the length-$\Theta(\tau)$ context of i in T) and samples everywhere in T except in highly periodic fragments (the so-called *density* condition). In all prior applications utilizing synchronizing sets, the goal is to ensure that S is *sparse on average*, i.e., that the size $|\mathsf{S}|$ is minimized. In this paper, we prove that at the cost of increasing the size of $|\mathsf{S}|$ by a mere factor $O(\log^*(\tau\sigma))$, we can additionally ensure that S is also *locally sparse*. We then show that such S can be maintained dynamically using a new construction of S from the signature parsing (a technique introduced in [51] and used, for example, in [3, 58]). The crucial benefit of local sparsity is that any auxiliary structure that depends on the length-$\Theta(\tau)$ contexts of positions in S, including S itself, can be updated efficiently. Another result of independent interest is the first dynamic construction of string synchronizing sets. For this, we internally represent some substrings of T using the abstraction of *dynamic strings* [3, 30, 51, 61]. The problem with all existing variants of dynamic strings, however, is that they rely on representing the strings using a hierarchical representation whose only goal is to ensure the string shrinks by a constant factor at each level. This, however, is not sufficient for our purpose: in order to satisfy the density condition for the resulting synchronizing set, we also need all symbols at any given level to have some common upper bound on the expansion length. The notion of such "balanced" parsing is known [11, 61], but has only been implemented in static settings. We show the first variant of dynamic strings that maintains a "balanced" parsing at every level, and consequently lets us dynamically maintain a locally sparse string synchronizing set.

The mainstay among data structures for pattern matching or SA queries is the use of (typically 2D) orthogonal range searching [16]. Example indexes include nearly all indexes based on LZ77 [7–9, 18, 27, 38, 45, 58], context-free grammars [10, 19, 20, 26, 50, 64, 65], and some recent BWT-based indexes [17, 42, 52]. Nearly all these structures maintain a set of points corresponding to some set of suffixes of T (identified with their starting positions $\mathsf{P} \subseteq [1..n]$) ordered lexicographically. The problem with adapting this to the dynamic setting is that once we modify T near the end, the order of (potentially all) elements in P changes. We overcome this obstacle as follows. Rather than on sampling of suffixes, we rely on $\log n$ levels of sampling of *substrings* (identified by the set S_k of the starting positions of their occurrences) of length roughly 2^k, where $k \in [0..\lceil \log n \rceil]$, implemented using dynamic locally sparse synchronizing sets. With such structure, we can efficiently update the sets S_k and the associated geometric structures, but we lose the ability to easily compare suffixes. In Section 5, we show that even with such partial sampling, we can nevertheless still implement SA queries. In $\log n$ steps, our query algorithm successively narrows the range of SA to contain only suffixes prefixed with $T[\mathsf{SA}[i]..\mathsf{SA}[i] + \ell)$, while also maintaining the starting position of some arbitrary suffix in the range, where $\ell = 2^k$ is the current prefix length. In the technical overview, we sketch the main ideas of this reduction, but we remark that the details of this approach are nontrivial and require multiple technical results (see [43]).

Finally, we remark that, even though (as noted earlier), dynamic suffix array is a strictly harder problem that text indexing (since one can be reduced to the other, but not the other way around), our result has implications even for the text indexing problem. The existing dynamic text indexes with fast queries and updates (such as [29, 58]) can only *list* all k occurrences of any pattern. One, however, cannot obtain the number of occurrences (which is the standard operation supported by most of the static indexes [54]) in time that is always bounded by $O(m \operatorname{polylog} n)$ (since k can be arbitrarily large).[1] Our dynamic suffix array, on the other hand, implements counting seamlessly: it suffices to perform the standard binary search [49] over SA for the pattern P, resulting in the endpoints of range $\mathsf{SA}[b..e)$ of suffixes of T prefixed with P, and return $e - b$.

Related Work. Chan, Hon, Lam, and Sadakane [13] introduced a problem of *indexing text collections*, which asks to store a dynamically changing collection C of strings (under insertions and deletions of *entire strings*) so that pattern matching queries on C can be answered efficiently. Although the resulting data structures are also called *dynamic full-text indexes* [48] or *dynamic suffix trees* [60], we remark that they are solving a different problem that what, by analogy to dynamic suffix array, we would mean by "dynamic suffix tree". Observe that we cannot simulate the insertion of a symbol in the middle of T using a collection of strings. Maintaining symbols of T as a collection of length-1 strings will not work because the algorithms in [13, 48, 60] only report occurrences entirely inside one of the strings. Since the insertion of a string S of length m into C takes $\Omega(m)$ time in [13, 48, 60] (similarly for deletion), one also cannot efficiently use C to represent the entire text T as a single element of C.

Related to the problem of text indexing is also the problem of *sequence representation* [55], where we store a string $S[1..n]$ under character insertions and deletions and support the access to S, $\mathrm{rank}(i, c)$ (returning $|\{j \in [1..i] : S[j] = c\}|$) and $\mathrm{select}(i, c)$ (returning the ith occurrence of c in S). A long line of research, including [31, 36, 48, 56], culminated with the work of Navarro and Nekrich, who achieved $O(\log n / \log\log n)$ time for all operations

[1] Efficient counting for indexes relying on orthogonal range searching is a technically challenging problem that has been solved only recently for the static case [18]. It is possible that these ideas can be combined with [29] or [58], but the result will nevertheless be significantly more complicated than counting using the suffix array.

i	SA[i]	T[SA[i]..n]
1	20	$
2	19	a$
3	14	aababa$
4	5	aababababaababa$
5	17	aba$
6	12	abaababa$
7	3	abaababababaababa$
8	15	ababa$
9	10	ababaababa$
10	8	abababaababa$
11	6	ababababaababa$
12	18	ba$
13	13	baababa$
14	4	baababababaababa$
15	16	baba$
16	11	babaababa$
17	2	babaababababaababa$
18	9	bababaababa$
19	7	bababababaababa$
20	1	bbabaababababaababa$

Figure 1: The suffix array of T = bbabaababababaababa$ along with a sorted list of the suffixes of T.

(amortized for updates), while using space close to $nH_0(S)$, where H_0 is the zeroth order empirical entropy.

Finally, there is a line of research focusing on storing a text T under updates so that we can support efficient *longest common extension* queries $\mathrm{LCE}_T(i, j)$ that return the length of the longest common prefix of $T[i..|T|]$ and $T[j..|T|]$. The research was initiated with the seminal work in [3, 51] (recently improved in [30]). More recently, a solution working in the compressed space (achieving similar runtimes as the index in [58]) was proposed in [57].

2 PRELIMINARIES

A *string* is a finite sequence of characters from some set Σ called the *alphabet*. We denote the length of a string S as $|S|$. For any index $i \in [1..|S|]$,[2] we denote the ith character of S as $S[i]$. A string of the form $S[i..j) = S[i]S[i+1]\dots S[j-1]$, where $1 \le i \le j \le |S|+1$ is called a *fragment* of S. If S is known, we will encode $S[i..j)$ in $O(1)$ space as a pair (i, j). Fragments of the form $S[1..j)$ and $S[i..|S|]$ are called *prefixes* and *suffixes*, respectively. By $\overline{S}$ we denote the *reverse* of S, i.e., $\overline{S} = S[|S|]\dots S[2]S[1]$. The *concatenation* of two strings U and V is denoted UV or $U \cdot V$. Moreover, $S^k = \bigodot_{i=1}^{k} S$ is the concatenation of k copies of S; note that $S^0 = \varepsilon$ is the *empty string*. An integer $p \in [1..|S|]$ is a called a *period* of S if $S[1..|S|-p] = S[1+p..|S|]$; we denote the shortest period of S as $\mathrm{per}(S)$. We use $\preceq$ to denote the order on Σ, extended to the *lexicographic* order on Σ^* (the set of strings over Σ) so that $U, V \in \Sigma^*$ satisfy $U \preceq V$ if and only if either U is a prefix of V, or $U[1..i) = V[1..i)$ and $U[i] \prec V[i]$ holds for some $i \in [1..\min(|U|, |V|)]$.

Throughout, we consider a string (called the *text*) T of length $n \ge 1$ over an integer alphabet $\Sigma = [0..\sigma)$. We assume that $T[n] = \$$ (where $\$ = \min\Sigma$) and $\$$ does not occur in $T[1..n)$.

The *suffix array* SA[1..n] of T is a permutation of $[1..n]$ such that $T[\mathrm{SA}[1]..n] \prec T[\mathrm{SA}[2]..n] \prec \cdots \prec T[\mathrm{SA}[n]..n]$, i.e., SA[$i$] is the starting position of the lexicographically ith smallest suffix of T; see Fig. 1 for an example. The *inverse suffix array* $\mathrm{SA}^{-1}[1..n]$ is the inverse permutation of SA, i.e., $\mathrm{SA}^{-1}[j] = i$ holds if and only if $\mathrm{SA}[i] = j$. By $\mathrm{lcp}(U, V)$ we denote the length of the longest common prefix of U and V. For $j_1, j_2 \in [1..n]$, we let $\mathrm{LCE}_T(j_1, j_2) = \mathrm{lcp}(T[j_1..], T[j_2..])$.

The *rotation* operation $\mathrm{rot}(\cdot)$, given a string $S \in \Sigma^+$, moves the last character of S to the front so that $\mathrm{rot}(S) = S[|S|] \cdot S[1..|S|-1]$.

We use the word RAM model of computation [35] with w-bit *machine words*, where $w \ge \log n$.

Definition 2.1 (τ-synchronizing set [40]). Let $T \in \Sigma^n$ be a string and let $\tau \in [1..\lfloor \frac{n}{2} \rfloor]$ be a parameter. A set $\mathrm{S} \subseteq [1..n-2\tau+1]$ is called a *τ-synchronizing set* of T if it satisfies the following *consistency* and *density* conditions:

(1) If $T[i..i+2\tau) = T[j..j+2\tau)$, then $i \in \mathrm{S}$ holds if and only if $j \in \mathrm{S}$ (for $i, j \in [1..n-2\tau+1]$),
(2) $\mathrm{S} \cap [i..i+\tau) = \emptyset$ if and only if $i \in \mathrm{R}(\tau, T)$ (for $i \in [1..n-3\tau+2]$), where

$$\mathrm{R}(\tau, T) := \{i \in [1..|T|-3\tau+2] : \mathrm{per}(T[i..i+3\tau-2]) \le \tfrac{1}{3}\tau\}.$$

In most applications, we want to minimize $|\mathrm{S}|$. The density condition imposes a lower bound $|\mathrm{S}| = \Omega(\frac{n}{\tau})$ for strings of length $n \ge 3\tau - 1$ that do not contain highly periodic substrings of length $3\tau - 1$. Hence, in the worst case, we cannot hope to improve upon the following bound.

Theorem 2.2 ([40, Proposition 8.10]). *For every $T \in \Sigma^n$ and $\tau \in [1..\lfloor \frac{n}{2} \rfloor]$, there exists a τ-synchronizing set S of size $|\mathrm{S}| = O(\frac{n}{\tau})$. Such S can be deterministically constructed in $O(n)$ time.*

3 TECHNICAL OVERVIEW

We derive our dynamic suffix array gradually. We start (Section 4), by introducing the auxiliary tools. In the first part of the paper (Section 5) we prove that if we have $\Theta(\log n)$ synchronizing sets for values of τ spanning across the whole range $[1..n]$, and we can support some set of queries (stated as "assumptions") concerning either positions in those synchronizing sets, gaps across successive elements, or their length-$\Theta(\tau)$ contexts in T, then we can support SA queries on T. In the second part of the paper (Section 6) we then show how to satisfy these assumptions for text supporting update operations. This approach lets us separate the clean (combinatorial) details concerning SA queries from (more algorithmic and technical) details concerning the maintenance of the underlying synchronizing sets and the associated structures. This also lets us for now treat T as well as each of the synchronizing sets as static, since the reduction works for *any* collection of such sets, and thus if after the update these sets (and the associated structures) change, the query algorithm is unaffected.

To give an overview of Section 5, we first introduce the key notation. Let $\ell \ge 1$. For any $j \in [1..n]$, we define

$$\mathrm{Occ}_\ell(j) = \{j' \in [1..n] : T^\infty[j'..j'+\ell) = T^\infty[j..j+\ell)\},$$
$$\mathrm{RBeg}_\ell(j) = |\{j' \in [1..n] : T[j'..n] \prec T[j..n] \wedge \mathrm{LCE}_T(j, j') < \ell\}|,$$
$$\mathrm{REnd}_\ell(j) = \mathrm{RBeg}_\ell(j) + |\mathrm{Occ}_\ell(j)|,$$

where T^∞ is defined by $T^\infty[i] = T[1 + (i-1) \bmod n]$ ($i \in \mathbb{Z}$). In particular, $T^\infty[1..n] = T$. Note that if $P = T^\infty[j..j+\ell)$, then $\{\mathrm{SA}[i] : i \in (\mathrm{RBeg}_\ell(j)..\mathrm{REnd}_\ell(j))\} = \{i \in [1..n] : T^\infty[i..i+|P|) = P\}$.

[2] For $i, j \in \mathbb{Z}$, denote $[i..j] = \{k \in \mathbb{Z} : i \le k \le j\}$, $[i..j) = \{k \in \mathbb{Z} : i \le k < j\}$, and $(i..j] = \{k \in \mathbb{Z} : i < k \le j\}$.

Moreover, for any $j \in [1..n]$, we define

$$\mathrm{Pos}_\ell(j) = \{j' \in [1..n] : T[j'..n] \prec T[j..n] \wedge \ell \le \mathrm{LCE}_T(j,j') < 2\ell\},$$
$$\mathrm{Pos}'_\ell(j) = \{j' \in [1..n] : T[j'..n] \succ T[j..n] \wedge \ell \le \mathrm{LCE}_T(j,j') < 2\ell\}.$$

We denote $\delta_\ell(j) := |\mathrm{Pos}_\ell(j)|$ and $\delta'_\ell(j) := |\mathrm{Pos}'_\ell(j)|$. Observe, that it holds

$$(\mathrm{RBeg}_{2\ell}(j), \mathrm{REnd}_{2\ell}(j)) = (\mathrm{RBeg}_\ell(j) + \delta_\ell(j), \mathrm{RBeg}_\ell(j) + \delta_\ell(j) + |\mathrm{Occ}_{2\ell}(j)|),$$
$$(\mathrm{RBeg}_{2\ell}(j), \mathrm{REnd}_{2\ell}(j)) = (\mathrm{REnd}_\ell(j) - \delta'_\ell(j) - |\mathrm{Occ}_{2\ell}(j)|, \mathrm{REnd}_\ell(j) - \delta'_\ell(j)).$$

The main idea of our algorithm is as follows. Suppose that we have obtained $(\mathrm{RBeg}_{16}(\mathrm{SA}[i]), \mathrm{REnd}_{16}(\mathrm{SA}[i]))$ and some $j \in \mathrm{Occ}_{16}(\mathrm{SA}[i])$ (Assumption 5.1). Then, for $q = 4, \ldots, \lceil \log n \rceil - 1$, denoting $\ell = 2^q$, we compute $(\mathrm{RBeg}_{2\ell}(\mathrm{SA}[i]), \mathrm{REnd}_{2\ell}(\mathrm{SA}[i]))$ and some $j' \in \mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$, by using as input some position $j \in \mathrm{Occ}_\ell(\mathrm{SA}[i])$ and the pair $(\mathrm{RBeg}_\ell(\mathrm{SA}[i]), \mathrm{REnd}_\ell(\mathrm{SA}[i]))$, i.e., the output of the earlier step. This lets us compute $\mathrm{SA}[i]$, since eventually we obtain some $j' \in \mathrm{Occ}_{2^{\lceil \log n \rceil}}(\mathrm{SA}[i])$, and for any $k \ge n$, $\mathrm{Occ}_k(\mathrm{SA}[i]) = \{\mathrm{SA}[i]\}$, i.e., we must have $j' = \mathrm{SA}[i]$.

Our goal is to show how to implement a single step of the above process. Let $\ell \in [16..n)$ and $i \in [1..n]$. Suppose that we are given $(b,e) = (\mathrm{RBeg}_\ell(\mathrm{SA}[i]), \mathrm{REnd}_\ell(\mathrm{SA}[i]))$ and some $j \in \mathrm{Occ}_\ell(\mathrm{SA}[i])$ as input. We aim to show that under specific assumptions about the ability to perform some queries, we can compute some $j' \in \mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$ and the pair $(\mathrm{RBeg}_{2\ell}(\mathrm{SA}[i]), \mathrm{REnd}_{2\ell}(\mathrm{SA}[i]))$.

Let $\tau := \lfloor \frac{\ell}{3} \rfloor$. Our algorithm works differently, depending on whether it holds $\mathrm{SA}[i] \in \mathrm{R}(\tau, T)$ or $\mathrm{SA}[i] \in [1..n] \setminus \mathrm{R}(\tau, T)$. Thus, we first need to efficiently implement this check. Observe that whether or not it holds $j \in \mathrm{R}(\tau, T)$ depends only on $T[j..j+3\tau-1)$. Therefore, by $3\tau - 1 \le \ell$, if $j \in \mathrm{Occ}_\ell(\mathrm{SA}[i])$ then $\mathrm{SA}[i] \in \mathrm{R}(\tau, T)$ holds if and only if $j \in \mathrm{R}(\tau, T)$. Consequently, we can determine if $\mathrm{SA}[i] \in \mathrm{R}(\tau, T)$ (such $\mathrm{SA}[i]$ is called *periodic*) or $\mathrm{SA}[i] \in [1..n] \setminus \mathrm{R}(\tau, T)$ (i.e., the position $\mathrm{SA}[i]$ is *nonperiodic*) using any $j \in \mathrm{Occ}_\ell(\mathrm{SA}[i])$.

The Nonperiodic Positions. Assume $\mathrm{SA}[i] \in [1..n] \setminus \mathrm{R}(\tau, T)$. We proceed in two steps. First, we show how to compute $|\mathrm{Pos}_\ell(\mathrm{SA}[i])|$ and $|\mathrm{Occ}_{2\ell}(\mathrm{SA}[i])|$ assuming we have some $j' \in \mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$. By the earlier observation, this yields $(\mathrm{RBeg}_{2\ell}(\mathrm{SA}[i]), \mathrm{REnd}_{2\ell}(\mathrm{SA}[i]))$. We then explain how to find j'.

Let $j' \in \mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$. By $T^\infty[j'..j'+2\ell) = T^\infty[\mathrm{SA}[i]..\mathrm{SA}[i]+2\ell)$, we have that $|\mathrm{Pos}_\ell(\mathrm{SA}[i])| = |\mathrm{Pos}_\ell(j')|$, $|\mathrm{Occ}_{2\ell}(\mathrm{SA}[i])| = |\mathrm{Occ}_{2\ell}(j')|$, and $j' \in [1..n] \setminus \mathrm{R}(\tau, T)$. We can thus focus on computing $|\mathrm{Pos}_\ell(j')|$ and $|\mathrm{Occ}_{2\ell}(j')|$. Let S be any τ-synchronizing set of T. First, observe that by $j' \notin \mathrm{R}(\tau, T)$, the position $s' = \mathrm{succ}_\mathrm{S}(j')$ (see Section 5.2 for the definition of succ_S) satisfies $s' - j' < \tau$. Thus, by the consistency of S and $3\tau \le \ell$, all $j'' \in \mathrm{Pos}_\ell(j')$ share a common offset $\delta_\mathrm{S} = s' - j'$ such that $j'' + \delta_\mathrm{S} = \min(\mathrm{S} \cap [j''..j''+\tau))$ and hence the relative lexicographical order between $T[j''..n]$ and $T[j'..n]$ is the same as between $T[j''+\delta_\mathrm{S}..n]$ and $T[j'+\delta_\mathrm{S}..n]$. Thus, to compute $|\mathrm{Pos}_\ell(j')|$ it suffices to count $s'' \in \mathrm{S}$ that are:

(1) Preceded in T by the string $T[j'..s')$, and
(2) For which it holds $T[s''..n] \prec T[s'..n]$ and $\mathrm{LCE}_T(s'', s') \in [\ell - \delta_\mathrm{S}..2\ell - \delta_\mathrm{S})$.

Observe, that for any $q \ge 2\ell$, Condition 1 is equivalent to position s'' having a reversed left length-q context in the lexicographical range $[X..X')$, where $\overline{X} = T[j'..s')$ and $X' = Xc^\infty$ (where $c = \max \Sigma$), and Condition 2 is equivalent to position s'' having a right length-q context in $[Y..Y')$, where $Y = T^\infty[s'..j'+\ell)$ and $Y' = T^\infty[s'..j'+2\ell)$ (Lemma 5.2). Consequently, the only queries needed to compute $|\mathrm{Pos}_\ell(j')|$ are succ_S and range queries on a labelled set of points $\mathcal{P} = \{(\overline{T^\infty[s'-q..s')}, T^\infty[s'..s'+q), s') : s' \in \mathrm{S}\}$. Since $\tau = \lfloor \frac{\ell}{3} \rfloor$ and $\ell \ge 16$, it suffices to choose $q = 7\tau$ to satisfy $q \ge 2\ell$. Thus, under Assumption 5.3, we can efficiently compute $|\mathrm{Pos}_\ell(j')| = |\mathrm{Pos}_\ell(\mathrm{SA}[i])|$.

The intuition for $|\mathrm{Occ}_{2\ell}(j')|$ is similar, except we observe that Condition 2 is that s'' satisfies $\mathrm{LCE}_T(s'', s') \ge 2\ell - \delta_\mathrm{S}$, which is equivalent to s'' having a right length-q context in $[Y'..Y'c^\infty)$. Thus, we can also count such s'' (and consequently, compute the value $|\mathrm{Occ}_{2\ell}(\mathrm{SA}[i])|$) using $\mathcal{P}$.

The above reductions are proved in Lemmas 5.5 and 5.6 and lead to the following result.

Proposition 3.1. *Let $i \in [1..n]$ be such that $\mathrm{SA}[i] \in [1..n] \setminus \mathrm{R}(\tau, T)$. Under Assumption 5.3, given a position $j' \in \mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$, we can efficiently compute $|\mathrm{Pos}_\ell(\mathrm{SA}[i])|$ and $|\mathrm{Occ}_{2\ell}(\mathrm{SA}[i])|$.*

It remains to show how to find some $j' \in \mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$. For this, observe that if we sort all $j'' \in \mathrm{Occ}_\ell(\mathrm{SA}[i])$ by their right length-2ℓ context in T^∞ then for the kth position j'' in this order we have $T^\infty[j''..j''+2\ell) = T^\infty[\mathrm{SA}[b+k]..\mathrm{SA}[b+k]+2\ell)$, since $\mathrm{SA}(b..e]$ also contains all $j'' \in \mathrm{Occ}_\ell(\mathrm{SA}[i])$ sorted by their length-2ℓ right context, although potentially in a different order. Note, however, that $j' \in \mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$ only requires $T^\infty[j'..j'+2\ell) = T^\infty[\mathrm{SA}[i]..\mathrm{SA}[i]+2\ell)$. Thus, the ability to find the $(i-b)$th element in the sequence of all $j'' \in \mathrm{Occ}_\ell(\mathrm{SA}[i])$ sorted by $T^\infty[j''..j''+2\ell)$ (with ties resolved arbitrarily) is all we need to compute some $j' \in \mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$. Recall, that to show a common offset δ_S, we used the fact that suffixes shared a prefix of length at least 3τ. By $3\tau \le \ell$, here we also have that all $j'' \in \mathrm{Occ}_\ell(\mathrm{SA}[i])$ share a common offset $\delta_\mathrm{S} = \mathrm{succ}_\mathrm{S}(\mathrm{SA}[i]) - \mathrm{SA}[i]$ such that $j'' + \delta_\mathrm{S} = \min(\mathrm{S} \cap [j''..j''+\tau))$. Consequently, to find j' we take some $q \ge 2\ell$ and:

(1) First, letting $\delta_\mathrm{S} = \mathrm{succ}_\mathrm{S}(\mathrm{SA}[i]) - \mathrm{SA}[i]$, we compute the number m of positions $s' \in \mathrm{S}$ that have their reversed left length-q context in the lexicographic range $[X..X')$ (where $\overline{X} = T[\mathrm{SA}[i]..\mathrm{SA}[i]+\delta_\mathrm{S})$ and $X' = Xc^\infty$) and right length-q context in $[\varepsilon..Y)$ (where $Y = T^\infty[\mathrm{SA}[i]+\delta_\mathrm{S}..\mathrm{SA}[i]+\ell)$).
(2) Then, for any $k \in [1..|\mathrm{Occ}_\ell(\mathrm{SA}[i])|]$, the $(m+k)$th element s' in the sequence of all positions from S sorted by the length-q right context that simultaneously have their reversed left length-q context in $[X..X')$, satisfies $T^\infty[s'-\delta_\mathrm{S}..s'-\delta_\mathrm{S}+2\ell) = T^\infty[\mathrm{SA}[b+k]..\mathrm{SA}[b+k]+2\ell)$. In particular, the position s' for $k = i - b$ satisfies $T^\infty[s'-\delta_\mathrm{S}..s'-\delta_\mathrm{S}+2\ell) = T^\infty[\mathrm{SA}[i]..\mathrm{SA}[i]+2\ell)$, i.e., $s' - \delta_\mathrm{S} \in \mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$. Thus, finding j' reduces to a range selection query on $\mathcal{P}$.

The above reduction is proved in Lemma 5.9. One last detail is that we need X, Y, and δ_S. We note, however, that they all depend only on $T^\infty[\mathrm{SA}[i]..\mathrm{SA}[i]+\ell)$, and thus can be computed using $j \in \mathrm{Occ}_\ell(\mathrm{SA}[i])$ (which we have as input). We have thus proved the following result, which combined with Proposition 3.1 concludes the description of the SA query for nonperiodic $\mathrm{SA}[i]$.

Proposition 3.2. *Let $i \in [1..n]$ be such that $\mathrm{SA}[i] \in [1..n] \setminus \mathrm{R}(\tau, T)$. Under Assumption 5.3, given i and $j \in \mathrm{Occ}_\ell(\mathrm{SA}[i])$ as well as the pair $(\mathrm{RBeg}_\ell(\mathrm{SA}[i]), \mathrm{REnd}_\ell(\mathrm{SA}[i]))$, we can efficiently find some position $j' \in \mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$.*

The Periodic Positions. Assume $\mathrm{SA}[i] \in \mathrm{R}(\tau, T)$. The standard way to introduce structure among periodic positions (see, e.g., [40]) is as follows. Note that if $j, j{+}1 \in \mathrm{R}(\tau, T)$, then $\mathrm{per}(T[j..j{+}3\tau{-}1)) = \mathrm{per}(T[j{+}1..j{+}1{+}3\tau{-}1))$. This implies that any maximal block of positions in $\mathrm{R}(\tau, T)$ defines a highly periodic fragment of T (called a "run") with an associated period p. To compare runs, we "anchor" each run $T[r_b..r_e)$ by selecting some $H \in \Sigma^p$, called its *root*, so that $T[r_b..r_e)$ is a substring of H^∞ and no nontrivial rotation of H is selected for other runs. Then, every sufficiently long right-maximal fragment $T[j..j')$ of some run can be uniquely written as $T[j..j') = H'H^kH''$, where H is some root, $k \geq 1$, and H' (resp. H'') is a proper suffix (resp. prefix) of H. The value k is called the *exponent* of j, and is denoted $\exp(j) = k$. We then classify $\mathrm{type}(j) = -1$ if $T[j'] \prec T[j' - |H|]$ and $\mathrm{type}(j) = +1$ otherwise.

The first major challenge in the periodic case is selecting roots efficiently. An easy solution in the static case (e.g. [40, 42]) is to choose the lexicographically smallest rotation of H (known as the *Lyndon root*). This seems very difficult in the dynamic case, however. We instead show a construction that exploits the presence of symmetry-breaking component in the signature parsing (i.e., the deterministic coin tossing [21]) to develop a custom computation of roots.

Recall that $b = \mathrm{RBeg}_\ell(\mathrm{SA}[i])$ and $e = \mathrm{REnd}_\ell(\mathrm{SA}[i])$, and observe that all $i' \in (b..e]$ with $\mathrm{type}(\mathrm{SA}[i']) = -1$ precede those with $\mathrm{type}(\mathrm{SA}[i']) = +1$. Moreover, the values $\exp(\mathrm{SA}[i'])$ among those with $\mathrm{type}(\mathrm{SA}[i']) = -1$ (resp. $\mathrm{type}(\mathrm{SA}[i']) = +1$) are non-decreasing (resp. non-increasing) as we increase i'. This structure has led to a relatively straightforward processing of periodic positions in previous applications (e.g., the BWT construction in [40]), where the positions were first grouped by the type, and then by the exponent. In our case, however, we need to exclude the positions with very small and very large exponents. We thus employ the following modification to the above scheme: Rather than computing $|\mathrm{Pos}_\ell(\mathrm{SA}[i])|$ in one step, we prove that $\mathrm{Pos}_\ell(\mathrm{SA}[i])$ can be expressed as a combination of three sets, $\mathrm{Pos}_\ell^{\mathrm{low}}(\mathrm{SA}[i])$ (containing exponents "truncated" at length ℓ), $\mathrm{Pos}_\ell^{\mathrm{high}}(\mathrm{SA}[i])$ (where truncation occurs for length 2ℓ), and $\mathrm{Pos}_\ell^{\mathrm{mid}}(\mathrm{SA}[i])$ (all exponents in between); see Fig. 2 for an example. In a sequence of steps, we then compute the following values: $\mathrm{type}(\mathrm{SA}[i])$, the size $|\mathrm{Pos}_\ell^{\mathrm{low}}(\mathrm{SA}[i])|$, the exponent $\exp(\mathrm{SA}[i])$, the size $|\mathrm{Pos}_\ell^{\mathrm{mid}}(\mathrm{SA}[i])|$, a position $j' \in \mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$, the size $|\mathrm{Pos}_\ell^{\mathrm{high}}(\mathrm{SA}[i])|$, and the size $|\mathrm{Occ}_{2\ell}(\mathrm{SA}[i])|$ (the details are described in the full version of this paper [43]). Finally, we derive $|\mathrm{Pos}_\ell(\mathrm{SA}[i])|$, which equals $|\mathrm{Pos}_\ell^{\mathrm{low}}(\mathrm{SA}[i])| + |\mathrm{Pos}_\ell^{\mathrm{mid}}(\mathrm{SA}[i])| - |\mathrm{Pos}_\ell^{\mathrm{high}}(\mathrm{SA}[i])|$ if $\mathrm{type}(\mathrm{SA}[i]) = -1$, as well as $\mathrm{RBeg}_{2\ell}(\mathrm{SA}[i]) = \mathrm{RBeg}_\ell(\mathrm{SA}[i]) + |\mathrm{Pos}_\ell(\mathrm{SA}[i])|$ and, lastly, the value $\mathrm{REnd}_{2\ell}(\mathrm{SA}[i]) = \mathrm{RBeg}_{2\ell}(\mathrm{SA}[i]) + |\mathrm{Occ}_{2\ell}(\mathrm{SA}[i])|$. The case of $\mathrm{type}(\mathrm{SA}[i]) = +1$ is symmetric: we then compute $|\mathrm{Pos}'_\ell(\mathrm{SA}[i])|$ instead of $|\mathrm{Pos}_\ell(\mathrm{SA}[i])|$.

Dynamic Text Implementation. In the second part of the paper (Section 6), we develop a data structure that maintains a dynamic text (subject to updates listed below) and provides efficient implementation of the auxiliary queries specified in the assumptions of Section 5.

Definition 3.3. We say that a *dynamic text* over an integer alphabet Σ (with $\$ = \min \Sigma$) is a data structure that maintains a text $T \in \Sigma^+$ subject to the following updates:

- initialize(σ): Given the alphabet size σ, initialize the data structure, setting $T := \$$;
- insert(i, a): Given $i \in [1..|T|]$ and $a \in \Sigma \setminus \{\$\}$, set $T := T[1..i) \cdot a \cdot T[i..|T|]$.
- delete(i): Given $i \in [1..|T|)$, set $T := T[1..i) \cdot T(i..|T|]$.
- swap(i, j, k): Given $i, j, k \in [1..|T|]$ with $i \leq j \leq k$, set $T := T[1..i) \cdot T[j..k) \cdot T[i..j) \cdot T[k..|T|]$.

Observe that our interface does not directly support character substitutions; this is because substitute(i, a) can be implemented as delete(i) followed by insert(i, a). Moreover, note that the interface enforces the requirement of Section 5 that $\{i \in [1..|T|] : T[i] = \$\} = \{|T|\}$.

Various components of our data structure, described in Section 6, maintain auxiliary information associated to individual positions of the text T (such as whether the position belongs to a synchronizing set). Individual updates typically alter the positions of many characters in T (for example, insertion moves the character at position j to position $j + 1$ for each $j \in [i..|T|]$), so addressing the characters by their position is volatile. To address this issue, we use a formalism of *labelled strings*, where each character is associated to a unique label that is fixed throughout character's lifetime. This provides a clean realization of the concept of *pointers to characters*, introduced in [3] along with $O(\log n)$-time conversion between labels (pointers) and positions.

The main challenge that arises in our implementation of a dynamic text is to maintain synchronizing sets. As for the succ_S queries alone, we could generate synchronizing positions at query time. Since, however, Assumption 5.3 also entails supporting range queries over a set of points in one-to-one correspondence with the synchronizing positions, we cannot evade robust maintenance.

Dynamic Synchronizing Sets. By the consistency condition in Definition 2.1, whether a position j belongs to a synchronizing set S is decided based on the right context $T[j..j + 2\tau)$. This means that the entire synchronizing set can be described using a family $\mathcal{F} \subseteq \Sigma^{2\tau}$ such that $j \in \mathrm{S}$ if and only if $T[j..j + 2\tau) \in \mathcal{F}$. The construction algorithms in [40] select such $\mathcal{F}$ adaptively (based on the contents of T) to guarantee that $|\mathrm{S}|$ is small. In a dynamic scenario, we could either select $\mathcal{F}$ non-adaptively and keep it fixed; or adaptively modify $\mathcal{F}$ as the text T changes.

The second approach poses a very difficult task: the procedure maintaining $\mathcal{F}$ does not know the future updates, yet it needs to be robust against any malicious sequence of updates that an adversary could devise. This is especially hard in the deterministic setting, where we cannot hide $\mathcal{F}$ from the adversary. Thus, we aim for non-adaptivity, which comes at the price of increasing the synchronizing set size by a factor of $O(\log^*(\sigma\tau))$. On the positive side, a non-adaptive choice of $\mathcal{F}$ means that S only undergoes local changes; for example, a substitution of $T[i]$ may only affect $\mathrm{S} \cap (i - 2\tau..i]$. Moreover, since $\mathcal{F}$ yields small synchronizing sets

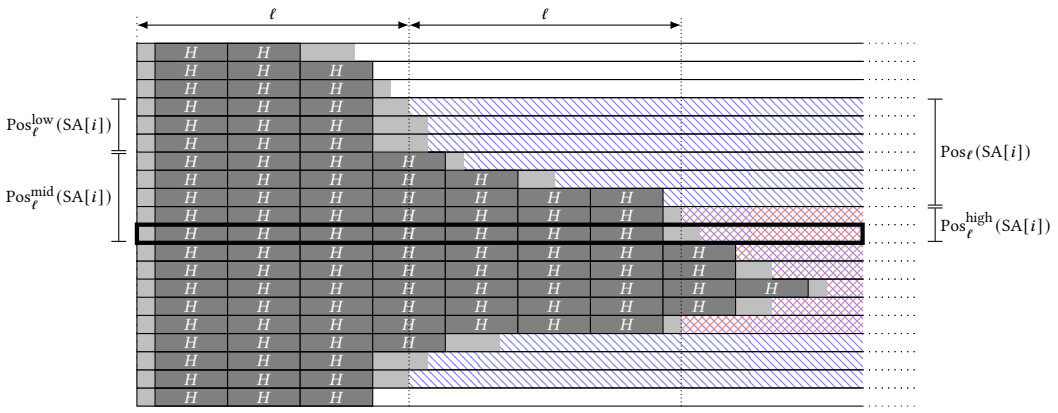

Figure 2: An example showing the sets $\mathrm{Pos}_\ell(\mathrm{SA}[i])$, $\mathrm{Pos}_\ell^{\mathrm{low}}(\mathrm{SA}[i])$, $\mathrm{Pos}_\ell^{\mathrm{mid}}(\mathrm{SA}[i])$, and $\mathrm{Pos}_\ell^{\mathrm{high}}(\mathrm{SA}[i])$. Note that $|\mathrm{Pos}_\ell(\mathrm{SA}[i])| = |\mathrm{Pos}_\ell^{\mathrm{low}}(\mathrm{SA}[i])| + |\mathrm{Pos}_\ell^{\mathrm{mid}}(\mathrm{SA}[i])| - |\mathrm{Pos}_\ell^{\mathrm{high}}(\mathrm{SA}[i])|$. The suffix $T[\mathrm{SA}[i]..n]$ is highlighted in bold. The sets $\mathrm{Occ}_\ell(\mathrm{SA}[i])$ and $\mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$ are marked with blue and red (respectively).

for all strings, this is in particular true for all substrings $T[i..j)$, whose synchronizing sets are in one-to-one correspondence with $\mathsf{S} \cap [i..j - 2\tau]$. This means that S is *locally sparse* and that each update incurs $O(\log^*(\sigma\tau))$ changes to S.

The remaining challenge is thus to devise a non-adaptive synchronizing set construction. Although all existing constructions are adaptive, Birenzwige, Golan, and Porat [11] provided a non-adaptive construction of a related notion of *partitioning sets*. While partitioning sets and synchronizing sets satisfy similar consistency conditions, the density condition of synchronizing sets is significantly stronger, which is crucial for a clean separation between periodic and nonperiodic positions. Thus, we strengthen the construction of [11] so that it produces synchronizing sets. This boils down to 'fixing' the set in the vicinity of positions in $\mathsf{R}(\tau, T)$.

Dynamic Strings over Balanced Signature Parsing. Unfortunately, the approach of [11] only comes with a static implementation. Thus, we need to dive into their techniques and provide an efficient dynamic implementation. Their central tool is a locally consistent parsing algorithm that iteratively parses the text using deterministic coin tossing [21] to determine phrase boundaries. Similar techniques have been used many times (see e.g. [3, 51, 58, 61, 62]), but the particular flavor employed in [11] involves a mechanism that, up to date, has not been adapted to the dynamic setting. Namely, as subsequent levels of the parsing provide coarser and coarser partitions into phrases, the procedure grouping phrases into blocks (to be merged in the next level) takes into account the lengths of the phrases, enforcing very long phrases to form single-element blocks. This trick makes the phrase lengths much more balanced, which is crucial in controlling the context size that governs the local consistency of the synchronizing sets derived from the parsing.

We thus proceed as follows. First, we develop *balanced signature parsing*: a version of signature parsing (originating from the early works on dynamic strings [3, 51]) that involves the phrase balancing mechanism. Then, we provide a dynamic strings implementation based on the balanced signature parsing. The main difference compared to the previous work [3, 30, 51] stems from the fact that the size of the *context-sensitive* part of the parsing (that may change depending on the context surrounding a given substring) is bounded in terms of the number of individual letters rather than the number of phrases at the respective level of the parsing. Due to this, we need to provide new (slightly modified) implementations of the basic operations (such as updates and longest common prefix queries). However, we also benefit from this feature, obtaining faster running times for more advanced queries, such as the period queries (which we utilize to retrieve $\mathsf{R}(\tau, T)$) compared to solutions using existing dynamic strings implementations [15].

4 GENERALIZED RANGE COUNTING AND SELECTION QUERIES

In this section we introduce generalized range counting and selection queries. The generalization lies in the fact that the "coordinates" of points can come from any ordered set. In particular, coordinates of points in some of our structures will be elements of Σ^* (i.e., the strings over alphabet Σ). Furthermore, the points in our data structures are labelled with distinct integer identifiers; we allow multiple points with the same coordinates, though.

The section is organized as follows. We start with the definition of range counting/selection queries. We then present an instance of the problem that will be of interest to us.

Let $\mathcal{X}$ and $\mathcal{Y}$ be some linearly ordered sets (we denote the order on both sets using $\prec$ or $\preceq$). Let $\mathcal{P} \subseteq \mathcal{X} \times \mathcal{Y} \times \mathbb{Z}$ be a finite set of points with distinct integer labels. We define the notation for *range counting* and *range selection* queries as follows.

Range counting query: Given $X_l, X_u \in \mathcal{X}$ and $Y_u \in \mathcal{Y}$, return $\mathsf{r\text{-}count}_{\mathcal{P}}(X_l, X_u, Y_u) := |\{(X, Y, \ell) \in \mathcal{P} : X_l \preceq X \prec X_u \text{ and } Y \prec Y_u\}|$. We denote $\mathsf{r\text{-}count}^{\mathrm{inc}}_{\mathcal{P}}(X_l, X_u, Y_u) := |\{(X, Y, \ell) \in \mathcal{P} : X_l \preceq X \prec X_u \text{ and } Y \preceq Y_u\}|$ and $\mathsf{r\text{-}count}_{\mathcal{P}}(X_l, X_u) := |\{(X, Y, \ell) \in \mathcal{P} : X_l \preceq X \prec X_u\}|$.

Range selection query: Given $X_l, X_u \in \mathcal{X}$ and an integer $r \in [1..\mathsf{r\text{-}count}_{\mathcal{P}}(X_l, X_u)]$, return any $\ell \in \mathsf{r\text{-}select}_{\mathcal{P}}(X_l, X_u, r) := \{\ell \in \mathbb{Z} : (X, Y, \ell) \in \mathcal{P}, X_l \preceq X \prec X_u, \text{ and } Y = Y_u\}$ for $Y_u \in \mathcal{Y}$ satisfying $r \in (\mathsf{r\text{-}count}_{\mathcal{P}}(X_l, X_u, Y_u)..\mathsf{r\text{-}count}^{\mathrm{inc}}_{\mathcal{P}}(X_l, X_u, Y_u)]$.

Theorem 4.1. *Suppose that the elements of $\mathcal{X}$ and $\mathcal{Y}$ can be compared in $O(t)$ time. Then, there is a deterministic data structure that maintains a set $\mathcal{P} \subseteq \mathcal{X} \times \mathcal{Y} \times [0..2^w)$ of size n, with insertions in $O((t+\log n)\log n)$ time and deletions in $O(\log^2 n)$ time so that range queries are answered in $O((t+\log^2 n)\log n)$ time.*

Proof. The set $\mathcal{P}$ is stored in a data structure of [46] (see also [16, 66]) for dynamic range counting queries; this component supports updates and queries in $O(\log^2 n)$ assuming $O(1)$-time comparisons. As for range selection queries, we resort to binary search with range counting queries as an oracle (the universe searched consists of the second coordinates of all points in $\mathcal{P}$). Thus, range selection queries cost $O(\log^3 n)$ time.

In order to substantiate the assumption on constant comparison time, we additionally maintain $\mathcal{P}$ in two instances of the order-maintenance data structure [22, 44], with points (X, Y, ℓ) ordered according to X in the first instance and according to Y in the second instance (ties are resolved arbitrarily). This allows for $O(1)$-time comparisons between points in $\mathcal{P}$. The overhead for deletions is $O(1)$, but insertions to the order-maintenance structure require specifying the predecessor of the newly inserted element; we find the predecessor in $O(t\log n)$ time using binary search. Similarly, at query time, we temporarily add the query coordinates $(X_l, X_u,$ and, if specified, $Y_u)$ to the appropriate order-maintenance structure, also at the cost of an extra $O(t\log n)$ term in the query time. □

A String-String Instance. We now present an instance of the above problem that will be used in our structure.

Definition 4.2. Let $T \in \Sigma^n$. For any $q \in \mathbb{Z}_+$ and $\mathsf{P} \subseteq [1..n]$, let $\mathrm{Points}_q(T, \mathsf{P}) := \{(\overline{T^\infty[p-q\,..\,p)}, T^\infty[p\,..\,p+q), p) : p \in \mathsf{P}\}$.

In other words, $\mathrm{Points}_q(T, \mathsf{P})$ is the collection of string-pairs (X, Y) composed of reversed length-q left context and a length-q right context (in T^∞) of every $p \in \mathsf{P}$, and for any $(X, Y, \ell) \in \mathrm{Points}_q(T, \mathsf{P})$, the label $\ell \in \mathsf{P}$ is the underlying position. Equivalently, $\mathrm{Points}_q(T, \mathsf{P})$ can be interpreted as a set of labelled points (X, Y, ℓ) with coordinates X, Y and an integer label ℓ, where the order on the $\mathcal{X} = \Sigma^*$ and $\mathcal{Y} = \Sigma^*$ axis is lexicographical.

Next, we define the problem of supporting range counting and selection queries on $\mathrm{Points}_q(T, \mathsf{P})$ for some special family of queries that can be succinctly represented as substrings of T^∞ or $\overline{T^\infty}$.

Problem 4.3 (String-String Range Queries). Let $T \in \Sigma^n$, $q \in [1..3n]$, and $\mathsf{P} \subseteq [1..n]$ be a set satisfying $|\mathsf{P}| = m$. Denote $\mathcal{P} = \mathrm{Points}_q(T, \mathsf{P})$ and $c = \max\Sigma$. Provide efficient support for the following queries:

(1) Given a position $i \in [1..n]$ and $q_l, q_r \in [0..2n]$, return $\text{r-count}_{\mathcal{P}}(X_l, X_u, Y_l)$ and $\text{r-count}_{\mathcal{P}}(X_l, X_u, Y_u)$, where $\overline{X_l} = T^\infty[i-q_l..i)$, $X_u = X_l c^\infty$, $Y_l = T^\infty[i..i+q_r)$, and $Y_u = Y_l c^\infty$,
(2) Given $i \in [1..n]$, $q_l \in [0..2n]$, and $r \in [1..\text{r-count}_{\mathcal{P}}(X_l, X_u)]$ (where $\overline{X_l} = T^\infty[i-q_l..i)$ and $X_u = X_l c^\infty$), return some position $p \in \text{r-select}_{\mathcal{P}}(X_l, X_u, r)$.

5 SA QUERY ALGORITHM

Let $T \in \Sigma^n$. In this section we show that under some small set of assumptions about the ability to perform queries on string synchronizing sets [40], we can perform SA queries for T.

Assumption 5.1. *For any $i \in [1..n]$, we can compute some $j \in \mathrm{Occ}_{16}(\mathrm{SA}[i])$ and the pair $(\mathrm{RBeg}_{16}(\mathrm{SA}[i]), \mathrm{REnd}_{16}(\mathrm{SA}[i]))$ in $O(t)$ time.*

Let us fix some $\ell \in [16..n)$. As outlined in Section 3, to answer the SA query, we need to show how given $i \in [1..n]$ along with some $j \in \mathrm{Occ}_\ell(\mathrm{SA}[i])$ and the pair $(\mathrm{RBeg}_\ell(\mathrm{SA}[i]), \mathrm{REnd}_\ell(\mathrm{SA}[i]))$ as input, to compute some $j' \in \mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$ as well as the pair $(\mathrm{RBeg}_{2\ell}(\mathrm{SA}[i]), \mathrm{REnd}_{2\ell}(\mathrm{SA}[i]))$. Due to space constraints, we will present the algorithm only for nonperiodic positions, i.e., $\mathrm{SA}[i] \in [1..n] \setminus \mathrm{R}(\lfloor\frac{\ell}{3}\rfloor, T)$ (Section 5.2). Handling of periodic positions is described in the full version of this paper [43] (see also Remark 5.13). All steps of the query algorithms are put together in Section 5.3.

5.1 Preliminaries

We start with a combinatorial result showing the three fundamental reductions. In the first and second, we show an equivalence between LCE queries for suffixes of T and comparisons of substrings of T. These results are used to characterize the set $\mathrm{Pos}_\ell(j)$ and its components. In the third reduction, we show an equivalence, where LCE queries are replaced with substring equalities. This is used to characterize the set $\mathrm{Occ}_{2\ell}(j)$.

Lemma 5.2 (♠). *Let $j \in [1..n]$ and $c = \max\Sigma$. Then:*

(1) If $0 \le \ell_1 < \ell_2 \le \ell_3$, then, for any $j' \in [1..n]$, the conjunction $T[j'..n] \prec T[j..n]$ and $\mathrm{LCE}_T(j, j') \in [\ell_1..\ell_2)$ holds if and only if $T^\infty[j..j+\ell_1) \le T^\infty[j'..j'+\ell_3) \prec T^\infty[j..j+\ell_2)$.
(2) If $0 \le \ell_1 \le \ell_2$, then, for any $j' \in [1..n]$, the disjunction $T[j'..n] \succeq T[j..n]$ or $\mathrm{LCE}_T(j, j') \ge \ell_1$ holds if and only if $T^\infty[j'..j'+\ell_2) \succeq T^\infty[j..j+\ell_1)$.
(3) If $0 \le \ell_1 \le \ell_2$, then, for any $j' \in [1..n]$, $T^\infty[j'..j'+\ell_1) = T^\infty[j..j+\ell_1)$ holds if and only if $T^\infty[j..j+\ell_1) \le T^\infty[j'..j'+\ell_2) \prec T^\infty[j..j+\ell_1)c^\infty$.

5.2 The Nonperiodic Positions

Let $\tau = \lfloor\frac{\ell}{3}\rfloor$. In this section, we show that assuming that for some τ-synchronizing set S of T we can efficiently perform succ_S queries (with $\mathrm{succ}_\mathsf{S}(i) := \min\{j \in \mathsf{S} \cup \{n-2\tau+2\} : j \ge i\}$ for any $i \in [1..n-2\tau+1]$), and support some string-string range queries (Assumption 5.3), given a position $i \in [1..n]$ satisfying $\mathrm{SA}[i] \in [1..n] \setminus \mathrm{R}(\tau, T)$, along with the pair $(\mathrm{RBeg}_\ell(\mathrm{SA}[i]), \mathrm{REnd}_\ell(\mathrm{SA}[i]))$ and some $j \in \mathrm{Occ}_\ell(\mathrm{SA}[i])$ as input, we can efficiently compute the pair $(\mathrm{RBeg}_{2\ell}(\mathrm{SA}[i]), \mathrm{REnd}_{2\ell}(\mathrm{SA}[i]))$ and some $j' \in \mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$.

Assumption 5.3. *For some τ-synchronizing set S of T (where $\tau = \lfloor\frac{\ell}{3}\rfloor$) the queries $\mathrm{succ}_\mathsf{S}(i)$ (where $i \in [1..n-3\tau+1] \setminus \mathrm{R}(\tau, T)$) and the string-string range queries (Problem 4.3) for text T, integer $q = 7\tau$, and the set of positions $\mathsf{P} = \mathsf{S}$ can be supported in $O(t)$ time.*

Remark 5.4. Note that by $\ell < n$ and $3\tau \le \ell$, the value $q = 7\tau \le 2\ell + \tau < 3n$ in the above assumption satisfies the requirement of Problem 4.3. Note also that $\tau = \lfloor\frac{\ell}{3}\rfloor \le \lfloor\frac{\ell}{2}\rfloor \le \lfloor\frac{n}{2}\rfloor$, i.e., S is well-defined (see Definition 2.1).

The section is organized into three parts. First, we show how under Assumption 5.3 to combine the properties of synchronizing sets with reductions in Lemma 5.2 to compute the sizes of sets $\mathrm{Pos}_\ell(j)$ and $\mathrm{Occ}_{2\ell}(j)$ (Section 5.2.1). Then, in Section 5.2.2, we

show how under the same assumptions to efficiently compute some $j' \in \mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$. In Section 5.2.3, we put everything together.

5.2.1 Computing the Size of $\mathrm{Pos}_\ell(j)$ and $\mathrm{Occ}_{2\ell}(j)$. Let $j \in [1..n] \setminus \mathsf{R}(\tau, T)$. In this section, we show how under Assumption 5.3 to efficiently compute the values $|\mathrm{Pos}_\ell(j)|$ and $|\mathrm{Occ}_{2\ell}(j)|$.

The section is organized as follows. First, we present two combinatorial results (Lemmas 5.5 and 5.6) characterizing the sets $\mathrm{Pos}_\ell(j)$ and $\mathrm{Occ}_{2\ell}(j)$ using the string synchronizing set S. We then use these characterizations to prove a formula for the cardinality of these sets (Lemma 5.7). We conclude with Proposition 5.8 showing how under Assumption 5.3, to utilize this formula to quickly compute values $|\mathrm{Pos}_\ell(j)|$ and $|\mathrm{Occ}_{2\ell}(j)|$ given position j.

Lemma 5.5. *Let $j \in [1..n-3\tau+1] \setminus \mathsf{R}(\tau,T)$ and $s = \mathrm{succ}_\mathsf{S}(j)$. Let $X \in \Sigma^*$ and $Y, Y' \in \Sigma^+$ be such that $\overline{X} = T[j..s)$, $T^\infty[j..j+\ell) = \overline{X}Y$, and $T^\infty[j..j+2\ell) = \overline{X}Y'$. Then, for any $j' \in [1..n]$, letting $s' = j' + |X|$, it holds*

$j' \in \mathrm{Pos}_\ell(j)$ *if and only if*

$$s' \in \mathsf{S},\ Y \preceq T^\infty[s'..s'+7\tau) \prec Y',\ \text{and } T^\infty[s'-|X|..s') = \overline{X}.$$

Proof. By the uniqueness of $T[n] = \$$, it holds $\mathrm{per}(T[n-3\tau+2..n]) = 3\tau-1$ and hence $\mathsf{S} \cap [n-3\tau+2..n-2\tau+2) \neq \emptyset$. Thus, by $j < n-3\tau+2$, $s = \mathrm{succ}_\mathsf{S}(j)$ satisfies $s \in \mathsf{S}$. Moreover, by $j \notin \mathsf{R}(\tau,T)$, it holds $\mathsf{S} \cap [j..j+\tau) \neq \emptyset$. Therefore, we have $|X| = s - j < \tau \le \ell$, and hence the strings Y and Y' (of length $\ell - |X|$ and $2\ell - |X|$, respectively) are well-defined and nonempty.

Let $j' \in \mathrm{Pos}_\ell(j)$, i.e., $T[j'..n] \prec T[j..n]$ and $\mathrm{LCE}_T(j,j') \in [\ell..2\ell)$. This implies $j \neq j'$ and $T[j'..j'+\ell) = T[j..j+\ell)$. Therefore, by $\ell - (s-j) \ge \ell - \tau \ge 2\tau$ and the consistency of the string synchronizing set S (Definition 2.1) applied for positions $j_1 = j + t$ and $j_2 = j' + t$, where $t \in [0..|X|)$, we obtain $\mathrm{succ}_\mathsf{S}(j') - j' = \mathrm{succ}_\mathsf{S}(j) - j$ (or equivalently, $\mathrm{succ}_\mathsf{S}(j') = j' + (s-j) = s'$) and $s' \in \mathsf{S}$. Next, by $T[j'..j'+\ell) = T[j..j+\ell)$ and $|X| < \tau \le \ell$, it holds $\mathrm{LCE}_T(j,j') = |X| + \mathrm{LCE}_T(s,s')$. Thus, $\mathrm{LCE}_T(s,s') \in [\ell - |X|..2\ell-|X|)$. By Lemma 5.2(1) (with parameters $\ell_1 = |Y| = \ell - |X|$, $\ell_2 = |Y'| = 2\ell - |X|$, and $\ell_3 = 7\tau$) and $2\ell \le 7\tau$ (holding for $\ell \ge 16$), this implies $Y \preceq T^\infty[s'..s'+7\tau) \prec Y'$. Finally, the equality $T^\infty[j'..j'+\ell) = T^\infty[j..j+\ell) = \overline{X}Y$ implies $T^\infty[s'-|X|..s') = T^\infty[j'..s') = \overline{X}$.

For the opposite implication, assume $s' \in \mathsf{S}$, $Y \preceq T^\infty[s'..s'+7\tau) \prec Y'$, and $T^\infty[s'-|X|..s') = \overline{X}$. By Lemma 5.2(1) (with the same parameters as above) and $2\ell \le 7\tau$, this implies $T[s'..n] \prec T[s..n]$ and $\mathrm{LCE}_T(s,s') \in [\ell-|X|..2\ell-|X|)$. Since $T[j..s) = \overline{X}$ and by $s \in \mathsf{S}$ we have $s < n$, X does not contain the symbol \$, and hence $j' \ge 1$. Thus, $T^\infty[j'..s') = X$ implies $T[j..s) = T[j'..s')$, and consequently, $T[j'..n] \prec T[j..n]$ and $\mathrm{LCE}_T(j,j') = |X| + \mathrm{LCE}_T(s,s') \in [\ell..2\ell)$. Thus, $j' \in \mathrm{Pos}_\ell(j)$. □

Lemma 5.6. *Let $j \in [1..n-3\tau+1] \setminus \mathsf{R}(\tau,T)$, $s = \mathrm{succ}_\mathsf{S}(j)$, and $d \in [\ell..2\ell]$. Let $X \in \Sigma^*$ and $Y' \in \Sigma^+$ be such that $\overline{X} = T[j..s)$ and $T^\infty[j..j+d) = \overline{X}Y'$. Then, for any $j' \in [1..n]$, letting $s' = j' + |X|$ and $c = \max\Sigma$, it holds*

$j' \in \mathrm{Occ}_d(j)$ *if and only if*

$$s' \in \mathsf{S},\ Y' \preceq T^\infty[s'..s'+7\tau) \prec Y'c^\infty,\ \text{and } T^\infty[s'-|X|..s') = \overline{X}.$$

Proof. Similarly as in Lemma 5.5, we first observe that by $j < n-3\tau+2$, $s = \mathrm{succ}_\mathsf{S}(j)$ satisfies $s \in \mathsf{S}$. Moreover, by $j \notin \mathsf{R}(\tau,T)$, it holds $\mathsf{S} \cap [j..j+\tau) \neq \emptyset$. Therefore, $|X| = s - j < \tau \le \ell$, and hence the string Y' (of length $d - |X|$) is well-defined and nonempty.

Let $j' \in \mathrm{Occ}_d(j)$, i.e., $T^\infty[j'..j'+d) = T^\infty[j..j+d)$. To show $s' \in \mathsf{S}$, we consider two cases. If $j = j'$ then $s' = s \in \mathsf{S}$ holds by definition. Otherwise, by $T^\infty[j..j+d) = T^\infty[j'..j'+d)$ and the uniqueness of $T[n] = \$$, we must have $T[j..j+\ell) = T[j'..j'+\ell)$. Thus, by $\ell - (s-j) \ge \ell - \tau \ge 2\tau$ and the consistency of S (applied as in the proof of Lemma 5.5) for $j_1 = j + t$ and $j_2 = j' + t$, where $t \in [0..|X|)$, we obtain $\mathrm{succ}_\mathsf{S}(j') = s' \in \mathsf{S}$. Next, by $T^\infty[s'..j'+d) = T^\infty[s..j+d)$ and $d \le 2\ell \le 7\tau$, we obtain from Lemma 5.2(3) (with parameters $\ell_1 = |Y'| = d - |X|$ and $\ell_2 = 7\tau$) that $Y' \preceq T^\infty[s'..s'+7\tau) \prec Y'c^\infty$. Finally, $T^\infty[j'..j'+d) = T^\infty[j..j+d) = \overline{X}Y'$ implies $T^\infty[s'-|X|..s') = T^\infty[j'..s') = \overline{X}$, i.e., the third condition.

For the opposite implication, assume $s' \in \mathsf{S}$, $Y' \preceq T^\infty[s'..s'+7\tau) \prec Y'c^\infty$, and $T^\infty[s'-|X|..s') = \overline{X}$. By Lemma 5.2(3) (with the same parameters as above) and $d \le 2\ell \le 7\tau$, this implies $T^\infty[s'..j'+d) = T^\infty[s..j+d)$. Combining this with the assumption $T^\infty[j'..s') = \overline{X} = T^\infty[j..s)$, we obtain $T^\infty[j'..j'+d) = T^\infty[j..j+d)$, i.e., $j' \in \mathrm{Occ}_d(j)$. □

Lemma 5.7. *Let $j \in [1..n-3\tau+1] \setminus \mathsf{R}(\tau,T)$ and $s = \mathrm{succ}_\mathsf{S}(j)$. Let $X \in \Sigma^*$ and $X', Y, Y' \in \Sigma^+$ be such that $\overline{X} = T[j..s)$, $X' = Xc^\infty$ (with $c = \max\Sigma$), $T^\infty[j..j+\ell) = \overline{X}Y$, and $T^\infty[j..j+2\ell) = \overline{X}Y'$. Then, letting $q = 7\tau$ and $\mathcal{P} = \mathrm{Points}_q(T,\mathsf{S})$, it holds:*

(1) $|\mathrm{Pos}_\ell(j)| = \mathsf{r\text{-}count}_\mathcal{P}(X,X',Y') - \mathsf{r\text{-}count}_\mathcal{P}(X,X',Y)$ and
(2) $|\mathrm{Occ}_{2\ell}(j)| = \mathsf{r\text{-}count}_\mathcal{P}(X,X',Y'c^\infty) - \mathsf{r\text{-}count}_\mathcal{P}(X,X',Y')$.

Proof. 1. By Lemma 5.5, we can write $\mathrm{Pos}_\ell(j) = \{s' - |X| : s' \in \mathsf{S},\ Y \preceq T^\infty[s'..s'+7\tau) \prec Y',\ \text{and } T^\infty[s'-|X|..s') = \overline{X}\}$. On the other hand, by Definition 4.2 and the definition of the rcount query:

$$\begin{aligned}
&\mathsf{r\text{-}count}_\mathcal{P}(X,X',Y)\\
&= |\{s' \in \mathsf{S} : T^\infty[s'..s'+7\tau) \prec Y \text{ and } X \preceq \overline{T^\infty[s'-7\tau..s')} \prec X'\}|\\
&= |\{s' \in \mathsf{S} : T^\infty[s'..s'+7\tau) \prec Y \text{ and } X \text{ is a prefix of } \overline{T^\infty[s'-7\tau..s')}\}|\\
&= |\{s' \in \mathsf{S} : T^\infty[s'..s'+7\tau) \prec Y \text{ and } T^\infty[s'-|X|..s') = \overline{X}\}|.
\end{aligned}$$

Analogously, $\mathsf{r\text{-}count}_\mathcal{P}(X,X',Y') = |\{s' \in \mathsf{S} : T^\infty[s'..s'+7\tau) \prec Y' \text{ and } T^\infty[s'-|X|..s') = \overline{X}\}|$. Since any position $s' \in \mathsf{S}$ that satisfies $T^\infty[s'..s'+7\tau) \prec Y$ also satisfies $T^\infty[s'..s'+7\tau) \prec Y'$, we obtain $\mathsf{r\text{-}count}_\mathcal{P}(X,X',Y') - \mathsf{r\text{-}count}_\mathcal{P}(X,X',Y) = |\{s' \in \mathsf{S} : Y \preceq T^\infty[s'..s'+7\tau) \prec Y' \text{ and } T^\infty[s'-|X|..s') = \overline{X}\}|$. The cardinality of this set is clearly the same as the earlier set characterizing $\mathrm{Pos}_\ell(j)$. Thus, $\mathsf{r\text{-}count}_\mathcal{P}(X,X',Y') - \mathsf{r\text{-}count}_\mathcal{P}(X,X',Y) = |\mathrm{Pos}_\ell(j)|$.

2. By Lemma 5.6, we have $\mathrm{Occ}_{2\ell}(j) = \{s' - |X| : s' \in \mathsf{S},\ Y' \preceq T^\infty[s'..s'+7\tau) \prec Y'c^\infty,\ \text{and } T^\infty[s'-|X|..s') = \overline{X}\}$. On the other hand, by Definition 4.2 and the definition of rcount queries, we also have that $\mathsf{r\text{-}count}_\mathcal{P}(X,X',Y') = |\{s' \in \mathsf{S} : T^\infty[s'..s'+7\tau) \prec Y' \text{ and } T^\infty[s'-|X|..s') = \overline{X}\}|$ and $\mathsf{r\text{-}count}_\mathcal{P}(X,X',Y'c^\infty) = |\{s' \in \mathsf{S} : T^\infty[s'..s'+7\tau) \prec Y'c^\infty \text{ and } T^\infty[s'-|X|..s') = \overline{X}\}|$. Since any position $s' \in \mathsf{S}$ that satisfies $T^\infty[s'..s'+7\tau) \prec Y'$ also satisfies $T^\infty[s'..s'+7\tau) \prec Y'c^\infty$, we thus obtain $\mathsf{r\text{-}count}_\mathcal{P}(X,X',Y'c^\infty) - \mathsf{r\text{-}count}_\mathcal{P}(X,X',Y') = |\{s' \in \mathsf{S} : Y' \preceq T^\infty[s'..s'+7\tau) \prec Y'c^\infty \text{ and } T^\infty[s'-|X|..s') = \overline{X}\}|$. The cardinality of this set is clearly the same as the earlier set characterizing $\mathrm{Occ}_{2\ell}(j)$. We therefore obtain $\mathsf{r\text{-}count}_\mathcal{P}(X,X',Y'c^\infty) - \mathsf{r\text{-}count}_\mathcal{P}(X,X',Y') = |\mathrm{Occ}_{2\ell}(j)|$. □

Proposition 5.8. *Under Assumption 5.3, given a position $j \in [1..n] \setminus \mathsf{R}(\tau, T)$, we can compute $|\mathrm{Pos}_\ell(j)|$ and $|\mathrm{Occ}_{2\ell}(j)|$ in $O(t)$ time.*

Proof. We first check if $j > n - 3\tau + 1$. If yes, then by the uniqueness of $T[n] = \$$, it holds $|\mathrm{Occ}_\ell(j)| = 1$. By $\mathrm{Occ}_{2\ell}(j) \neq \emptyset$, $\mathrm{Occ}_{2\ell}(j) \subseteq \mathrm{Occ}_\ell(j)$, we can therefore return $|\mathrm{Pos}_\ell(j)| = 0$ and $|\mathrm{Occ}_{2\ell}(j)| = 1$. Let us thus assume $j \le n - 3\tau + 1$ and recall from the proof of Lemma 5.5 that then $s = \mathrm{succ}_{\mathsf{S}}(j)$ satisfies $s \in \mathsf{S}$. Using Assumption 5.3 we compute s. Let $X \in \Sigma^*$ and $X', Y, Y' \in \Sigma^+$ be such that $\overline{X} = T[j..s)$, $X' = Xc^\infty$, $T^\infty[j..j+\ell) = \overline{X}Y$, and $T^\infty[j..j+2\ell) = \overline{X}Y'$. Then:

(1) By Lemma 5.7, we have $|\mathrm{Pos}_\ell(j)| = \text{r-count}_{\mathcal{P}}(X, X', Y') - \text{r-count}_{\mathcal{P}}(X, X', Y)$ (where $\mathcal{P} = \mathrm{Points}_{7\tau}(T, \mathsf{S})$) which under Assumption 5.3 we can efficiently compute using the query arguments $(i, q_l, q_r) = (s, s - j, 2\ell - (s - j))$ and then with arguments $(i, q_l, q_r) = (s, s - j, \ell - (s - j))$ (see Problem 4.3). By $j \notin \mathsf{R}(\tau, T)$ and the density property of S, we have $s - j < \tau \le \ell < n$. On the other hand, $q_r \le 2\ell - (s - j) \le 2\ell < 2n$. Thus, the arguments q_l and q_r of both queries satisfy the requirements in Problem 4.3.

(2) By Lemma 5.7, we have $|\mathrm{Occ}_{2\ell}(j)| = \text{r-count}_{\mathcal{P}}(X, X', Y'c^\infty) - \text{r-count}_{\mathcal{P}}(X, X', Y')$ (where $\mathcal{P}$ is defined as above), which under Assumption 5.3 we can compute using the query arguments $(i, q_l, q_r) = (s, s - j, 2\ell - (s - j))$ (see Problem 4.3). As noted above, these arguments satisfy the requirements in Problem 4.3.

By Assumption 5.3, the query takes $O(t)$ time in total. □

5.2.2 Computing a Position in $\mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$. Assume that $i \in [1..n]$ satisfies $\mathrm{SA}[i] \in [1..n] \setminus \mathsf{R}(\tau, T)$. In this section, we show how under Assumption 5.3, given the index i along with values $\mathrm{RBeg}_\ell(\mathrm{SA}[i])$, $\mathrm{REnd}_\ell(\mathrm{SA}[i])$, and some position $j \in \mathrm{Occ}_\ell(\mathrm{SA}[i])$, to efficiently compute some position $j' \in \mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$.

The section is organized as follows. First, we present a combinatorial result (Lemma 5.9) that reduces the computation of $j' \in \mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$ to a generalized range selection query (see Section 4). We then use this reduction to present the query algorithm for the computation of some $j' \in \mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$ in Proposition 5.10.

Lemma 5.9. *Assume $i \in [1..n]$ is such that $\mathrm{SA}[i] \in [1..n - 3\tau + 1] \setminus \mathsf{R}(\tau, T)$. Denote $b = \mathrm{RBeg}_\ell(\mathrm{SA}[i])$, $d = |\mathrm{Occ}_\ell(\mathrm{SA}[i])|$, and $s = \mathrm{succ}_{\mathsf{S}}(\mathrm{SA}[i])$. Let $X \in \Sigma^*$ and $X', Y \in \Sigma^+$ be such that $\overline{X} = T[\mathrm{SA}[i]..s)$, $X' = Xc^\infty$ (with $c = \max\Sigma$), and $T^\infty[\mathrm{SA}[i]..\mathrm{SA}[i] + \ell) = \overline{X}Y$. Let also $\mathcal{P} = \mathrm{Points}_{7\tau}(T, \mathsf{S})$, $m = \text{r-count}_{\mathcal{P}}(X, X', Y)$, and $m' = \text{r-count}_{\mathcal{P}}(X, X')$. Then, $m + d \le m'$. Moreover:*

(1) For $\delta \in [1..d]$, any position $p \in \text{r-select}_{\mathcal{P}}(X, X', m + \delta)$ satisfies $T^\infty[p - |X|..p - |X| + 2\ell) = T^\infty[\mathrm{SA}[b + \delta]..\mathrm{SA}[b + \delta] + 2\ell)$.

(2) For $\delta = i - b$, any position $p \in \text{r-select}_{\mathcal{P}}(X, X', m + \delta)$ satisfies $p - |X| \in \mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$.

Proof. By the uniqueness of $T[n] = \$$, it holds $\mathrm{per}(T[n - 3\tau + 2..n]) = 3\tau - 1$ and hence $\mathsf{S} \cap [n - 3\tau + 2..n - 2\tau + 2) \neq \emptyset$. Thus, by $\mathrm{SA}[i] < n - 3\tau + 2$, $s = \mathrm{succ}_{\mathsf{S}}(\mathrm{SA}[i])$ satisfies $s \in \mathsf{S}$. Denote $q = |\mathsf{S}|$. Let $(a_j)_{j \in [1..q]}$ be a sequence containing all positions $p \in \mathsf{S}$ ordered according to the string $T^\infty[p..p + 7\tau)$. In other words, for any $j, j' \in [1..q]$, $j < j'$ implies $T^\infty[a_j..a_j + 7\tau) \le T^\infty[a_{j'}..a_{j'} + 7\tau)$. Note, that the sequence $(a_j)_{j \in [1..q]}$ is not unique. Since $\{a_j : j \in [1..q]\} = \mathsf{S}$, it holds $|\{a_j - |X| : j \in [1..q] \text{ and } T^\infty[a_j - |X|..a_j) = \overline{X}\}| = |\{j \in [1..q] : T^\infty[a_j - |X|..a_j) = \overline{X}\}| = |\{p \in \mathsf{S} : T^\infty[p - |X|..p) = \overline{X}\}| = m'$, where the last equality follows by Lemma 5.2(3) and the definition of $\text{r-count}_{\mathcal{P}}(X, X')$ (see Section 4). By the same argument (utilizing the definition of $\text{r-count}_{\mathcal{P}}(X, X', Y)$ instead of $\text{r-count}_{\mathcal{P}}(X, X')$), we have $|\{a_j - |X| : j \in [1..q], T^\infty[a_j - |X|..a_j) = \overline{X}, \text{ and } T^\infty[a_j..a_j + 7\tau) \prec Y\}| = |\{j \in [1..q] : T^\infty[a_j - |X|..a_j) = \overline{X} \text{ and } T^\infty[a_j..a_j + 7\tau) \prec Y\}| = m$. By Lemma 5.6, for any $j \in [1..n]$, it holds $j \in \mathrm{Occ}_\ell(\mathrm{SA}[i])$ if and only if $j + |X| \in \mathsf{S}$, $T^\infty[j..j + |X|) = \overline{X}$, and $Y \le T^\infty[j + |X|..j + |X| + 7\tau) \prec Yc^\infty$. In other words, $\mathrm{Occ}_\ell(\mathrm{SA}[i]) = \{a_j - |X| : j \in [1..q], T^\infty[a_j - |X|..a_j) = \overline{X}, \text{ and } Y \le T^\infty[a_j..a_j + 7\tau) \prec Yc^\infty\}$. The latter set (whose cardinality is equal to d) is clearly a subset of $\{a_j - |X| : j \in [1..q] \text{ and } T^\infty[a_j - |X|..a_j) = \overline{X}\}$ (whose cardinality, as shown above, is equal to m'). Thus, $d \le m'$. On the other hand, the set $\{a_j - |X| : j \in [1..q], T^\infty[a_j - |X|..a_j) = \overline{X}, \text{ and } T^\infty[a_j..a_j + 7\tau) \prec Y\}$ (whose cardinality, as shown above, is m) is also clearly a subset of $\{a_j - |X| : j \in [1..q] \text{ and } T^\infty[a_j - |X|..a_j) = \overline{X}\}$. Thus, $m \le m'$. Since $j \in [1..q]$ cannot simultaneously satisfy $T^\infty[a_j..a_j + 7\tau) \prec Y$ and $Y \le T^\infty[a_j..a_j + 7\tau)$, these subsets are disjoint. Hence, it follows that $m + d \le m'$.

1. As shown above, $|\{j \in [1..q] : T^\infty[a_j - |X|..a_j) = \overline{X}\}| = m'$. Let $(b_j)_{j \in [1..m']}$ be a subsequence of $(a_j)_{j \in [1..q]}$ containing all elements of $\{a_j : j \in [1..q] \text{ and } T^\infty[a_j - |X|..a_j) = \overline{X}\}$ (in the same order as they appear in the sequence $(a_j)_{j \in [1..q]}$). Our proof consists of three steps:

(i) Let $j \in [1..m']$. We start by showing that it holds $b_j \in \text{r-select}_{\mathcal{P}}(X, X', j)$. Let Q, Q' be such that $\overline{Q} = T^\infty[b_j - 7\tau..b_j)$ and $Q' = T^\infty[b_j..b_j + 7\tau)$. Let

$$r_{\mathrm{beg}} = |\{a_t : t \in [1..q], T^\infty[a_t - |X|..a_t) = \overline{X}, \text{ and } T^\infty[a_t..a_t + 7\tau) \prec Q'\}|;$$
$$r_{\mathrm{end}} = |\{a_t : t \in [1..q], T^\infty[a_t - |X|..a_t) = \overline{X}, \text{ and } T^\infty[a_t..a_t + 7\tau) \le Q'\}|.$$

If $t \in [1..q]$ satisfies $T^\infty[a_t - |X|..a_t) = \overline{X}$, then $a_t \in \{b_1, \ldots, b_{m'}\}$. Moreover, since for any $t, t' \in [1..m']$, $t < t'$ implies $T^\infty[b_t..b_t + 7\tau) \le T^\infty[b_{t'}..b_{t'} + 7\tau)$, any $t \in [1..q]$ that additionally satisfies $T^\infty[a_t..a_t + 7\tau) \prec T^\infty[b_j..b_j + 7\tau)$, also satisfies $a_t \in \{b_1, \ldots, b_{j-1}\}$. Thus, $r_{\mathrm{beg}} < j$. On the other hand, every $t \in [1..j]$ satisfies $T^\infty[b_t - |X|..b_t) = \overline{X}$ and $T^\infty[b_t..b_t + 7\tau) \le T^\infty[b_j..b_j + 7\tau)$. Thus, $j \le r_{\mathrm{end}}$. Altogether, $j \in (r_{\mathrm{beg}}..r_{\mathrm{end}}]$. Recall now the definition $\mathcal{P} = \mathrm{Points}_{7\tau}(T, \mathsf{S})$ (Definition 4.2) and note that by Lemma 5.2(3), we have $r_{\mathrm{beg}} = \text{r-count}_{\mathcal{P}}(X, X', Q')$ and $r_{\mathrm{end}} = \text{r-count}^{\mathrm{inc}}_{\mathcal{P}}(X, X', Q')$. Therefore, $j \in (\text{r-count}_{\mathcal{P}}(X, X', Q')..\text{r-count}^{\mathrm{inc}}_{\mathcal{P}}(X, X', Q')]$. On the other hand, $(Q, Q', b_j) \in \mathcal{P}$ and $T^\infty[b_j - |X|..b_j) = \overline{X}$, so $b_j \in \text{r-select}_{\mathcal{P}}(X, X', j)$ holds as claimed.

(ii) Let $j \in [1..m']$. We show that $T^\infty[p - |X|..p + 7\tau) = T^\infty[b_j - |X|..b_j + 7\tau)$ holds for any $p \in \text{r-select}_{\mathcal{P}}(X, X', j)$. By Item (i) and the definition of $\text{r-select}_{\mathcal{P}}(X, X', j)$, the choice of p implies $T^\infty[p..p + 7\tau) = T^\infty[b_j..b_j + 7\tau)$. Moreover, letting Q be such that $\overline{Q} = T^\infty[p - 7\tau..p)$, it also implies $X \le Q \prec X'$. Thus,

by Lemma 5.2(3), p is preceded by $\overline{X}$ in T. Since by definition of $(b_j)_{j\in[1..m']}$, the position b_j is also preceded by $\overline{X}$ in T, we obtain $T^\infty[p-|X|..p+7\tau) = T^\infty[b_j-|X|..b_j+7\tau)$.

(iii) We are now ready to prove the main claim. As observed above, $\mathrm{Occ}_\ell(\mathrm{SA}[i]) = \{a_j-|X| : j\in[1..q],\ T^\infty[a_j-|X|..a_j) = \overline{X},\ \text{and } Y \preceq T^\infty[a_j..a_j+7\tau) \prec Yc^\infty\}$. Note, that since the positions k in the sequence $(a_j)_{j\in[1..q]}$ are sorted by $T^\infty[k..k+7\tau)$, we can simplify the second condition. Denoting $j_{\mathrm{skip}} = |\{j\in[1..q] : T^\infty[a_j..a_j+7\tau)\prec Y\}|$, we have

$$\mathrm{Occ}_\ell(\mathrm{SA}[i]) = \left\{a_j-|X| : {}^{j\in(j_{\mathrm{skip}}..q],\ T^\infty[a_j-|X|..a_j)=\overline{X},}_{\text{and } T^\infty[a_j..a_j+7\tau)\prec Yc^\infty}\right\}.$$

Let us now estimate $|\{j\in[1..j_{\mathrm{skip}}] : T^\infty[a_j-|X|..a_j)=\overline{X}\}|$. Any j in this set satisfies $j\in[1..q]$, $T^\infty[a_j-|X|..a_j)=\overline{X}$, and $T^\infty[a_j..a_j+7\tau)\prec Y$. Earlier we observed that the number of such j is precisely m. Combining this fact with the above formula for $\mathrm{Occ}_\ell(\mathrm{SA}[i])$ and the definition of $(b_j)_{j\in[1..m']}$, we have $\mathrm{Occ}_\ell(\mathrm{SA}[i]) = \{b_j-|X| : j\in(m..m+d]\}$. On the other hand, we have $b+d = \mathrm{RBeg}_\ell(\mathrm{SA}[i])+|\mathrm{Occ}_\ell(\mathrm{SA}[i])| = \mathrm{REnd}_\ell(\mathrm{SA}[i])$. Therefore, $\mathrm{Occ}_\ell(\mathrm{SA}[i]) = \{\mathrm{SA}[j] : j\in(b..b+d]\}$. We now observe:

- Let $j_1, j_2\in(m..m+d]$ and assume $j_1<j_2$. Since the elements of (b_j) occur in the same order as in (a_j), and positions p in (a_j) are sorted by $T^\infty[p..p+7\tau)$, it follows that $T^\infty[b_{j_1}..b_{j_1}+7\tau)\preceq T^\infty[b_{j_2}..b_{j_2}+7\tau)$. On the other hand, by definition of (b_j), both positions b_{j_1} and b_{j_2} are preceded in T by the string $\overline{X}$. Thus, $T^\infty[b_{j_1}-|X|..b_{j_2}+7\tau)\preceq T^\infty[b_{j_2}-|X|..b_{j_2}+7\tau)$.
- On the other hand, by definition of lexicographical order, for any $j_1,j_2\in[1..d]$, the assumption $j_1<j_2$ implies $T^\infty[\mathrm{SA}[b+j_1]..\mathrm{SA}[b+j_1]+|X|+7\tau)\preceq T^\infty[\mathrm{SA}[b+j_2]..\mathrm{SA}[b+j_2]+|X|+7\tau)$.

We have shown that the sequences $\mathrm{SA}[b+1],\ldots,\mathrm{SA}[b+d]$ and $b_{m+1}-|X|,\ldots,b_{m+d}-|X|$ both contain the same set of positions $\mathrm{Occ}_\ell(\mathrm{SA}[i])$ ordered according to the length-$(|X|+7\tau)$ right context in T^∞. Therefore, regardless of how ties are resolved in each sequence, for any $\delta\in[1..d]$, we have

$$T^\infty[\mathrm{SA}[b+\delta]..\mathrm{SA}[b+\delta]+|X|+7\tau) = T^\infty[b_{m+\delta}-|X|..b_{m+\delta}+7\tau).$$

To finalize the proof of the claim, take any $p\in\text{r-select}_{\mathcal{P}}(X,X',m+\delta)$. By Item (ii), for $j=m+\delta$, we have $T^\infty[p-|X|..p+7\tau) = T^\infty[b_{m+\delta}-|X|..b_{m+\delta}+7\tau) = T^\infty[\mathrm{SA}[b+\delta]..\mathrm{SA}[b+\delta]+|X|+7\tau)$. In particular, by $2\ell\le 7\tau\le 7\tau+|X|$, we obtain $T^\infty[p-|X|..p-|X|+2\ell) = T^\infty[\mathrm{SA}[b+\delta]..\mathrm{SA}[b+\delta]+2\ell)$, i.e., the claim.

2. Applying Item 1 for $\delta=i-b$, we conclude that any position $p\in\text{r-select}_{\mathcal{P}}(X,X',m+\delta)$, satisfies $T^\infty[p-|X|..p-|X|+2\ell) = T^\infty[\mathrm{SA}[b+\delta]..\mathrm{SA}[b+\delta]+2\ell) = T^\infty[\mathrm{SA}[i]..\mathrm{SA}[i]+2\ell)$, i.e., $p-|X|\in\mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$. □

Proposition 5.10. *Let $i\in[1..n]$ be such that $\mathrm{SA}[i]\in[1..n]\setminus\mathrm{R}(\tau,T)$. Under Assumption 5.3, given the values i, $\mathrm{RBeg}_\ell(\mathrm{SA}[i])$, $\mathrm{REnd}_\ell(\mathrm{SA}[i])$, and some $j\in\mathrm{Occ}_\ell(\mathrm{SA}[i])$ as input, we can compute some $j'\in\mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$ in $O(t)$ time.*

Proof. We start by calculating $|\mathrm{Occ}_\ell(\mathrm{SA}[i])| = \mathrm{REnd}_\ell(\mathrm{SA}[i])-\mathrm{RBeg}_\ell(\mathrm{SA}[i])$ using the input arguments. If $|\mathrm{Occ}_\ell(\mathrm{SA}[i])|=1$, then by $\mathrm{Occ}_{2\ell}(\mathrm{SA}[i])\neq\emptyset$ and $\mathrm{Occ}_{2\ell}(\mathrm{SA}[i])\subseteq\mathrm{Occ}_\ell(\mathrm{SA}[i])$ we have $\mathrm{RBeg}_{2\ell}(\mathrm{SA}[i]) = \mathrm{RBeg}_\ell(\mathrm{SA}[i])$, $\mathrm{REnd}_{2\ell}(\mathrm{SA}[i]) = \mathrm{REnd}_\ell(\mathrm{SA}[i])$ and $\mathrm{Occ}_{2\ell}(\mathrm{SA}[i]) = \mathrm{Occ}_\ell(\mathrm{SA}[i])$. Thus, we return $j' := j$. Let us thus assume $|\mathrm{Occ}_\ell(\mathrm{SA}[i])|>1$, and observe that by the uniqueness of $T[n]=\$$, $3\tau-1\le\ell$, and $T^\infty[\mathrm{SA}[i]..\mathrm{SA}[i]+\ell) = T^\infty[j..j+\ell)$, this implies $\mathrm{SA}[i], j\in[1..n-3\tau+1]$. Moreover, by $\mathrm{SA}[i]\notin\mathrm{R}(\tau,T)$, we also have $j\notin\mathrm{R}(\tau,T)$. Therefore, as noted in the proof of Lemma 5.5, $s=\mathrm{succ}_{\mathrm{S}}(j)$ satisfies $s\in\mathrm{S}$ and using Assumption 5.3 we can compute s in $O(t)$ time. Let $X\in\Sigma^*$ and $X',Y\in\Sigma^+$ be such that $\overline{X}=T[j..s)$, $X'=Xc^\infty$, and $T^\infty[j..j+\ell) = \overline{X}Y$ (where $c=\max\Sigma$). By $T^\infty[j..j+\ell) = T^\infty[\mathrm{SA}[i]..\mathrm{SA}[i]+\ell)$ and the consistency condition of S (Definition 2.1), for any $t\in[0..\tau)$, $\mathrm{SA}[i]+t\in\mathrm{S}$ holds if and only if $j+t\in\mathrm{S}$. Thus, by $\mathrm{S}\cap[\mathrm{SA}[i]..\mathrm{SA}[i]+\tau)\neq\emptyset$, denoting $s'=\mathrm{succ}_{\mathrm{S}}(\mathrm{SA}[i])$, it holds $s'-\mathrm{SA}[i] = s-j$. Consequently, $T^\infty[\mathrm{SA}[i]..s') = T^\infty[j..s) = \overline{X}$ and $T^\infty[s'..\mathrm{SA}[i]+\ell) = T^\infty[s..j+\ell) = Y$. We can thus apply Lemma 5.9 without knowing the values of $\mathrm{SA}[i]$ or s' (we only need to know $|X|$ and the starting position of some occurrence of $\overline{X}Y$ in T). First, using Item 1 of Problem 4.3 with the query arguments $(i,q_l,q_r) = (s,s-j,\ell-(s-j))$ we compute in $O(t)$ time (which is possible under Assumption 5.3) the value $m := \text{r-count}_{\mathcal{P}}(X,X',Y)$ (the arguments satisfy the requirements of Problem 4.3 since $q_l = s-j<\tau\le\ell<n$ and $q_r=\ell-(s-j)\le\ell<n$). We then calculate $\delta = i-\mathrm{RBeg}_\ell(\mathrm{SA}[i])$ and using Item 2 of Problem 4.3 with the query arguments $(i,q_l,r) = (s,s-j,m+\delta)$ we compute in $O(t)$ time some position $p\in\text{r-select}_{\mathcal{P}}(X,X',m+\delta)$ (q_l satisfies the requirements of Problem 4.3 by the argument as above). By Lemma 5.9, for $j' := p-q_l$ we then have $j'\in\mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$. In total, we spend $O(t)$ time. □

5.2.3 The Data Structure. By combining the above results, we obtain that under Assumption 5.3, given an index $i\in[1..n]$ satisfying $\mathrm{SA}[i]\in[1..n]\setminus\mathrm{R}(\tau,T)$, along with $\mathrm{RBeg}_\ell(\mathrm{SA}[i])$, $\mathrm{REnd}_\ell(\mathrm{SA}[i])$ and some $j\in\mathrm{Occ}_\ell(\mathrm{SA}[i])$ as input, we can efficiently compute $(\mathrm{RBeg}_{2\ell}(\mathrm{SA}[i]),\mathrm{REnd}_{2\ell}(\mathrm{SA}[i]))$ and some $j'\in\mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$.

Proposition 5.11. *Let $i\in[1..n]$ be such that $\mathrm{SA}[i]\in[1..n]\setminus\mathrm{R}(\tau,T)$. Under Assumption 5.3, given the index i along with values $\mathrm{RBeg}_\ell(\mathrm{SA}[i])$, $\mathrm{REnd}_\ell(\mathrm{SA}[i])$, and some position $j\in\mathrm{Occ}_\ell(\mathrm{SA}[i])$, we can compute $(\mathrm{RBeg}_{2\ell}(\mathrm{SA}[i]),\mathrm{REnd}_{2\ell}(\mathrm{SA}[i]))$ and some position $j'\in\mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$ in $O(t)$ time.*

Proof. First, using Proposition 5.10, we compute some $j'\in\mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$. This takes $O(t)$ time and all the required values (i, $\mathrm{RBeg}_\ell(\mathrm{SA}[i])$, $\mathrm{REnd}_\ell(\mathrm{SA}[i])$, and some $j\in\mathrm{Occ}_\ell(\mathrm{SA}[i])$) are given as input. We now observe that since for j' we have $T^\infty[j'..j'+2\ell) = T^\infty[\mathrm{SA}[i]..\mathrm{SA}[i]+2\ell)$, we have $j''\in\mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$ if and only if $j''\in\mathrm{Occ}_{2\ell}(j')$. Thus, $\mathrm{Occ}_{2\ell}(\mathrm{SA}[i]) = \mathrm{Occ}_{2\ell}(j')$. On the other hand, by Lemma 5.2(1), we have $j''\in\mathrm{Pos}_\ell(\mathrm{SA}[i])$ if and only if $T^\infty[\mathrm{SA}[i]..\mathrm{SA}[i]+\ell)\preceq T^\infty[j''..j''+2\ell)\prec T^\infty[\mathrm{SA}[i]..\mathrm{SA}[i]+2\ell)$, i.e., whether $j''\in\mathrm{Pos}_\ell(\mathrm{SA}[i])$ depends only on $T^\infty[\mathrm{SA}[i]..\mathrm{SA}[i]+2\ell)$. Therefore, $j'\in\mathrm{Occ}_{2\ell}(\mathrm{SA}[i])$ implies $\mathrm{Pos}_\ell(\mathrm{SA}[i]) = \mathrm{Pos}_\ell(j')$. Thus, in the second step of the query, using Proposition 5.8 we compute in $O(t)$ time the values

$$\delta := |\mathrm{Pos}_\ell(j')| = |\mathrm{Pos}_\ell(\mathrm{SA}[i])| \text{ and}$$
$$m := |\mathrm{Occ}_{2\ell}(j')| = |\mathrm{Occ}_{2\ell}(\mathrm{SA}[i])|.$$

Letting $b=\mathrm{RBeg}_\ell(\mathrm{SA}[i])$, then $(\mathrm{RBeg}_{2\ell}(\mathrm{SA}[i]),\mathrm{REnd}_{2\ell}(\mathrm{SA}[i])) = (b+\delta,b+\delta+m)$. □

5.3 The Final Data Structure

Proposition 5.12. *Under Assumption 5.1 and Assumption 5.3 for $\ell = 2^q$, where $q \in [4..\lceil\log n\rceil)$, given any index $i \in [1..n] \setminus \bigcup_{q=4}^{\lceil\log n\rceil-1} \mathsf{R}(\lfloor \frac{2^q}{3} \rfloor, T)$, we can compute $\mathrm{SA}[i]$ in $O(t \log n)$ time.*

Proof. First, using Assumption 5.1, compute $(\mathrm{RBeg}_{16}(\mathrm{SA}[i]), \mathrm{REnd}_{16}(\mathrm{SA}[i]))$ and some $j \in \mathrm{Occ}_{16}(\mathrm{SA}[i])$ in $O(t)$ time. Then, for $q = 4, \ldots, \lceil\log n\rceil - 1$, we use Proposition 5.11 with $\ell = 2^q$ to compute in $O(t)$ time the pair $(\mathrm{RBeg}_{2^{q+1}}(\mathrm{SA}[i]), \mathrm{REnd}_{2^{q+1}}(\mathrm{SA}[i]))$ and some $j' \in \mathrm{Occ}_{2^{q+1}}(\mathrm{SA}[i])$, given index i along with the pair $(\mathrm{RBeg}_{2^q}(\mathrm{SA}[i]), \mathrm{REnd}_{2^q}(\mathrm{SA}[i]))$ and some $j \in \mathrm{Occ}_{2^q}(\mathrm{SA}[i])$ as input. After executing all steps, we get $(\mathrm{RBeg}_\ell(\mathrm{SA}[i]), \mathrm{REnd}_\ell(\mathrm{SA}[i]))$ and some $j' \in \mathrm{Occ}_\ell(\mathrm{SA}[i])$, where $\ell = 2^{\lceil\log n\rceil} \geq n$. Since for any $k \geq n$, we have $\mathrm{Occ}_k(\mathrm{SA}[i]) = \{\mathrm{SA}[i]\}$, we finally return $\mathrm{SA}[i] = j'$. In total, the query takes $O(t \log n)$ time. □

Remark 5.13. In the full version [43], we show how to generalize Proposition 5.11 to also handle the case $\mathrm{SA}[i] \in \mathsf{R}(\tau, T)$, leading to a version of Proposition 5.12 holding for all $i \in [1..n]$. For this, we develop a component (the index "core") that, given any $i \in [1..n]$, lets us efficiently check if $\mathrm{SA}[i] \in \mathsf{R}(\tau, T)$. Positions satisfying $\mathrm{SA}[i] \notin \mathsf{R}(\tau, T)$ are handled exactly as in Section 5.2. The case $\mathrm{SA}[i] \in \mathsf{R}(\tau, T)$ is processed using a separate component.

6 DYNAMIC SUFFIX ARRAY

In this section, we obtain our main result, a dynamic suffix array, using the abstract query algorithm of Section 5 on top of a data structure that maintains a dynamic text $T \in \Sigma^+$ (see Definition 3.3). Due to space constraints, our discussion of the dynamic text data structure is limited to a specification of the supported operations – a complete implementation is provided in the full version [43].

We define a *labelled string* over an alphabet Σ to be a string over $\Sigma \times \mathbb{Z}_{\geq 0}$. For $c := (a, \ell) \in \Sigma \times \mathbb{Z}_{\geq 0}$, we say that $\mathrm{val}(c) := a$ is the *value* of c and $\mathrm{label}(c) := \ell$ is the *label* of c. For a labelled string $S \in (\Sigma \times \mathbb{Z}_{\geq 0})^*$, we define the set of labels $L(S) = \{\mathrm{label}(S[i]) : i \in [1..|S|]\}$ and the string of values $\mathrm{val}(S) = \mathrm{val}(S[1]) \cdots \mathrm{val}(S[|S|])$.

Instead of maintaining a single labelled string representing T, our data structure internally allows maintaining multiple labelled strings (with character labels unique across the entire collection). This lets us decompose each update into smaller building blocks; for example, a swap (cut-paste) operation can be implemented using three splits followed by three concatenations. This internal interface matches the setting often considered in the literature [3, 30, 51].

For a finite family $\mathcal{L} \subseteq (\Sigma \times \mathbb{Z}_{\geq 0})^*$, we set $L(\mathcal{L}) = \bigcup_{S \in \mathcal{L}} L(S)$ and $\|\mathcal{L}\| = \sum_{S \in \mathcal{L}} |S|$. We say $\mathcal{L}$ is *uniquely labelled* if $|L(\mathcal{L})| = \|\mathcal{L}\|$; equivalently, for each label $\ell \in L(\mathcal{L})$ there exist unique $S \in \mathcal{L}$ and $i \in [1..|S|]$ such that $\ell = \mathrm{label}(S[i])$.

Lemma 6.1 (♠). *There is a data structure maintaining a uniquely labelled family $\mathcal{L} \subseteq (\Sigma \times \mathbb{Z}_{\geq 0})^+$ using the following interface, where $\mathrm{label}(S[1])$ is used as a reference to any string $S \in \mathcal{L}$:*

$\mathrm{concat}(R, S)$: *Given distinct $R, S \in \mathcal{L}$, set $\mathcal{L} := \mathcal{L} \setminus (\{R, S\}) \cup \{R \cdot S\}$.*
$\mathrm{split}(S, i)$: *Given $S \in \mathcal{L}$ and $i \in [1..|S|)$, set $\mathcal{L} := \mathcal{L} \setminus (\{S\}) \cup \{S[1..i], S(i..|S|]\}$.*
$\mathrm{insert}(a, \ell)$: *Given $a \in \Sigma$ and $\ell \in \mathbb{Z}_{\geq 0} \setminus L(\mathcal{L})$, set $\mathcal{L} := \mathcal{L} \cup \{(a, \ell)\}$.*
$\mathrm{delete}(S)$: *Given $S \in \mathcal{L}$ with $|S| = 1$, set $\mathcal{L} := \mathcal{L} \setminus \{S\}$.*
$\mathrm{label}(S, i)$: *Given $S \in \mathcal{L}$ and $i \in [1..|S|]$, return $\mathrm{label}(S[i])$.*
$\mathrm{val}(S, i)$: *Given $S \in \mathcal{L}$ and $i \in [1..|S|]$, return $\mathrm{val}(S[i])$.*
$\mathrm{unlabel}(\ell)$: *Given $\ell \in L(\mathcal{L})$, return (S, i), where $S \in \mathcal{L}$ and $i \in [1..|S|]$ are such that $\ell = \mathrm{label}(S[i])$.*

In the word RAM model with word size w satisfying $L(\mathcal{L}) \subseteq [0..2^w)$, each of these operations can be implemented in $O(\log \|\mathcal{L}\|)$ time.

A simple extension of Lemma 6.1 allows for efficient random access to a dynamic text; it is also rather easy to implement the queries of Assumption 5.1 without any overhead in the update time.

Corollary 6.2 (♠). *A dynamic text $T \in \Sigma^n$ can be implemented so that initialization takes $O(1)$ time, updates take $O(\log n)$ time, and the following $\mathrm{access}(i)$ queries take $O(\log n)$ time:*

$\mathrm{access}(i)$: *given $i \in [1..n]$, return $T[i]$.*

Proposition 6.3 (♠). *A dynamic text $T \in \Sigma^n$ can be implemented so that initialization takes $O(1)$ time, updates take $O(\log n)$ time, and the queries of Assumption 5.1 take $O(\log n)$ time.*

The remaining operations use balanced signature parsing on top of the basic data structure of Lemma 6.1 (see Section 3 for an overview of this technique). We aim for a deterministic implementation of balanced signature parsing, which requires using deterministic dynamic dictionaries and, with current state of the art [25], incurs a multiplicative overhead of $O(\frac{\log^2 \log m}{\log \log \log m})$, where m is the dictionary size. Henceforth, we denote $d(x) := \frac{\log^2 \log x}{\log \log \log x}$.

Lemma 6.4 (♠). *A dynamic text $T \in \Sigma^n$ can be implemented so that initialization takes $O(d(m) \cdot \log^* m)$ time, updates take $O(\log n \cdot d(m) \cdot \log^* m)$ time, and any two fragments of T can be compared lexicographically in $O(\log n \cdot d(m) \cdot \log^* m)$ time, where $m = \sigma + t$ and t is the total number of instruction that the data structure has performed so far.*

Proposition 6.5 (♠). *For any fixed $\ell \in \mathbb{Z}_+$, a dynamic text $T \in \Sigma^n$ can be implemented so that initialization takes $O(d(m) \cdot \log^* m)$ time, updates take $O(\log^2 n \cdot d(m) \cdot (\log^* m)^2)$ time, and the queries of Assumption 5.3 take $O(\log^3 n + \log^2 n \cdot d(m) \cdot \log^* m)$ time, where $m = |\Sigma| + t$ and t is the total number of instructions that the data structure has performed so far.*

We are now ready to describe the main result of our work: a dynamic text implementation that can answer suffix array queries. We start with a version with a bounded lifespan: it takes an additional parameter N at initialization time, and it is only able to handle N operations. Then, we use this solution as a black box to develop an 'everlasting' dynamic suffix array.

Proposition 6.6 (♠). *For any given integer $N \geq \sigma$, a dynamic text $T \in [0..\sigma)^+$ can be implemented so that initialization takes $O(\log N \cdot d(N) \cdot \log^* N)$ time, updates take $O(\log^3 N \cdot d(N) \cdot (\log^* N)^2)$ time, the suffix array queries take $O(\log^4 N)$ time, and the inverse suffix array queries take $O(\log^5 N)$ time, provided that the total number of updates and queries does not exceed N.*

Proof sketch. We maintain T using data structures of Proposition 6.3 and Lemma 6.4, as well as several instances of the data structures of Proposition 6.5 for $\ell = 2^q$, where $q \in [4..\lceil\log N\rceil)$. The initialization and each update operation needs to be replicated in all these components.

The suffix queries are implemented using a procedure developed in Section 5. Proposition 5.12 provides such a procedure under some restrictions on the allowed queries and, due to the fact that $|T| \leq N$, the components maintained are sufficient to satisfy the assumption required in Proposition 5.12. In the complete proof (in the full version [43]), we use a variant of Proposition 5.12 supporting arbitrary queries, and this requires additional components.

As for the inverse suffix array queries, we perform binary search. In each of the $O(\log |T|) = O(\log N)$ steps, we compare the specified suffix $T[j..|T|]$ with the suffix $T[\mathrm{SA}[i]..|T|]$; here, we use a suffix array query of to determine $\mathrm{SA}[i]$ and the lexicographic comparison (of Lemma 6.4) to compare the two suffixes.

Recall that the running times of all the components are expressed in terms of parameters $n = |T|$ (which does not exceed N) and $m = \sigma + t$, where t is the total number of instructions performed so far by the respective component. This value may differ across components, but we bound it from above by the total number of instructions performed so far by all the components; let us call this value M. Note that each update and query costs $O(\log^{O(1)}(N+M))$ time, which means that $M = O(\sigma + N \log^{O(1)} N) = O(N \log^{O(1)} N)$, where the last step follows from the assumption $N \geq \sigma$.

Consequently, the initialization takes $O(\log N \cdot d(M) \cdot \log^* M) = O(\log N \cdot d(N) \cdot \log^* N)$ time and the updates take $O(\log N \cdot \log^2 n \cdot d(M) \cdot (\log^* M)^2) = O(\log^3 N \cdot d(N) \cdot (\log^* N)^2)$ time. By Proposition 5.12, suffix queries take $O(\log n \cdot (\log^3 n + \log^2 n \cdot d(M) \cdot \log^* M)) = O(\log^4 N)$ time. The inverse suffix array queries cost $O(\log N \cdot (\log^4 N + \log n \cdot d(M) \cdot \log^* M)) = O(\log^5 N)$ time. □

Theorem 6.7. *A dynamic text $T \in [0..\sigma)^n$ can be implemented so that initialization takes $O(\log \sigma \cdot d(\sigma) \cdot \log^* \sigma)$ time, updates take $O(\log^3(n\sigma) \cdot d(n\sigma) \cdot (\log^*(n\sigma))^2)$ time, the suffix array queries take $O(\log^4(n\sigma))$ time, and the inverse suffix array queries take $O(\log^5(n\sigma))$ time, where $d(x) = \frac{\log^2 \log x}{\log \log \log x}$.*

Proof. We first describe an amortized-time solution which performs a *reorganization* every $\Omega(n)$ operations. This reorganization takes $O(n \cdot \log^3(n\sigma) \cdot d(n\sigma) \cdot (\log^*(n\sigma))^2)$ time.

The text T is stored using the data structures of Corollary 6.2 and Proposition 6.6. Moreover, we maintain a counter t representing the number of operations that can be performed before reorganization. At initialization time, we set $t = 1$ and initialize both components, setting $N = \sigma$ for Proposition 6.6. The updates and queries are forwarded to the component of Proposition 6.6, but we first perform reorganization (if $t = 0$) and decrement t (unconditionally). As for the reorganization, we set $t = \lceil \frac{1}{2}|T| \rceil$, discard the component of Proposition 6.6, and initialize a fresh copy using $N = \max(\sigma, \lceil \frac{3}{2}|T| \rceil - 1)$; we then insert characters of T one by one using access of Corollary 6.2 and insert of Proposition 6.6.

To prove that this implementation is correct, we must argue that each instance of Proposition 6.6 performs no more than N operations. The instance created at initialization time is limited to a single operation, which is no more than the allowance of $N = \sigma$ operations. On the other hand, an instance created during a reorganization performs $|T|-1$ insertions during the reorganization, and is then limited to $\lceil \frac{1}{2}|T| \rceil$ operations. In total, this does not exceed the allowance of $N = \max(\sigma, \lceil \frac{3}{2}|T| \rceil - 1)$ operations.

It remains to analyze the time complexity. For this, we observe that, if $N > \sigma$, then $|T| \geq |T| - t \geq \lfloor \frac{1}{3}N \rfloor$ is preserved as an invariant. This means that $N = O(\max(\sigma, |T|)) = O(\sigma|T|)$, and thus the operation times of Proposition 6.6 can be expressed using $n\sigma$ instead of N. This also applies to the cost of reorganization, which uses initialization and $n-1$ updates.

As for the deamortization, we use the standard technique of maintaining two instances of the above data structure. At any time, one them is active (handles updates and queries), whereas the other undergoes reorganization. The lifetime of the entire solution is organized into *epochs*. At the beginning of each epoch, the active instance is ready to handle $t \geq \frac{1}{2}n$ forthcoming operations, whereas the other instance needs to be reorganized. The epoch lasts for t operations. During the first half of the epoch, the reorganization is performed in the background and the updates are buffered in a queue. For each operation in the second half of the epoch, at most one update in buffered (none if the operation is a query) and two buffered updates are executed (unless there are already fewer updates in the buffer). Since the reorganization cost is bounded by $O(t \cdot \log^3 t \cdot d(t) \cdot (\log^* t)^2)$ and since the query cost is larger than the update cost, the deamortized solution has the same asymptotic time complexity as the amortized one. □

REFERENCES

[1] Donald Adjeroh, Tim Bell, and Amar Mukherjee. 2008. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching.* Springer, Boston, MA, USA. https://doi.org/10.1007/978-0-387-78909-5

[2] Shyan Akmal and Ce Jin. 2022. Near-Optimal Quantum Algorithms for String Problems. In *Proc. SODA*. 2791–2832. https://doi.org/10.1137/1.9781611977073.109

[3] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. 2000. Pattern matching in dynamic texts. In *Proc. SODA*. 819–828. http://dl.acm.org/citation.cfm?id=338219.338645

[4] Mai Alzamel, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Waleń, and Wiktor Zuba. 2019. Quasi-Linear-Time Algorithm for Longest Common Circular Factor. In *Proc. CPM*. 25:1–25:14. https://doi.org/10.4230/LIPIcs.CPM.2019.25

[5] Amihood Amir and Itai Boneh. 2020. Update Query Time Trade-Off for Dynamic Suffix Arrays. In *Proc. ISAAC*. 63:1–63:16. https://doi.org/10.4230/LIPIcs.ISAAC.2020.63

[6] Amihood Amir and Itai Boneh. 2021. Dynamic Suffix Array with Sub-linear update time and Poly-logarithmic Lookup Time. arXiv:2112.12678

[7] Diego Arroyuelo, Gonzalo Navarro, and Kunihiko Sadakane. 2012. Stronger Lempel-Ziv Based Compressed Text Indexing. *Algorithmica* 62, 1-2 (2012), 54–101. https://doi.org/10.1007/s00453-010-9443-8

[8] Djamal Belazzougui, Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Alberto Ordóñez Pereira, Simon J. Puglisi, and Yasuo Tabei. 2015. Queries on LZ-bounded encodings. In *Proc. DCC*. 83–92. https://doi.org/10.1109/DCC.2015.69

[9] Philip Bille, Mikko Berggren Ettienne, Inge Li Gørtz, and Hjalte Wedel Vildhøj. 2018. Time-space trade-offs for Lempel-Ziv compressed indexing. *Theor. Comput. Sci.* 713 (2018), 66–77. https://doi.org/10.1016/j.tcs.2017.12.021

[10] Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. 2015. Random access to grammar-compressed strings and trees. *SIAM J. Comput.* 44, 3 (2015), 513–539. https://doi.org/10.1137/130936889

[11] Or Birenzwige, Shay Golan, and Ely Porat. 2020. Locally Consistent Parsing for Text Indexing in Small Space. In *Proc. SODA*. 607–626. https://doi.org/10.1137/1.9781611975994.37

[12] Michael Burrows and David J. Wheeler. 1994. *A block-sorting lossless data compression algorithm.* Technical Report 124. Digital Equipment Corporation, Palo Alto, California. http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf

[13] Ho-Leung Chan, Wing-Kai Hon, Tak Wah Lam, and Kunihiko Sadakane. 2007. Compressed indexes for dynamic text collections. *ACM Trans. Algorithms* 3, 2 (2007), 21. https://doi.org/10.1145/1240233.1240244

[14] Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. 2021. Faster Algorithms for Longest Common Substring. In *Proc. ESA*. 30:1–30:17. https://doi.org/10.4230/LIPIcs.ESA.2021.30

[15] Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. 2020. Faster Approximate Pattern Matching: A Unified Approach. In *Proc. FOCS*. 978–989. https://doi.org/10.1109/FOCS46700.2020.00095

[16] Bernard Chazelle. 1988. A Functional Approach to Data Structures and Its Use in Multidimensional Searching. *SIAM J. Comput.* 17, 3 (1988), 427–462. https://doi.org/10.1137/0217026

[17] Yu-Feng Chien, Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. 2015. Geometric BWT: Compressed Text Indexing via Sparse Suffixes and Range Searching. *Algorithmica* 71, 2 (2015), 258–278. https://doi.org/10.1007/s00453-013-9792-1

[18] Anders Roy Christiansen, Mikko Berggren Ettienne, Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. 2021. Optimal-Time Dictionary-Compressed Indexes. *ACM Trans. Algorithms* 17, 1 (2021), 8:1–8:39. https://doi.org/10.1145/3426473

[19] Francisco Claude and Gonzalo Navarro. 2011. Self-Indexed Grammar-Based Compression. *Fundam. Informaticae* 111, 3 (2011), 313–337. https://doi.org/10.3233/FI-2011-565

[20] Francisco Claude, Gonzalo Navarro, and Alejandro Pacheco. 2021. Grammar-compressed indexes with logarithmic search time. *J. Comput. Syst. Sci.* 118 (2021), 53–74. https://doi.org/10.1016/j.jcss.2020.12.001

[21] Richard Cole and Uzi Vishkin. 1986. Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking. *Inf. Control.* 70, 1 (1986), 32–53. https://doi.org/10.1016/S0019-9958(86)80023-7

[22] Paul F. Dietz and Daniel Dominic Sleator. 1987. Two Algorithms for Maintaining Order in a List. In *Proc. STOC*. 365–372. https://doi.org/10.1145/28395.28434

[23] Andrzej Ehrenfeucht, Ross M. McConnell, Nissa Osheim, and Sung-Whan Woo. 2011. Position heaps: A simple and dynamic text indexing data structure. *J. Discrete Algorithms* 9, 1 (2011), 100–121. https://doi.org/10.1016/j.jda.2010.12.001

[24] Paolo Ferragina and Giovanni Manzini. 2005. Indexing compressed text. *J. ACM* 52, 4 (2005), 552–581. https://doi.org/10.1145/1082036.1082039

[25] Johannes Fischer and Paweł Gawrychowski. 2015. Alphabet-Dependent String Searching with Wexponential Search Trees. In *Proc. CPM*. 160–171. https://doi.org/10.1007/978-3-319-19929-0_14

[26] Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. 2012. A Faster Grammar-Based Self-index. In *Proc. LATA*. 240–251. https://doi.org/10.1007/978-3-642-28332-1_21

[27] Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. 2014. LZ77-Based Self-indexing with Faster Pattern Matching. In *Proc. LATIN*. 731–742. https://doi.org/10.1007/978-3-642-54423-1_63

[28] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. 2020. Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space. *J. ACM* 67, 1 (apr 2020), 1–54. https://doi.org/10.1145/3375890

[29] Paweł Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Łącki, and Piotr Sankowski. 2015. Optimal Dynamic Strings. arXiv:1511.02612

[30] Paweł Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Łącki, and Piotr Sankowski. 2018. Optimal Dynamic Strings. In *Proc. SODA*. 1509–1528. https://doi.org/10.1137/1.9781611975031.99

[31] Rodrigo González and Gonzalo Navarro. 2009. Rank/select on dynamic compressed sequences and applications. *Theor. Comput. Sci.* 410, 43 (2009), 4414–4422. https://doi.org/10.1016/j.tcs.2009.07.022

[32] Roberto Grossi and Jeffrey Scott Vitter. 2005. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM J. Comput.* 35, 2 (2005), 378–407. https://doi.org/10.1137/S0097539702402354

[33] Ming Gu, Martin Farach, and Richard Beigel. 1994. An Efficient Algorithm for Dynamic Text Indexing. In *Proc. SODA*. 697–704. http://dl.acm.org/citation.cfm?id=314464.314675

[34] Dan Gusfield. 1997. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press. https://doi.org/10.1017/cbo9780511574931

[35] Torben Hagerup. 1998. Sorting and Searching on the Word RAM. In *Proc. STACS*. 366–398. https://doi.org/10.1007/BFb0028575

[36] Meng He and J. Ian Munro. 2010. Succinct Representations of Dynamic Strings. In *Proc. SPIRE*. 334–346. https://doi.org/10.1007/978-3-642-16321-0_35

[37] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. 2015. Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture. In *Proc. STOC*. 21–30. https://doi.org/10.1145/2746539.2746609

[38] Juha Kärkkäinen. 1999. *Repetition-based Text Indexes*. Ph. D. Dissertation. University of Helsinki. https://helda.helsinki.fi/bitstream/handle/10138/21348/repetiti.pdf

[39] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. 2001. Linear-Time Longest-Common-Prefix computation in suffix arrays and its applications. In *Proc. CPM*. 181–192. https://doi.org/10.1007/3-540-48194-X_17

[40] Dominik Kempa and Tomasz Kociumaka. 2019. String synchronizing sets: Sublinear-time BWT construction and optimal LCE data structure. In *Proc. STOC*. 756–767. https://doi.org/10.1145/3313276.3316368

[41] Dominik Kempa and Tomasz Kociumaka. 2020. Resolution of the Burrows-Wheeler Transform Conjecture. In *Proc. FOCS*. 1002–1013. https://doi.org/10.1109/FOCS46700.2020.00097

[42] Dominik Kempa and Tomasz Kociumaka. 2021. Breaking the $O(n)$-Barrier in the Construction of Compressed Suffix Arrays. arXiv:2106.12725

[43] Dominik Kempa and Tomasz Kociumaka. 2022. Dynamic Suffix Array with Polylogarithmic Queries and Updates. arXiv:2201.01285

[44] Tsvi Kopelowitz. 2012. On-Line Indexing for General Alphabets via Predecessor Queries on Subsets of an Ordered List. In *Proc. FOCS*. 283–292. https://doi.org/10.1109/FOCS.2012.79

[45] Sebastian Kreft and Gonzalo Navarro. 2013. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.* 483 (2013), 115–133. https://doi.org/10.1016/j.tcs.2012.02.006

[46] George S. Lueker and Dan E. Willard. 1982. A Data Structure for Dynamic Range Queries. *Inf. Process. Lett.* 15, 5 (1982), 209–213. https://doi.org/10.1016/0020-0190(82)90119-3

[47] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. 2015. *Genome-scale algorithm design: Biological sequence analysis in the era of high-throughput sequencing*. Cambridge University Press, Cambridge, UK. https://doi.org/10.1017/cbo9781139940023

[48] Veli Mäkinen and Gonzalo Navarro. 2008. Dynamic entropy-compressed sequences and full-text indexes. *ACM Trans. Algorithms* 4, 3 (2008), 32:1–32:38. https://doi.org/10.1145/1367064.1367072

[49] Udi Manber and Eugene W. Myers. 1993. Suffix Arrays: A new method for on-line string searches. *SIAM J. Comput.* 22, 5 (1993), 935–948. https://doi.org/10.1137/0222058

[50] Shirou Maruyama, Masaya Nakahara, Naoya Kishiue, and Hiroshi Sakamoto. 2013. ESP-index: A compressed index based on edit-sensitive parsing. *J. Discrete Algorithms* 18 (2013), 100–112. https://doi.org/10.1016/j.jda.2012.07.009

[51] Kurt Mehlhorn, R. Sundar, and Christian Uhrig. 1997. Maintaining Dynamic Sequences under Equality Tests in Polylogarithmic Time. *Algorithmica* 17, 2 (1997), 183–198. https://doi.org/10.1007/BF02522825

[52] J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. 2020. Text Indexing and Searching in Sublinear Time. In *Proc. CPM*. 24:1–24:15. https://doi.org/10.4230/LIPIcs.CPM.2020.24

[53] Gonzalo Navarro. 2016. *Compact data structures: A practical approach*. Cambridge University Press, Cambridge, UK. https://doi.org/10.1017/cbo9781316588284

[54] Gonzalo Navarro and Veli Mäkinen. 2007. Compressed full-text indexes. *ACM Comput. Surv.* 39, 1 (2007), 2. https://doi.org/10.1145/1216370.1216372

[55] Gonzalo Navarro and Yakov Nekrich. 2014. Optimal Dynamic Sequence Representations. *SIAM J. Comput.* 43, 5 (2014), 1781–1806. https://doi.org/10.1137/130908245

[56] Gonzalo Navarro and Kunihiko Sadakane. 2014. Fully Functional Static and Dynamic Succinct Trees. *ACM Trans. Algorithms* 10, 3 (2014), 16:1–16:39. https://doi.org/10.1145/2601073

[57] Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. 2016. Fully Dynamic Data Structure for LCE Queries in Compressed Space. In *Proc. MFCS*. 72:1–72:15. https://doi.org/10.4230/LIPIcs.MFCS.2016.72

[58] Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. 2020. Dynamic index and LZ factorization in compressed space. *Discret. Appl. Math.* 274 (2020), 116–129. https://doi.org/10.1016/j.dam.2019.01.014

[59] Enno Ohlebusch. 2013. *Bioinformatics algorithms: Sequence analysis, genome rearrangements, and phylogenetic reconstruction*. Oldenbusch Verlag, Ulm, Germany.

[60] Luís M. S. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. 2008. Dynamic Fully-Compressed Suffix Trees. In *Proc. CPM*. 191–203. https://doi.org/10.1007/978-3-540-69068-9_19

[61] Süleyman Cenk Sahinalp and Uzi Vishkin. 1994. Symmetry breaking for suffix tree construction. In *Proc. STOC*. 300–309. https://doi.org/10.1145/195058.195164

[62] Süleyman Cenk Sahinalp and Uzi Vishkin. 1996. Efficient Approximate and Dynamic Matching of Patterns Using a Labeling Paradigm (extended abstract). In *Proc. FOCS*. 320–328. https://doi.org/10.1109/SFCS.1996.548491

[63] Mikaël Salson, Thierry Lecroq, Martine Léonard, and Laurent Mouchard. 2010. Dynamic extended suffix arrays. *J. Discrete Algorithms* 8, 2 (2010), 241–257. https://doi.org/10.1016/j.jda.2009.02.007

[64] Yoshimasa Takabatake, Yasuo Tabei, and Hiroshi Sakamoto. 2014. Improved ESP-index: A Practical Self-index for Highly Repetitive Texts. In *Proc. SEA*. 338–350. https://doi.org/10.1007/978-3-319-07959-2_29

[65] Kazuya Tsuruta, Dominik Köppl, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. 2020. Grammar-compressed Self-index with Lyndon Words. arXiv:2004.05309

[66] Dan E. Willard and George S. Lueker. 1985. Adding Range Restriction Capability to Dynamic Data Structures. *J. ACM* 32, 3 (1985), 597–617. https://doi.org/10.1145/3828.3839