Sliding Window String Indexing in Streams

Philip Bille ✉ 🔾
DTU Compute, Technical University of Denmark, Lyngby, Denmark

Johannes Fischer ✉ 🔾
Department of Computer Science, Technische Universität Dortmund, Germany

Inge Li Gørtz ✉ 🔾
DTU Compute, Technical University of Denmark, Lyngby, Denmark

Max Rishøj Pedersen ✉ 🔾
DTU Compute, Technical University of Denmark, Lyngby, Denmark

Tord Joakim Stordalen ✉ 🔾
DTU Compute, Technical University of Denmark, Lyngby, Denmark

── **Abstract** ──────────────

Given a string S over an alphabet Σ, the *string indexing problem* is to preprocess S to subsequently support efficient pattern matching queries, that is, given a pattern string P report all the occurrences of P in S. In this paper we study the *streaming sliding window string indexing problem*. Here the string S arrives as a stream, one character at a time, and the goal is to maintain an index of the last w characters, called the *window*, for a specified parameter w. At any point in time a pattern matching query for a pattern P may arrive, also streamed one character at a time, and all occurrences of P within the current window must be returned. The streaming sliding window string indexing problem naturally captures scenarios where we want to index the most recent data (i.e. the window) of a stream while supporting efficient pattern matching.

Our main result is a simple $O(w)$ space data structure that uses $O(\log w)$ time with high probability to process each character from both the input string S and any pattern string P. Reporting each occurrence of P uses additional constant time per reported occurrence. Compared to previous work in similar scenarios this result is the first to achieve an efficient worst-case time per character from the input stream with high probability. We also consider a delayed variant of the problem, where a query may be answered at any point within the next δ characters that arrive from either stream. We present an $O(w + \delta)$ space data structure for this problem that improves the above time bounds to $O(\log(w/\delta))$. In particular, for a delay of $\delta = \epsilon w$ we obtain an $O(w)$ space data structure with constant time processing per character. The key idea to achieve our result is a novel and simple hierarchical structure of suffix trees of independent interest, inspired by the classic log-structured merge trees.

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).
Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 4; pp. 4:1–4:18

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The *string indexing problem* is to preprocess a string S into a compact data structure that supports efficient subsequent pattern matching queries, that is, given a pattern string P, report all occurrences of P within S. In this paper, we introduce a basic variant of string indexing called the *streaming sliding window string indexing (SSWSI) problem*. Here, the string S arrives as a stream one character at a time, and the goal is to maintain an index of a *window* of the last w characters, for a specified parameter w. At any point in time a pattern matching query for a pattern P may arrive, also streamed one character at a time, and we need to report the occurrences of P within the current window. The goal is to compactly maintain the index while processing the characters arriving in either stream efficiently. We consider two variants of the problem: a *timely* variant where each query must be answered immediately, and a *delayed* variant where it may be answered at any point within the next δ characters arriving from either stream, for a specified parameter δ. See Section 1.1 for precise definitions.

The SSWSI problem naturally captures scenarios where we want to index the most recent data (i.e. the window) of a stream while supporting efficient pattern matching. For instance, monitoring a high-rate data stream system where we cannot feasibly index the entire stream but still want to support efficient queries. Depending on the specific system we may require immediate answers to queries, or we may be able to afford a delay that allows for more efficient queries and updates.

The SSWSI problem has not been explicitly studied before in our precise formulation, but for the timely variant several closely related problem are well-studied. In particular, the *sliding window suffix tree problem* [8, 13, 24, 25, 28] is to maintain the *suffix tree* of the current window (i.e., the compact trie of the suffixes of the window) as each character arrives. With appropriate augmentation the suffix tree can be used to process pattern matching queries efficiently, leading to a solution to the timely SSWSI problem. For constant-sized alphabets, the best of these solutions [8] maintains the sliding window suffix tree in constant *amortized* time per character while supporting efficient pattern matching queries. The worst-case time for updates is $\Omega(w)$. The other solutions achieve similar amortized time bounds. This amortization cannot be avoided since explicitly maintaining the suffix tree after the arrival of a new character may incur $\Omega(w)$ changes.

Another closely related problem is the *online string indexing problem* [3,4,5,7,14,21,22,23]. Here the goal is to process S one character at a time (in either left-to-right or right-to-left order), while incrementally building an index of the string read so far. The best of these solutions update the index in either constant time per character for constant-sized alphabets [23] or $O(\log \log n + \log \log |\Sigma|)$ time for any alphabet where each character fits in a constant number of machine words [21]. These solutions all heavily rely on processing the string in right-to-left order to avoid the inherent linear time suffix tree updates due to appending, as mentioned above. Therefore they cannot be applied in our left-to-right streaming setting. Alternatively, we can instead apply these solutions on the reverse of the string S, but then each pattern must be processed in reverse order, which also cannot be done in our setting. Also, note that these solutions index the entire string read so far. It is not clear if they can be adapted to efficiently index a sliding window.

Another line of work shows how to maintain a fully dynamic suffix array under insertions and deletions [1, 2, 19, 27]. These can also be used to solve SSWSI but are more general and lead to polylogarithmically slower bounds than our results while being more complicated.

Our main result is an efficient and simple solution to the SSWSI problem in both the timely and delayed variant. Let w denote the size of the window. For the timely variant, we present a string index that uses $O(w)$ space and processes a character from the stream S in $O(\log w)$ time. Each pattern matching query P is also supported in $O(\log w)$ time per character with additional $O(\text{occ})$ time incurred after receiving the last character of P, where occ is the number of occurrences of P in the current window. The index is randomized and both time bounds hold with high probability. Compared to previous suffix tree based approaches for indexing a sliding window, we improve the worst-case time bounds per character in the stream from $\Omega(w)$ to $O(\log w)$ with high probability. This is particularly important in the above mentioned applications, such as high-rate data stream systems. Our solution generalizes to the delayed variant of the problem. If we allow a delay of δ before answering each query we achieve $O(w + \delta)$ space while improving the above time bounds to $O(\log(w/\delta))$. In particular, if we allow a delay of $\delta = \epsilon w$ for any constant $\epsilon > 0$, we achieve linear space and optimal constant time (reporting the occurrences still takes $O(\text{occ})$ time, and we do not count the reporting time towards the delay). Note that $\delta \leq w$ is sufficient delay for optimal time bounds and we can assume $O(w + \delta) = O(w)$. The results hold on a word RAM and for any alphabet size, assuming that each character fits into a constant number of machine words.

The key idea to achieve our result is a novel and simple hierarchical structure of suffix trees inspired by log-structured merge trees [26]. Instead of maintaining a single suffix tree on the window we maintain a collection of suffix trees of exponentially increasing sizes that cover the current window. We show how to efficiently maintain the structure as new characters from the stream arrive by incrementally "merging" suffix trees, while supporting efficient pattern matching queries within the window.

Our solution uses randomization to construct suffix trees in linear time with high probability. Plugging in a deterministic construction algorithm such as the one by Ukkonen [30], we obtain a solution using $O(\log w \log |\Sigma|)$ time for both queries and updates. With more recent deterministic suffix tree solutions [6, 10, 14] we can improve this to obtain $O(\log w \log \log n)$ time per character for both queries and updates. Note that the $O(\log \log |\Sigma|)$ in the time bounds of [14] has been replaced by $O(\log \log n)$ here due to an additional sorting step using [17].

1.1 Setup and Results

We formally define the problem as follows. Let S be a stream over any alphabet Σ where each character fits in a constant number of machine words. For given integer parameters $w \geq 1$ and $\delta \geq 0$, the δ-*delayed streaming sliding window string indexing ((w, δ)-SSWSI)* problem is to maintain a data structure that, after receiving the first i characters of S, supports

- Report(P): report all the occurrences of P in $S[i - w + 1, i]$ before an additional δ characters have arrived from either stream.
- Update(): process the next character in the stream S.

In the Report(P) query the pattern string P is also streamed. When P is streamed it interrupts the stream S, arrives one character at a time, and all characters of P arrive before the streaming of S resumes. Furthermore, we do not assume that we know the length of P before the arrival of its last character. Although P is streamed we assume random access to its characters after they arrive, as any pattern that fits in the window is at most w characters long and we can afford to store it. The delay is counted from after the last character of P arrives. Characters from S and from new patterns count towards the delay, while reported occurrences do not (otherwise it would be impossible to answer the query in time if there are more than δ occurrences).

We define the *timely streaming sliding window string indexing (w-SSWSI) problem* to be $(w, 0)$-SSWSI, that is, queries must be answered immediately as the last character of the pattern arrives.

We show the following general main result.

▶ **Theorem 1.** *Let S be a stream and let $w \geq 1$ and $\delta \geq 0$ be integers. We can solve the (w, δ)-SSWSI problem on S with an $O(w + \delta)$ space data structure that supports* Update *and* Report *in $O(\log \frac{w}{\delta+1})$ time per character with high probability. Furthermore,* Report *uses additional worst-case constant time per reported occurrence.*

Here, with high probability means with probability at least $1 - \frac{1}{w^d}$ for any constant d. Theorem 1 provides a trade-off in the delay parameter δ. In particular, plugging in $\delta = 0$ in Theorem 1 we obtain a solution to the timely SSWSI problem that uses $O(w)$ space and $O(\log w)$ time per character for both Update and Report. Compared to the previous work on sliding window stream indexing [8, 13, 18, 24, 25, 28, 29] this improves the worst-case bounds on the Update operation from $\Omega(w)$ to $O(\log w)$ with high probability and also removes the restriction on the alphabet. At the other extreme, plugging in $\delta = \epsilon w$ for constant $\epsilon > 0$ in Theorem 1 we obtain a solution to the delayed SSWSI problem that uses $O(w)$ space and optimal constant time per character with high probability. All our results hold on a word RAM where each machine word has at least $\log w$ bits, and where each character of the alphabet fits into a constant number of machine words.

1.2   Techniques

We obtain our result for the timely variant, but without high probability guarantees, as follows. At all times we maintain at most $\log w$ suffix trees that do not overlap and together cover the window. The trees are organized by the *log-structured merge technique* [26], where the rightmost tree is the smallest and their sizes increase exponentially towards the left. For each new character that arrives we append its suffix tree to the right side of our data structure. Whenever there are two trees of the same size next to each other we "merge" them by constructing a new suffix tree covering them both. Each character from S is involved in at most $\log w$ merges and each merge takes expected linear time, so we spend expected amortized $O(\log w)$ time per character in S. We deamortize the updates by temporarily keeping both trees while merging them in the background. Note that for each adjacent pair of suffix trees we also store a suffix tree approximately covering them both, referred to as *boundary trees* (see details below).

We find the occurrences of a pattern P in the window by querying each of these trees, which takes $O(\log w)$ time per character in P. For adjacent pairs of trees larger than $|P|$ we find the occurrences of P crossing from one into the other using the boundary trees. The remaining trees cover a suffix of the window of length $O(|P|)$, and we grow a suffix tree to answer queries in this suffix *at query time*. Our data structure has some "overhang" on the left side of the window, and we use range maximum queries to report only the occurrences that start inside the window.

This solution is generalized to incorporate a delay of δ as follows. We store the $O(\log(w/\delta))$ largest trees from the timely solution and leave a suffix of size $\Theta(\delta)$ of the window uncovered by suffix trees. We answer queries as follows. If $|P| > \delta/4$ we say that P is *long*, and otherwise it is *short*. For long patterns we do as in the timely case; the suffix tree we grow at query time now must also contain the uncovered suffix, but it still has size $O(|P|)$ since the uncovered part of the window has length $O(\delta) = O(|P|)$. We show how to do this in $O(\log(w/\delta))$ time per character in P. For short patterns we utilize that they are smaller

than the delay to temporarily buffer the queries and later batch process them. We buffer up to $O(\delta \log(w/\delta))$ work and deamortize it over $\Theta(\delta)$ characters, obtaining the same bound as for long patterns. Updates run in the same bound since each character from S is involved in at most $O(\log(w/\delta))$ merges before it leaves the window.

Finally, we improve the time bounds by proving that for any substring S' of our window, we can construct the suffix tree over S' in $O(|S'|)$ time with probability $1 - w^{-d}$ for any constant $d > 1$. We do so by reducing the alphabet $\Sigma' = \{c \in S'\}$ of S' to rank-space $\{1, 2, \ldots, |\Sigma'|\}$ from which the algorithm by Farach-Colton et al. [12] can construct the suffix tree in worst-case linear time. For large strings ($|S'| > w^{1/5}$) we pick a hash function from $\Sigma \to [0, w^c]$ that with high probability is injective on S', and then we use radix sort to reduce to rank-space in linear time. For small strings ($|S'| \le w^{1/5}$) we pick a hash function from $\Sigma \to [0, w/\log w]$ that is injective with (almost) high probability, and use this to manually construct a mapping into rank space in $O(S')$ time. This mapping algorithm uses additional $O(w/\log w)$ space, but we construct at most $O(\log w)$ suffix trees at any time so the total space is linear.

1.3   Outline

In Section 2 we cover the preliminaries, including some useful facts about suffix trees. In Section 3 we give a solution to the timely SSWSI problem that supports each operation in expected logarithmic time per character. In Section 4 we show how to generalize this to incorporate delay, and in Section 5 we show how to get good probability guarantees, proving Theorem 1.

2   Preliminaries

Given a string X of length n over an alphabet Σ, the ith character is denoted $X[i]$ and the substring starting at $X[i]$ and ending at $X[j]$ is denoted $X[i, j]$. The substrings of the form $X[i, n]$ are the *suffixes* of X.

A *segment* of X is an interval $[i, j] = \{i, i+1, \ldots, j\}$ for $1 \le i \le j \le n$. We will sometimes refer to segments as strings, i.e., the segment $[i, j]$ refers to the string $X[i, j]$. The definition differs from "substring" by being specific about position; even if $X[1, 2] = X[3, 4]$ we have $[1, 2] \ne [3, 4]$. A *segmentation* of X is a decomposition of X into disjoint segments that cover it. For instance, $x_1 = [1, i]$ and $x_2 = [i + 1, n]$ is a segmentation of X into two parts. The two segments x_1 and x_2 are *adjacent* since x_2 starts immediately after x_1 ends, and for a pair of adjacent segments we define the *boundary* (x_1, x_2) to be the implicit position between i and $i + 1$.

The *suffix tree* [31] T over X is the compact trie of all suffixes of $X\$$, where $\$ \notin \Sigma$ is lexicographically smaller than any letter in the alphabet. Each leaf corresponds to a suffix of X, and the leaves are ordered from left to right in lexicographically increasing order. The suffix tree uses $O(n)$ space by implicitly representing the string associated with each edge using two indices into X. Farach-Colton et al. [12] show that the optimal construction time for T is $\text{sort}(n, |\Sigma|)$, i.e., the time it takes to sort n elements from the universe Σ. For alphabets of the form $\Sigma = \{0, \ldots, n^c\}$ for constant $c \ge 1$ this implies that T can be built in worst-case $O(n)$ time using radix sort. For larger alphabets we can reduce to the polynomial case in expected linear time using hashing, building T in expected linear time (see Section 5 for details).

The *suffix array* L of X is the array where $L[i]$ is the starting position of the ith lexicographically smallest suffix of X. Note that $L[i]$ corresponds to the ith leaf of T in left-to-right order. Furthermore, let v be an internal node in T and let s_v be the string

spelled out by the root-to-v path. The descendant leaves of v exactly correspond to the suffixes of X that start with s_v, and these leaves correspond to a consecutive range $[\alpha, \beta]_v$ in L.

We augment the suffix tree to support efficient pattern matching queries as follows. First, we use the well-known FKS perfect hashing scheme [15] to store the edges of the suffix tree, so we can for any node determine if there is an outgoing edge matching a character $a \in \Sigma$ in worst-case constant time. Note that this construction takes *expected* linear time. Furthermore, we also build a *range maximum query* data structure over L. This data structure supports range maximum queries, i.e., given a range $[\alpha, \beta]$ return the $j \in [\alpha, \beta]$ maximizing $L[j]$. It also supports range minimum queries, defined analogously. The data structure can be built in linear time and supports queries in constant time [16]. Finally, we preprocess the suffix tree in linear time such that each internal node v stores the range $[\alpha, \beta]_v$ into L corresponding to the occurrences of s_v.
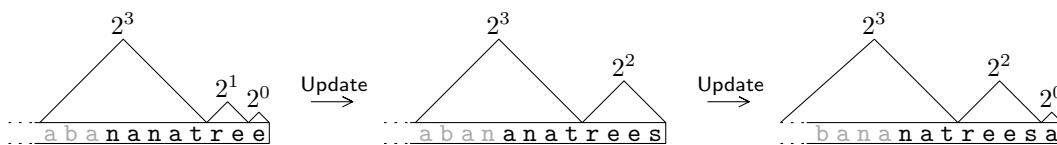
We can use this structure to efficiently find all the occurrences of P in $O(|P| + \text{occ})$ time, where occ is the number of occurrences, or the leftmost and rightmost occurrence of P in $O(|P|)$ time. The *locus* of a string P is the minimum depth node v such that P is a prefix of s_v. First we find the locus by walking downwards in the suffix tree, matching each character in P in worst-case constant time using the dictionary. Once we have found v we can report all the occurrences in $[\alpha, \beta]_v$ in $O(\text{occ})$ time. Alternatively, we can find the rightmost occurrence of P in constant time by doing a range maximum query on the range $[\alpha, \beta]_v$ in L, which returns the $j \in [\alpha, \beta]_v$ maximizing the *string position* $L[j]$. We can also find the leftmost occurrence by doing a range minimum query.

Finally, note that it is possible to deamortize algorithms with *expected* running time using the standard technique of distributing the work evenly. Specifically, if an algorithm runs in expected λn time we can do λ work for $n - 1$ steps; by linearity of expectation only expected λ work remains for the last step.

3   The Timely SSWSI Problem

Here we present a solution for the timely variant that matches the bounds in Theorem 1 in expectation. Section 5 shows how to get the bounds with high probability. Throughout this section we assume without loss of generality that w is a power of two. Section 3.3 briefly mentions how to generalize to arbitrary w.

The main idea is as follows. We maintain a suffix of S of length at least w. This suffix is segmented into at most $\log w$ segments whose sizes are distinct powers of two, in increasing order from right to left. The length of the suffix we store is at most $2^0 + \ldots + 2^{\log w} = 2w - 1$. When a new character arrives, we append a new size-one segment to our data structure and merge equally-sized segments until they all have distinct sizes again. We also discard the largest segment when it no longer intersects the window. For each segment we store a suffix tree, and for every pair of adjacent segments we store a *boundary tree* approximately covering them both (see below). To support queries we query the suffix tree for each individual segment, and also each boundary tree. For the segments larger than the pattern, the boundary trees are sufficient to find the occurrences crossing the respective boundary. The remaining trees cover a suffix of S that is $O(|P|)$ long, and we grow a suffix tree at query time to find the remaining occurrences in this suffix.

Figure 1 Example of updating the data structure with a window size of $w = 8$. Here we illustrate the segments by the suffix trees built over them. Characters outside of the window are gray. As the character `s` arrives we construct a new suffix tree of size one, which is then immediately merged with the existing size-one suffix tree over `e` into a size-two suffix tree over `es`, which is then merged to into the final size-four suffix tree over `rees`. After receiving `a` we again have a size-one suffix tree. Note that after three more updates the suffix tree of size eight will no longer overlap the window and will be discarded.

3.1 Data Structure

At any point, the data structure contains a suffix s of S of length $w \leq |s| \leq 2w - 1$ and a segmentation of s into at most $\log w$ segments. Specifically, if $|s| = 2^{b_1} + \ldots + 2^{b_k}$ for integers $b_1 < \ldots < b_k$ then we have the segmentation $s_1, \ldots, s_k$ where $|s_i| = 2^{b_i}$, and s is the concatenation of the strings $s_k, s_{k-1}, \ldots, s_1$, in that order. The set $\{b_1, \ldots, b_k\}$ is unique and corresponds to the 1-bits in the binary encoding of $|s|$. Three different configurations can be seen in Figure 1.

For each segment s_i we store the suffix tree T_i over s_i, along with a range maximum query data structure over the suffix array of s_i. For each boundary (s_{i+1}, s_i) we store the *boundary tree B_i*, which is the suffix tree over the substring centered at the boundary and extending $|s_i|$ characters in both directions. We augment B_i with an additional data structure that we will use for reporting occurrences across the boundary. Let BL_i be the suffix array corresponding to B_i. We define the *modified suffix array BL'_i* as

$$BL'_i[j] = \begin{cases} BL_i[j] & \text{if } BL_i[j] \text{ corresponds to a suffix starting in } s_{i+1} \\ -\infty & \text{if } BL_i[j] \text{ corresponds to a suffix starting in in } s_i \end{cases}$$

We store a range maximum query data structure over BL'_i. Each of the data structures use $O(s_i)$ space, so the whole data structures uses $O(s) = O(w)$ space.

We note a few properties of the data structure. Let $S[n]$ be the most recent character to arrive and let $W_n = S[n - w + 1, n]$ be the current window. Then W_n is a suffix of s since $|s| \geq w$. The largest, and leftmost, segment s_k always has size $2^{\log w} = w$; it is not larger since $\log w$ bits are sufficient to represent $|s| \leq 2w - 1$, and it is always there since $|s| \geq w$ cannot be represented with $\log w - 1$ bits. For the same reason, s_k always intersects at least partially with W_n, and each of $s_1, \ldots, s_{k-1}$ are fully contained in W_n.

3.2 Queries

The idea is as follows, as exemplified in Figure 2. Any occurrence of a pattern P that is fully contained in a segment is found using the suffix tree over that segment. Similarly, any occurrence that only crosses a single boundary far enough away from the end of the window is found in the respective boundary tree. Note that in the leftmost segment we must be careful to not report any occurrences that start before the left window boundary. The remaining occurrences are not contained in any of the trees in the data structure (either because they cross multiple boundaries or because they cross a single boundary (s_{i+1}, s_i)
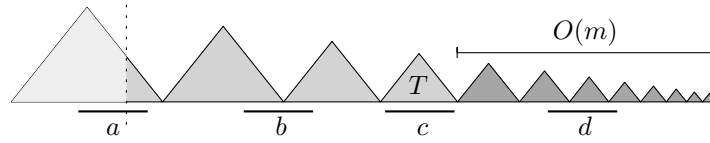
Figure 2 Illustration of how we answer queries for a pattern P of length m. The lines denoted a, b, c, and d indicate occurrences of P. The segmentation is illustrated by the trees over the segments. The leftmost window boundary is marked with a vertical dashed line. Note that the leftmost segment intersects only partially with the window. The tree T marks the smallest segment of size at least m. The segments to the right of T are all smaller than m, so they cover at most $m + m/2 + \ldots + 1 = O(m)$ characters. To answer the query we match P in the tree over each segment and in each boundary tree, and we also build a suffix tree over the segments smaller than m at query time. We find b because the respective boundary tree is sufficiently large. We find c because it is fully contained in a segment. We find d in the suffix tree that we build at query time. Note that a is not contained in the window; we avoid reporting it by recursively using range maximum queries to find the *rightmost* occurrence of P in the leftmost segment.

but start more than $|s_i|$ characters to the left of the boundary). However, these occurrences are all located within a substring of size $O(m)$ ending at position $S[n]$, so we build, at query time, a suffix tree to find these occurrences.

Let P be the length-m pattern being queried, $S[n]$ be the most recent character to arrive, and let W_n, the suffix s, the segmentation $s_1, \ldots, s_k$, and the indices $b_1 < \ldots < b_k$ be defined as above. As mentioned, any occurrence of P in W_n must either be fully contained within one of the segments, or it must cross the boundary between two adjacent segments. We will show how to handle each of these cases separately.

Fully Contained in a Segment

Fix a specific segment s_i. As each character of P arrives we match it in T_i. When the last character arrives we have a (possibly empty) range $[\alpha, \beta]$ into the suffix array of s_i corresponding to the occurrences of P. If s_i is not the leftmost segment then it is fully contained in W_n and we report all the occurrences. Otherwise, $s_i = s_k$ is the leftmost segment, which might overlap only partially with W_n, and it may contain occurrences of P that are not contained in the window. However, note that the intersection between W_n and s_k is a suffix of s_k. Therefore, if an occurrence of P in s_k *starts* inside W_n it also ends inside W_n. We find all such occurrences as follows. Let L_k be the suffix array of s_k. As described in Section 2 we find the index j of the rightmost occurrence of P by doing a range maximum query on the range $[\alpha, \beta]$ in L_k. If $L_k[j]$ is not inside W_n then none of the occurrences are, and we are done. Otherwise we recurse on $[\alpha, j-1]$ and $[\beta, j-1]$. Matching P in the trees of all the segments takes $O(\log w)$ overall time per character of P. Reporting each occurrence takes constant time since range maximum queries run in constant time.

Crossing a Boundary

We now show how to report the occurrences of P that span a boundary. The main idea is as follows, as illustrated in Figure 3. Let s_i be the smallest segment where $|s_i| \geq m$. Consider any boundary (s_{j+1}, s_j) to the left of s_i, i.e., where $j \geq i$. Since both of these segments have size at least $|s_i| \geq m$, the boundary tree B_j extends at least m characters in both directions from the boundary. Therefore, all the occurrences of P crossing the boundary are contained in B_j, and none of them can cross another boundary as well. Now

consider the suffix $\mathcal{R}$ of s containing the $m - 1$ last characters of s_i and extending to the end of s. This substring contains all the other boundary-crossing occurrences. Furthermore, all the occurrences in $\mathcal{R}$ cross at least one boundary since the longest consecutive part of a single segment in $\mathcal{R}$ is the $m - 1$ characters in s_i. Note that the length of $\mathcal{R}$ is at most $m - 1 + |s_{i-1}| + |s_{i-1}|/2 + \ldots + 1 < m - 1 + 2|s_{i-1}| < 3m$ since $|s_{i-1}| < m$. Thus, the number of boundary-crossing occurrences of P equals the number of occurrences in $\mathcal{R}$ plus the number of occurrences crossing the boundaries $(s_k, s_{k-1}), (s_{k-1}, s_{k-2}), \ldots (s_{i+1}, s_i)$.

The algorithm for finding the occurrences in the sufficiently large boundary trees is as follows. Fix a boundary (s_{x+1}, s_x). We match each character of P in B_x as it arrives. When the last character arrives we know if $|s_x| \geq m$, and also the range $[\alpha, \beta]$ corresponding to the occurrences of P in the boundary tree. If $|s_x| \geq m$ (hence $x \geq i$) we report the occurrences as follows. As above we do a range maximum query to find the j maximizing $BL'_x[j]$. If $BL'_x[j] = -\infty$ then all occurrences of P start in s_x, and there are no occurrences crossing the boundary. Otherwise, $BL'_x[j]$ corresponds to the starting position of the rightmost occurrence of P in s_{x+1}. Since all of P has arrived and we now know m, we know that this occurrence crosses the boundary if and only if $BL'_x[j] \geq |s_x| - m + 2$ (recall that B_x extends $|s_x|$ characters in both directions from the boundary). If it does not cross the boundary, then none of the other occurrences do either. Otherwise we report $BL'_x[j]$ and recurse on $[\alpha, j - 1]$ and $[j + 1, \beta]$ to find the remaining occurrences. Matching P in all boundary trees takes $O(\log w)$ overall time per character, and reporting each occurrence with range maximum queries takes constant time.

We now show how to find the occurrences of P in $\mathcal{R}$ with the same bounds. Assume that we know that $2^\ell \leq m < 2^{\ell+1}$ for some integer ℓ. We build the suffix tree over the last $3 \cdot 2^{\ell+1}$ characters of s, deamortized over receiving the first $2^{\ell-1}$ characters of P. Over the next $2^{\ell-1}$ characters we match P in the tree, at a rate of two characters per new character from P. Then, when the 2^ℓth character arrives, we have caught up to the stream P, and we match the remaining $m - 2^\ell$ characters as they arrive. When the last character arrives we have matched P in a tree of size at least $3m$, and we can start reporting occurrences. Note that we are overestimating the size of the tree, and it potentially includes some occurrences of P that are contained in s_i. To avoid reporting these, we also build a range maximum query data structure over the suffix array such that we can use recursive range maximum queries. When deamortized, we construct the tree in expected constant time per character of P. Matching P also takes constant time per character. We know that $m \leq w$, so we run this algorithm simultaneously for each of the $\log w$ different choices for ℓ, using expected $O(\log w)$ time per character in P. Note that the trees use $O(w)$ space in total since the sum of the space is a geometric sum where the largest term is $O(w)$.

3.3   Amortized Updates

We show how to support updates in amortized $O(\log w)$ time. Let $S[n]$ be the last character to arrive and as in the description of the data structure let $b_1 < b_2 < \ldots < b_k$ be the positions of the 1-indices in the binary encoding of $|s|$. When the new character $c = S[n+1]$ arrives, we update s and the segmentation $s_1 \ldots s_k$ to create the new suffix s' with the new segmentation $s'_1, \ldots s'_{k'}$. See Figure 1 for an example.

If $|s| < 2w - 1$ then we set $s' = sc$. The segmentation of s' corresponds to the unique binary encoding of $|s'| = |s| + 1$, so we update the segmentation analogously to a "binary increment". One way to do so is as follows. We create a new segment of size one over c. If there was not already a segment of size one, then we add the new segment and we are done. Otherwise we *merge* (see below) the two size-one segments to create a segment of size
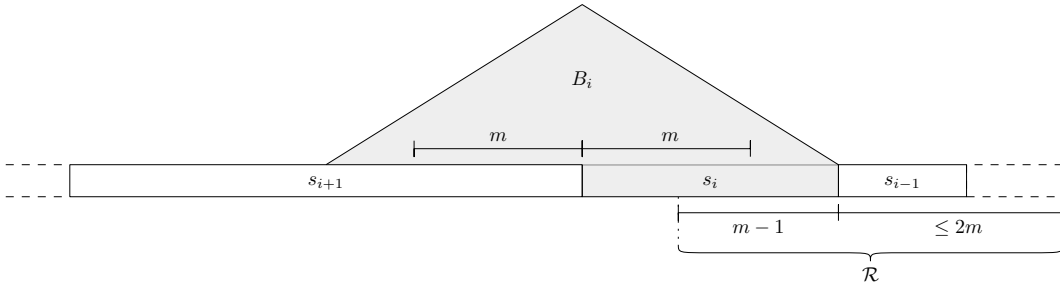
Figure 3 The segment s_i is the smallest segment where $|s_i| \geq m$. For each boundary (s_{j+1}, s_j) where $j \geq i$, the tree B_j is large enough to find all occurrences of P across the boundary. All other occurrences of P that cross a boundary must be in $\mathcal{R}$, the string covering the $m-1$ rightmost characters of s_i and extending to the end of the window. The length of $\mathcal{R}$ is no more than $m-1+|s_{i-1}|+|s_{i-1}|/2+\ldots+1 < 3m$.

two. The process cascades until we reach a size 2^b that does not exist in the segmentation of s (i.e., the smallest index $b \notin \{b_1, \ldots, b_k\}$). At this point we replace all of the segments $s_{b-1}, \ldots, s_1$ with s_1' covering the last 2^b characters of s'. The remaining segments for s' are the same as the segments $s_{b+1}, \ldots, s_k$. If $|s| = 2w - 1$ then there is a segment of each size $2^0, 2^1, \ldots, 2^{\log w}$. Since the segments have decreasing size from left to right, the $\log w - 1$ rightmost segments cover the last $2^0 + \ldots + 2^{\log w - 1} = w - 1$ characters of s. Thus, after c arrives, the leftmost segment of size $2^{\log w} = w$ no longer intersects the window. We remove it by setting $s' = s[w+1, |s|]c$, and update the segmentation as above.

Let s_a, s_b and s_c be three adjacent segments, in that order. To *merge* s_b and s_c we combine them into a new segment s_d that spans them both, construct the suffix tree over s_d, and construct a range maximum query data structure on the suffix array of s_d. Furthermore, since s_a and s_d are now adjacent we also construct the boundary-spanning suffix tree for the boundary (s_a, s_d) that extends $|s_d|$ characters in each direction. The construction of all of these data structures takes expected $O(|s_d|)$ time (see Section 2). Thus, it takes expected constant time per character every time it moves into a new, larger segment. Each character is contained in at most $\log w$ segments before it leaves the window, so the amortized update time is expected $O(\log w)$ per character.

Note that all but the last merge are unnecessary to actually compute s_1'; in the amortized setting we can simply determine where the cascade will end and immediately construct the suffix tree over the corresponding segment. However, the cascading merges will come into play in the deamortized variant.

Also note that if w is not a power of two we can use a similar scheme where we allow either two simultaneous trees of size $2^{\lfloor \log w \rfloor}$, or one tree of size $2^{\lceil \log w \rceil}$. In both cases, there are some straightforward edge cases for when to remove the leftmost segment.

3.4   Deamortized Updates

We now show how to deamortize the updates. Unfortunately the previous construction cannot be directly deamortized since the suffix tree construction algorithm by Farach-Colton et al. [12] requires access to the whole string. Therefore, if a new character c causes a cascade of merges resulting in a new segment of size 2^i we have to build the suffix tree over that segment when c arrives.

Instead, we modify the structure slightly. When two segments of size 2^i become adjacent we temporarily keep both while deamortizing the cost of merging them over the *next* 2^i characters of S, doing expected constant work per character. Note that queries are unaffected,

with one exception for reporting occurrences across the boundaries; there might now be two adjacent segments s_{i+1} and s_i of the same size that are both the smallest segment at least as large as $|P|$. In this case the suffix $\mathcal{R}$ extends only $m-1$ characters into the rightmost segment s_i. The boundary tree for (s_{i+1}, s_i) is large enough to report all occurrence crossing that boundary since both segments have size at least $|P|$. Furthermore, $\mathcal{R}$ potentially becomes twice as long, so we adjust the constants of the trees that we grow at query time.

To bound the time for updates we show that we are constructing at most $\log w$ suffix trees at any point, from which it follows that the update time is expected $O(\log w)$. To do so we show the following lemma.

▶ **Lemma 2.** *When the construction of a segment of size 2^i finishes there is exactly one segment of each size $2^{i-1}, \ldots, 2^0$.*

Proof. The proof is by induction on i. For $i = 1$, when two size-one segments become adjacent we merge them when the next character c from S arrives. This results in a segment of size two, as well as a size-one segment containing c, proving the base case.

Inductively, consider the first time two segments of size 2^i become adjacent. By the induction hypothesis, there is one segment of each size $2^0, 2^1, \ldots, 2^{i-1}$ to the right of these two segments. For another segment of size 2^i to be constructed, we must first receive one more character, which triggers a merge that eventually cascades through all $i-1$ of these segments. For this to happen, $1 + (2^0 + 2^1 + \ldots + 2^{i-1}) = 2^i$ more characters from S must arrive, where the 1 is for the next character to arrive, and 2^j is the amount of characters the jth merge is deamortized over. However, at this point the merge of the two segments of size 2^i is complete, so we constructed two new segments, one of size 2^{i+1} and one of size 2^i. By the induction hypothesis, there is also one segment of each size $2^0, \ldots, 2^{i-1}$, concluding the proof.   ◀

Lemma 2 implies that there are never more than two segments of the same size adjacent to each other, and therefore at most one merging process for each segment size $2^0, 2^1, \ldots, 2^{\log w}$. To see this, consider the first time two segments a and b of size 2^i are adjacent. At this point, there are $2^0 + 2^1 + \ldots + 2^{i-1} = 2^i - 1$ characters to the right of b. When the next segment c of size 2^i arrives there are $2^i - 1$ characters to the right of that, too. But then there are $|c| + 2^i - 1 = 2^i + 2^i - 1$ characters to the right of b. Thus 2^i new characters must have arrived in the meanwhile, and the merging of a and b is done.

We obtain the following theorem.

▶ **Theorem 3.** *Let S be a stream and let $w \geq 1$ be an integer. We can solve the w-SSWSI problem on S with an $O(w)$ space data structure that supports* Update *and* Report *in expected $O(\log w)$ time per character. Furthermore,* Report *uses additional worst-case constant time per reported occurrence.*

4   The Delayed SSWSI Problem

In this section we show how to improve the result from Section 3 if we are allowed a delay of δ. The main idea is as follows. As before, we maintain suffix trees of exponentially increasing sizes, although only the $O(\log(w/\delta))$ largest of them. As a result there are fewer trees to query, but also an *uncovered* suffix of size $\Theta(\delta)$ of the window for which we do not have any suffix trees. As in Section 3 we denote the part of S covered by suffix trees by s and we denote the uncovered suffix by t. As above, s is segmented into $s_1, \ldots, s_k$.

We will first explain how to solve the problem when all patterns are *long*, that is, $|P| > \delta/4$, and then when all patterns are *short*, that is, $|P| \leq \delta/4$. Finally we show how to combine these solutions. When all the patterns are long we can afford to construct, at query time,

a suffix tree covering t. On the other hand, when all the patterns are short we can do both updates and queries in an offline fashion; we buffer queries and updates until we have approximately $\delta/2$ operations to do, at which point we can afford to construct a suffix tree over t in a deamortized manner. See Figure 4 for an example.

Throughout this section we assume without loss of generality that δ is a power of two. Otherwise we instead use a more restrictive delay of $\delta' = 2^{\lfloor \log \delta \rfloor}$ and achieve the same asymptotic bounds.

4.1 Long Patterns

We first show how to support queries if all patterns have a length $m > \delta/4$. We modify the data structure from Section 3 slightly. The smallest tree now has size $\delta/2$ as opposed to 1, so there are $\Theta(\log w - \log(\delta/2)) = O(\log(w/\delta))$ segments and boundary trees. The uncovered suffix t has length at most δ.

We answer queries the same way as in Section 3.2, with only small modifications. Let P be a pattern of length $m > \delta/4$. As before, let s_i be the smallest and rightmost segment with $|s_i| \geq m$. We find any occurrence within a segment or crossing a single boundary by using the suffix trees over each segment and the boundary trees to the left of s_i, as before. The remaining occurrences we again find by growing suffix trees of exponentially increasing sizes from the right window boundary. The only change is that we now grow the trees faster, as we must also cover t, and we can afford to let the smallest tree have size δ since we have $m > \delta/4$ characters in the pattern to deamortize the work over. As above, let $\mathcal{R}$ be the string covering the $m - 1$ last characters of s_i and extending to the right window boundary, which now also includes t. As $|t| < \delta$ the length of $\mathcal{R}$ is $|\mathcal{R}| < 3m + \delta < 7m$. Assuming $2^\ell \leq m < 2^{\ell+1}$, we build the suffix tree of size $7 \cdot 2^{\ell+1}$ and match P in it, amortized over the characters of P. As we have $m > \delta/4$ characters to deamortize the work over, we only do this for each choice of ℓ where $2^{\ell+1} \geq \delta$, which results in $O(\log w - \log \delta) = O(\log(w/\delta))$ work per character in P. As in Section 3.2 we use recursive range maximum queries to avoid double reporting any occurrences of P that are also in s. As there are also only $O(\log(w/\delta))$ segments and boundary trees we spend $O(\log(w/\delta))$ time per character in P. Note that we answer these queries without delay.

Updates are performed as follows. For each segment of $\delta/2$ characters that arrives we construct the suffix tree over it, deamortized over the next $\delta/2$ characters of S. We merge suffix trees as before, also deamortized over new characters of S. The induction proof from Section 3.4 still works by modifying the base case; the merging of two trees of size $\delta/2$ takes $\delta/2$ characters, at which point another tree of size $\delta/2$ is constructed. The inductive step follows from the fact that δ is a power of two. Thus, we spend expected $O(\log(w/\delta))$ time per update.

4.2 Short Patterns

We now show how to support queries if all patterns have a length $m \leq \delta/4$. We extend the data structure with a buffer of size δ. This buffer will contain queries that we have not yet answered and characters for S that we have not yet processed. The total space is still $O(w + \delta) = O(w)$.

Whenever a character from S arrives we append it to both t and to the buffer. When a pattern arrives we append the full pattern to the buffer, and along with it we store the current position of the right window boundary. Once the buffer has more than $\delta/2$ characters
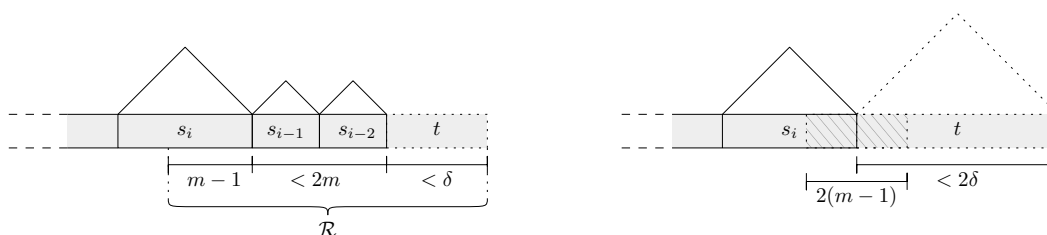
Figure 4 *Left:* Example of a query with a long pattern. Here s_i is the smallest and rightmost segment with $|s_i| \geq m$. Note that the non-indexed suffix t is less than $\delta < 4m$ characters long. *Right:* Example of a query with a short pattern. Note that for short patterns, s_i is always the rightmost segment. Any occurrence in s cross at most a single boundary and is found using the constructed trees. Any occurrence in t is found by the suffix tree over t that we construct when we flush the buffer. Any occurrence that cross the boundary (s, t) is found by the KMP automaton we build over the substring the extends $m - 1$ characters in both directions from the boundary, which is hatched in the figure.

(patterns and text combined) we immediately allocate a new buffer of size δ and *flush* the old buffer as follows. Note that at this point there are strictly less $\frac{3}{4}\delta$ characters in the buffer since each pattern is short.

When we flush the buffer, we first answer all the buffered queries, and then we process all the buffered updates. We deamortize this work over the next $\delta/4$ characters that arrive from either stream. To answer the buffered queries we do as follows. Let $P_1, \ldots, P_\ell$ be the patterns in the buffer, let $m_i = |P_i|$, and let $M = \sum_{1 \leq i \leq \ell} m_i$. We have $M < \delta$. We start by building a suffix tree over t, along with a range maximum query data structure over the suffix array of t. This takes expected $O(\delta)$ time. An occurrence of P_i is either contained in s, or it crosses the boundary (s, t), or it is contained in t. Since P_i is smaller than each segment s_j we can find all the occurrences within s using the suffix trees over the segments and the boundary trees in $O(m_i \log(w/\delta))$ time. To find the occurrences crossing the boundary we build the KMP matching automaton [20] for P_i. In it we match the string that is centered at the boundary (s, t) and extends $m_i - 1$ characters in each direction. This takes $O(m_i)$ time. To find the occurrences in t we match P_i in the suffix tree over t in $O(m_i)$ time. In total, this takes $O(M \log(w/\delta)) = O(\delta \log(w/\delta))$ time for all the patterns, or expected $O(\log(w/\delta))$ time per character when deamortized. Note however, that after P_i arrived more characters from S could have arrived and been appended to t. We must therefore take care not to report any occurrences of P_i that extend past what *was* the right window boundary when P_i arrived. The KMP automaton finds the occurrences in left-to-right order, and in t we avoid reporting too far right using recursive range minimum queries.

Finally, we process each update in the buffer in the order they arrived, using the same procedure as for long patterns. This takes $O(\log(w/\delta))$ time per update and $O(\delta \log(w/\delta))$ time in total. Thus flushing the buffer takes expected $O(\log(w/\delta))$ time per character since we deamortize the expected $O(\delta \log(w/\delta))$ work over $\delta/4$ characters. Since we allocate a new buffer immediately when we begin flushing, we will complete the flush before the next flush begins.

4.3 Both Long and Short Patterns

We now show how to combine the solutions for short and long patterns, to obtain a solution that handles patterns of any length. The data structure is the same as for small patterns above. As above, we append each new character to the buffer. However, whenever we start

streaming a pattern we also proceed as if P were long. If P turns out to fit in the buffer without triggering a flush (which might also happen if P is long), we simply discard the work we did for the long-pattern case. However, if adding P to the buffer results in more than $\frac{3}{4}\delta$ characters being in the buffer, then P must be long. We immediately start flushing the buffer (ignoring the characters related to P) and also continue processing P as a long pattern. Note that since we are potentially streaming a long pattern while batch processing the updates in the buffer, the data structure might change while we are matching in it. However, it only changes when a merge finishes, replacing a pair of suffix trees by a larger tree. If this happens we keep the old trees in memory until we are done processing the pattern, at which point we discard them.

We obtain the following theorem.

▶ **Theorem 4.** *Let S be a stream and let $w \geq 1$ and $\delta \geq 1$ be integers. We can solve the (w,δ)-SSWSI problem on S with an $O(w)$ space data structure that supports* Update *and* Report *in expected $O(\log(w/\delta))$ time per character. Furthermore,* Report *uses additional worst-case constant time per reported occurrence.*

5   Obtaining High Probability

In this section we show how to improve the time bounds to $O(\log(w/\delta))$ with probability $1 - w^{-d}$ for any constant $d \geq 1$.

The expectation in the time bounds in Section 4 comes from the construction of suffix trees (recall that we also build suffix trees at query time). Below, in Lemma 5, we prove that given a string $\mathcal{K}$ of length $k = O(w)$ we can construct the suffix tree over $\mathcal{K}$ in $O(k)$ time with probability $1 - 1/w^{1+\epsilon}$, using additional $O(w/\log w)$ space. We use this algorithm to construct suffix trees during updates and queries, deamortizing them as before and doing $O(\log(w/\delta))$ work per character that arrives. When a new character arrives from S or P, at most $O(\log(w/\delta)) = O(\log w)$ suffix tree constructions will finish. At this point, we finish constructing those trees that did not finish in time, that is, used more more time than what was allotted to them. By the union bound, the probability that any of them fail to finish in time (and thus incurring extra construction cost) is no more than $c \log w/w^{1+\epsilon}$ for some constant c which is no more than $1/w$ for large w. Thus, for each character from S or P we spend $O(\log(w/\delta))$ time with high probability in w. We obtain the $1 - 1/w^d$ probability bound by probability boosting, running $d = O(1)$ independent copies of the construction algorithm simultaneously. The algorithm from Lemma 5 uses additional $O(w/\log w)$ space, but we are never constructing more than $O(\log w)$ suffix trees, so the space usage is $O(w)$ in total.

Furthermore, as mentioned in Section 2, we previously used an FKS dictionary [15] to store the edges to support reporting queries in worst-case constant time per character in the pattern. The construction time of this dictionary is expected linear, so it can no longer be used. Instead we use a dictionary by Dietzfelbinger and Meyer auf der Heide [11]. If there are n elements in the dictionary it supports searches in worst-case constant time and any sequence of $\frac{1}{2}n$ updates takes constant time per update with probability $1 - 1/n^{d'}$ for any constant $d' \geq 1$. We store all the edges of all the suffix trees in one such dictionary. At all times, we keep $\Theta(w)$ dummy-elements in the dictionary to ensure that we get good probability bounds in terms of w, and we choose d' large enough that any sequence of $O(w)$ operations (e.g., the construction of any one of our suffix trees) runs in $O(w)$ time with probability $1 - 1/w^{d+\epsilon}$.

Universal Hashing

Before we prove Lemma 5 we restate some basic facts about universal hashing, introduced by Carter and Wegman [9]. Let $M, m > 0$ be integers, $\mathcal{H}$ be a set of functions $[0, M] \to [0, m]$, and $h \in \mathcal{H}$ be selected uniformly at random. Then $\mathcal{H}$ is *universal* if $P[h(x) = h(y) \mid x \neq y] \leq 1/m$. Let $R \subseteq [0, M]$ and $|R| = r$. It follows from the union bound that h has a *collision* on R with probability at most

$$P[h(x) = h(y) \text{ for some } x \neq y] \leq \sum_{x \neq y \in R} P[h(x) = h(y)] = \frac{r(r-1)}{2} \cdot \frac{1}{m} < \frac{r^2}{m} . \qquad (1)$$

In particular, if $m = r^c$ for constant $c \geq 1$ then h is *injective* (i.e., has no collisions) on R with probability at least $1 - 1/r^{c-2}$. Carter and Wegman gave several classes of universal hash functions from which we can sample a function uniformly at random in constant time.

Fast Suffix Tree Construction

We now prove Lemma 5, showing how to construct our suffix trees in linear time with high probability.

▶ **Lemma 5.** *Given a string $\mathcal{K}$ of length $k \leq 2w$ there is an algorithm that uses $O(k + w/\log w)$ space and constructs the suffix tree over $\mathcal{K}$ in $O(k)$ time with probability $1 - 1/w^{1+\epsilon}$ for some $\epsilon > 0$.*

Proof. Let $\sigma = \{\mathcal{K}[i] \mid i \in [1, k]\} \subseteq \Sigma$ be the alphabet of $\mathcal{K}$. We show how to, in $O(k)$ time, find a function $h : \Sigma \to [1, k^{O(1)}]$ such that h is injective on σ with probability at least $1 - 1/w^{1+\epsilon}$. If h is injective on σ, we can construct the suffix tree over $\mathcal{K}'$ where $\mathcal{K}'[i] = h(\mathcal{K}[i])$ in time $O(\text{sort}(k, k^{O(1)})) = O(k)$ using radix sort. After the tree is constructed we can substitute for the original alphabet in linear time. Therefore, the construction algorithm finishes in $O(k)$ time with probability at least $1 - 1/w^{1+\epsilon}$ (otherwise we make no guarantee on the construction time and we can build the suffix tree in any way).

For some m to be determined later, let $f : \Sigma \to [1, m]$ be chosen uniformly at random from a class of universal hash functions. By Equation 1, the probability that f has a collision on σ is

$$P[f \text{ has collisions on } \sigma] < \frac{|\sigma|^2}{m} \leq \frac{k^2}{m} .$$

We divide into the cases of large trees ($k \geq w^{1/5}$) and small trees ($k < w^{1/5}$). If k is large then $w^{1/5} \leq k \leq 2w$, and we set $m = w^4$ so the probability that f has a collision is at most

$$\frac{k^2}{m} \leq \frac{(2w)^2}{w^4} = \frac{4}{w^2} \leq \frac{1}{w^{1+\epsilon}}$$

for some $\epsilon > 0$. We check whether f is injective by sorting the set $\{(x, f(x)) \mid x \in \sigma\}$ with respect to the $f(\cdot)$-values and checking if two consecutive elements $(x, f(x))$ and $(y, f(y))$ have $x \neq y$ and $f(x) = f(y)$. This takes time $O(\text{sort}(k, w^4)) = O(k)$ using radix sort since $k \geq w^{1/5}$. If f is injective we set $h = f$, concluding the proof of the large case.

If k is small then we allocate an array A of length $w/\log w$ in constant time. For simplicity we assume that A is initialized such that $A[i] = 0$ for all i. This can be avoided using standard constant-time initialization schemes; assume each entry in A contains an arbitrary value initially. We maintain two other arrays B and C such that if we have written a value to $A[i]$

at least once then $A[i]$ is a pointer to some $B[j]$, $B[j]$ is a pointer to $A[i]$, and $C[j]$ stores the value most recently written to $A[i]$. From this we can determine if $A[i]$ has been initialized (check if the pointers match), and if it has not we can initialize it in constant time.

Then we set $m = w/\log w$ such that the probability that f has a collision is no more than

$$\frac{k^2}{m} < \frac{w^{2/5}}{w/\log w} = \frac{\log w}{w^{3/5}} = \frac{\log w}{w^{1/2}} \cdot \frac{1}{w^{1/10}} \leq \frac{1}{w^{1/10}}$$

for $w \geq 16$. We check if f is injective on σ by for each character x in $\mathcal{K}$ setting $A[f(x)] = x$ and seeing if two distinct characters hash to the same index. If f is injective we then arbitrarily assign the values $1, \ldots, |\sigma|$ to the now non-zero indices of A and let $h(x) = A[f(x)]$ (at this point we know σ since it is equal to the number of entries in A that we modified). To boost the probability of success we run this algorithm up to eleven times with independent choices for f. The probability that all of them fail is at most $1/w^{11/10} \leq 1/w^{1+\epsilon}$ concluding the proof for the small case.   ◄

In conjunction with Theorems 3 and 4, this proves Theorem 1.

6   Conclusion and Future Work

We have studied two variants of the streaming sliding window string indexing problem; the timely variant, where queries must be answered immediately, and the delayed variant where a query may be answered at any point within the next δ characters received, for a specified parameter δ. For a sliding window of size w we have given an $O(w)$ space data structure that, in the timely variant, supports updates in $O(\log w)$ time with high probability and queries in $O(\log w)$ time with high probability per character in the pattern; each occurrence is reported in additional constant time. For the delayed variant we improved these bounds to $O(\log(w/\delta))$, where each occurrence is still reported in constant time.

One open problem is whether these bounds can be improved. Another is to find efficient solutions when queries may be interleaved with new updates to the stream. That is, while you are streaming a pattern, new characters of S might arrive that move the current window.

─── **References** ───

1   Amihood Amir and Itai Boneh. Update query time trade-off for dynamic suffix arrays. In *Proc. 31st ISAAC*, volume 181, pages 63:1–63:16, 2020. `doi:10.4230/LIPIcs.ISAAC.2020.63`.

2   Amihood Amir and Itai Boneh. Dynamic suffix array with sub-linear update time and poly-logarithmic lookup time. *CoRR*, abs/2112.12678, 2021. `arXiv:2112.12678`.

3   Amihood Amir, Gianni Franceschini, Roberto Grossi, Tsvi Kopelowitz, Moshe Lewenstein, and Noa Lewenstein. Managing Unbounded-Length Keys in Comparison-Driven Data Structures with Applications to Online Indexing. *SIAM J. Comput.*, 43(4):1396–1416, 2014. `doi:10.1137/110836377`.

4   Amihood Amir, Tsvi Kopelowitz, Moshe Lewenstein, and Noa Lewenstein. Towards real-time suffix tree construction. In *Proc. 12th SPIRE*, volume 3772, pages 67–78. Springer, 2005. `doi:10.1007/11575832_9`.

5   Amihood Amir and Igor Nor. Real-time indexing over fixed finite alphabets. In *Proc. 19th SODA*, pages 1086–1095, 2008. URL: `http://dl.acm.org/citation.cfm?id=1347082.1347201`.

6   Philip Bille, Inge Li Gørtz, and Frederik Rye Skjoldjensen. Deterministic Indexing for Packed Strings. In *Proc. 28th CPM*, volume 78, pages 6:1–6:11, 2017. `doi:10.4230/LIPIcs.CPM.2017.6`.

7 Dany Breslauer and Giuseppe F. Italiano. Near real-time suffix tree construction via the fringe marked ancestor problem. *J. Discrete Algorithms*, 18:32–48, 2013. `doi:10.1016/j.jda.2012.07.003`.

8 Andrej Brodnik and Matevz Jekovec. Sliding suffix tree. *Algorithms*, 11(8):118, 2018. `doi:10.3390/a11080118`.

9 Larry Carter and Mark N. Wegman. Universal Classes of Hash Functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979. `doi:10.1016/0022-0000(79)90044-8`.

10 Richard Cole, Tsvi Kopelowitz, and Moshe Lewenstein. Suffix Trays and Suffix Trists: Structures for Faster Text Indexing. *Algorithmica*, 72(2):450–466, 2015. `doi:10.1007/s00453-013-9860-6`.

11 Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A New Universal Class of Hash Functions and Dynamic Hashing in Real Time. In *Proc. 17th ICALP*, pages 6–19, 1990. `doi:10.1007/BFb0032018`.

12 Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000. `doi:10.1145/355541.355547`.

13 Edward R. Fiala and Daniel H. Greene. Data compression with finite windows. *Commun. ACM*, 32(4):490–505, 1989. `doi:10.1145/63334.63341`.

14 Johannes Fischer and Pawel Gawrychowski. Alphabet-Dependent String Searching with Wexponential Search Trees. In *Proc. 26th CPM*, pages 160–171, 2005. `doi:10.1007/978-3-319-19929-0_14`.

15 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984. `doi:10.1145/828.1884`.

16 Harold N. Gabow, Jon Louis Bentley, and Robert Endre Tarjan. Scaling and related techniques for geometry problems. In *Proc. 16th STOC*, pages 135–143. ACM, 1984. `doi:10.1145/800057.808675`.

17 Yijie Han. Deterministic sorting in $O(n \log \log n)$ time and linear space. In *Proc. 34th STOC*, pages 602–608, 2002. `doi:10.1145/509907.509993`.

18 Shunsuke Inenaga, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. Compact directed acyclic word graphs for a sliding window. *J. Discrete Algorithms*, 2(1):33–51, 2004. `doi:10.1016/S1570-8667(03)00064-9`.

19 Dominik Kempa and Tomasz Kociumaka. Dynamic suffix array with polylogarithmic queries and updates. In *Proc. 54th STOC*, pages 1657–1670, 2022. `doi:10.1145/3519935.3520061`.

20 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast Pattern Matching in Strings. *SIAM J. Comput.*, 6(2):323–350, 1977. `doi:10.1137/0206024`.

21 Tsvi Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *53rd FOCS*, pages 283–292, 2012. `doi:10.1109/FOCS.2012.79`.

22 S. Rao Kosaraju. Real-time pattern matching and quasi-real-time construction of suffix trees (preliminary version). In *Proc. 26th STOC*, pages 310–316, 1994. `doi:10.1145/195058.195170`.

23 Gregory Kucherov and Yakov Nekrich. Full-Fledged Real-Time Indexing for Constant Size Alphabets. *Algorithmica*, 79(2):387–400, 2017. `doi:10.1007/s00453-016-0199-7`.

24 N. Jesper Larsson. *Structures of String Matching and Data Compression*. PhD thesis, Lund University, Sweden, 1999. URL: `http://lup.lub.lu.se/record/19255`.

25 Joong Chae Na, Alberto Apostolico, Costas S. Iliopoulos, and Kunsoo Park. Truncated suffix trees and their application to data compression. *Theor. Comput. Sci.*, 304(1-3):87–101, 2003. `doi:10.1016/S0304-3975(03)00053-7`.

26 Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996. `doi:10.1007/s002360050048`.

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

28 M Senft. Suffix tree for a sliding window: An overview. In *Proc. WDS*, volume 5, pages 41–46, 2005.

29 Martin Senft and Tomás Dvorák. Sliding CDAWG perfection. In *Proc. 15th SPIRE*, pages 109–120, 2008. `doi:10.1007/978-3-540-89097-3_12`.

30 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. `doi:10.1007/BF01206331`.

31 Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th SWAT*, pages 1–11, 1973. `doi:10.1109/SWAT.1973.13`.