流中的滑动窗口字符串索引

Philip Bille* Johannes Fischer IngeLi Gørtz*
phbi@dtu.dk johannes.fischer@cs.tu-dortmund.de inge@dtu.dk
Max Rishøj Pedersen* Tord Joakim Stordalen
mhrpe@dtu.dk tjost@dtu.dk

arXiv:2301.09477v1 [cs.DS] 23 Jan 2023

摘要

给定一个字母表 Σ 上的字符串 S，目标是对 S 进行预处理，以便随后高效地支持模式匹配查询：给定一个模式字符串 P，报告它在 S 中的所有出现位置 P。本文研究流式滑动窗口字符串索引问题 *streaming sliding window string indexing problem*。在此，字符串 S 以数据流的形式逐个字符到达，目标是维护最近的 w 个字符的索引，这些字符称为滑动窗口 *window*，其中参数 w 已指定。任何时刻都可能到达针对模式 P 的模式匹配查询，而且该模式也以数据流的形式逐个字符到达；必须返回当前窗口中 P 的所有出现位置。流式滑动窗口字符串索引问题自然适用于这样的场景：我们希望为数据流中最近的数据（即窗口）建立索引，同时高效地支持模式匹配。

我们的主要结果是一种简单的 $O(w)$ 空间数据结构，该结构以高概率使用 $O(\log w)$ 时间处理来自输入字符串 S 和模式字符串 P的每个字符。报告来自 P 的每个出现位置时，每个报告的出现位置仅需额外的常数时间。与类似场景中的先前工作相比，该结果首次实现了以高概率保证输入数据流中每个字符的高效最坏情况时间。我们还考虑了该问题的延迟变体：查询可以在来自任一数据流的后续 δ 个字符中的任意时刻得到回答。针对这一问题，我们提出了一种 $O(w+\delta)$ 空间数据结构，将上述时间界改进为 $O(\log(w/\delta))$。特别地，当延迟为 $\delta = \epsilon w$ 时，我们得到一种 $O(w)$ 空间数据结构，并可在处理每个字符时使用常数时间。实现该结果的关键思想是一种新颖而简单的、具有独立研究价值的后缀树分层结构，其灵感来源于经典的日志结构合并树。

1 引言

所谓字符串索引问题，是指对字符串 S 进行预处理，将其构建为紧凑的数据结构，以支持高效的后续模式匹配查询；也就是说，给定一个模式字符串 P，报告其在 S中的所有出现位置P。本文介绍一种称为流式滑动窗口字符串索引（*SSWSI*）问题的字符串索引基本变体。在该问题中，字符串 S 以数据流的形式逐个字符到达，而问题是维护一个包含最近 w 个字符的窗口的索引，其中参数 w 为指定参数。在任意时刻，都可能收到针对模式 P 的模式匹配查询，该模式也以数据流的形式逐个字符到达；我们需要报告 P 在当前窗口中的所有出现位置。目标是在高效处理两个数据流中到达的字符的同时，紧凑地维护索引。我们考虑该问题的两种变体：一种是及时变体，每个查询都必须立即回答；另一种是延迟变体，查询可以在来自任一数据流的接下来 δ 个字符到达期间的任意时刻回答，其中参数 δ 为指定参数。精确定义请参见第 1.1 节。

SSWSI 问题自然地涵盖了这样一类场景：我们希望为数据流中最近的数据（即窗口）建立索引，同时支持高效的模式匹配。例如，监控高速数据流系统时，我们无法为整个数据流建立索引，但仍希望支持高效查询。根据具体系统的不同，我们可能需要立即获得查询结果，也可能能够接受一定的延迟，以实现更高效的查询和更新。

此前尚未有人以我们这种精确的表述形式明确研究过 SSWSI 问题，但及时变体中的若干密切相关问题已得到充分研究。特别是，滑动窗口后缀

*由丹麦研究理事会资助，项目编号 DFF-8021-002498。

树问题 [5, 9, 18, 19, 21] 是在每个字符到达时，维护当前窗口的后缀树 （即窗口所有后缀的紧凑字典树）。经过适当扩充后，后缀树可用于高效处理模式匹配查询，从而解决及时型 SSWSI 问题。对于常数规模字母表，这些方案中最优者 [5]能够在每个字符上以常数摊销 时间维护滑动窗口后缀树，同时支持高效的模式匹配查询。其他方案也能达到类似的摊销时间界限。由于新字符到达后，显式维护后缀树可能需要进行 Ω(w) 次更改，因此无法避免这种摊销。

另一个密切相关的问题是在线字符串索引问题 [1, 2, 3, 4, 10, 15, 16, 17]。这里的目标是每次处理 S 一个字符（按从左到右或从右到左的顺序），同时逐步为迄今读入的字符串建立索引。这些方案中最优者在常数规模字母表 [17] 下 O每个字符只需常数时间，而在一般字母表 [15] 下构建索引则需要（log log n +log log $|\Sigma|$）时间。这些方案都高度依赖按从右到左的顺序处理字符串，以避免如上所述，因追加操作而导致的后缀树更新所需的固有线性时间。因此，它们无法应用于我们的从左到右的流式处理场景。另一种做法是将这些方案应用于字符串 S的逆序，但这样一来，每个模式都必须按逆序处理，在我们的场景中同样行不通。此外，值得注意的是，这些方案为迄今读入的整个字符串建立索引。目前尚不清楚能否对它们进行改造，以便高效地为滑动窗口建立索引。

我们的主要结果是为及时变体和延迟变体下的 SSWSI 问题提供了一种高效的解决方案。令 w 表示窗口的大小。对于及时变体，我们提出了一种字符串索引，它使用$O(w)$空间，并在 $O(\log w)$时间内处理来自数据流 S 的一个字符。每个模式匹配查询P 也受支持，处理每个字符需要 $O(\log w)$时间；收到 P的最后一个字符后，还需额外花费 O(occ)时间，其中 occ 是当前窗口中 P 的出现位置数。该索引采用随机化方法，且两项时间界均以高概率成立。结果适用于任意整数字母表（其大小不必是 n的多项式）。与先前工作相比，我们将数据流中每个字符对应的最坏情况时间界从 Ω(w) 改进为 $O(\log w)$，且以高概率成立。这对于上述应用尤为重要，例如高速数据流系统。我们将解决方案推广到了该问题的延迟变体。如果允许在回答每个查询前延迟 δ ，则所需空间为 $O(w + \delta)$，同时可将上述时间界改进为 $O(\log(w/\delta))$。特别是，如果允许延迟 $\delta = \epsilon w$ ，其中 $\epsilon > 0$为任意常数，则可达到线性空间和最优常数时间（报告出现位置仍需 O(occ)时间，且报告时间不计入延迟）。我们的所有结果均适用于字长为对数级的字 RAM。

实现这一结果的关键思路，是借鉴日志结构合并树 [20]，采用一种新颖而简单的后缀树分层结构。我们不再为窗口维护一棵单独的后缀树，而是维护一组大小按指数级增长、覆盖当前窗口的后缀树。我们展示了如何通过增量式“合并”后缀树，在数据流中有新字符到达时高效维护该结构，同时支持在窗口内高效执行模式匹配查询。

1.1 设置与结果

我们将问题形式化定义如下。设 S 是由任意整数字母表 Σ上的字符构成的数据流。给定整数参数 $w \geq 1$ 和 $\delta \geq 0$, *δ-delayed* 流式滑动窗口字符串索引 *((w, δ)-SSWSI)*问题要求维护一种数据结构；在接收到 S的前 i 个字符后，该结构支持

- 报告(P)：在额外接收 δ 个字符之前，报告 P在 $S[i - w + 1, i]$ 中的所有出现位置，无论字符来自哪个数据流。
- 更新()：处理数据流中的下一个字符 S。

在 Report(P) 查询中，模式字符串 P 也会被流式处理。当 P 被流式处理时，它会中断数据流 S，并逐个字符到达；在 S 的流式处理恢复之前， P的所有字符都会到达。此外，我们不假设在 P 的最后一个字符到达之前就知道它的长度。延迟从 P 的最后一个字符到达之后开始计算。来自 S 的每个字符以及新模式中的字符都会计入延迟，而报告出现位置则不计入（否则，如果出现位置超过 δ 个，就无法及时回答查询）。

我们将及时流式滑动窗口字符串索引（w-*SSWSI*）问题定义为(w,0)-SSWSI，也就是说，模式的最后一个字符一到达，就必须立即回答查询。

我们给出以下一般性的主要结果。

定理 1. 令 S 为数据流，$w \geq 1$ 和 $\delta \geq 0$ 为整数。我们可以在 S 上解决 *(w, δ)-SSWSI* 问题，使用一个空间为 $O(w+\delta)$ 的空间数据结构，以高概率支持每个字符耗时 $O(\log \frac{w}{\delta+1})$ 的更新和报告。此外，报告每个出现位置还需额外的最坏常数时间。

这里，高概率意味着对于任意常数 d，概率至少为 $1-\frac{1}{w^d}$。定理 1 给出了延迟参数 δ 上的权衡。具体而言，在定理 1 中代入 $\delta = 0$，我们得到一个及时型 SSWSI 问题的解，其空间开销为 $O(w)$，对每个字符执行更新和报告的时间为 $O(\log w)$。与先前工作 $[5, 9, 13, 18, 19, 21, 22]$ 相比，这将更新操作的最坏情况界从 $\Omega(w)$ 改进为 $O(\log w)$，且以高概率成立；同时还取消了对字母表的限制。另一方面，在定理 1 中取 $\delta = \epsilon w$，并令 $\epsilon > 0$ 为常数，我们得到一个延迟型 SSWSI 问题的解，其空间开销为 $O(w)$，并且以高概率实现每个字符的最优常数时间。我们的所有结果均适用于字长为对数级的字 RAM。

1.2 技术

我们通过以下方法得到及时变体的结果，但不提供高概率保证。我们始终维护至多 $\log w$ 棵互不重叠且共同覆盖窗口的后缀树。这些树按照日志结构合并技术 $[20]$, 组织，其中最右侧的树最小，树的大小从右向左呈指数增长。每当有新字符到达时，我们就将其后缀树追加到数据结构的右侧。每当相邻的两棵树大小相同时，我们就通过构建一棵同时覆盖这两棵树的新后缀树来“合并”它们。来自S 的每个字符至多参与 $\log w$ 次合并，而每次合并需要期望线性时间，因此我们在 S 中每个字符花费期望摊销 $O(\log w)$ 时间。我们通过在后台进行合并时暂时保留两棵树，来实现更新的去摊销化。注意，对于每对相邻的后缀树，我们还存储一棵大致覆盖它们二者的后缀树，称为边界树 （详见下文）。

我们通过查询这些树中的每一棵，来查找模式 P 在窗口中的出现位置；对于 P中的每个字符，耗时为$O(\log w)$。对于大于 $|P|$ 的相邻树对，我们使用边界树查找P 从一棵树跨入另一棵树的出现位置。其余树覆盖了长度为 $O(|P|)$的窗口后缀，我们会扩展一棵后缀树，以便在查询时回答该后缀中的查询。我们的数据结构在窗口左侧有一些“溢出”，因此我们使用区间最大值查询，仅报告起始位置位于窗口内的出现位置。

我们将此解决方案推广，以纳入 δ 的延迟，方法如下。我们存储及时解决方案中的 $O(\log(w/\delta))$棵最大树，并让窗口中长度为 $\Theta(\delta)$的后缀不被后缀树覆盖。我们按如下方式回答查询。如果 $|P| > \delta/4$，我们就称 P 为长模式；否则，它就是短模式。对于长模式，我们采用与及时情形相同的方法；此时在查询时扩展的后缀树还必须包含未覆盖后缀，但其大小仍为 $O(|P|)$，因为窗口未覆盖部分的长度为 $O(\delta) = O(|P|)$。我们将展示如何在 P中的每个字符耗时 $O(\log(w/\delta))$ 的情况下完成此操作。对于短模式，我们利用它们短于延迟这一点，暂时缓冲查询，之后再批处理。我们最多缓冲 $O(\delta \log(w/\delta))$ 的工作量，并将其去摊销化到 $\Theta(\delta)$ 个字符上，从而获得与长模式相同的界。更新也能在相同的界内完成，因为窗口中的每个字符在离开窗口之前，最多参与 $O(\log(w/\delta))$ 次合并。

最后，我们通过证明以下结论改进时间界：对于窗口中的任意子串 S'，我们都可以以概率 $1-w^{-d}$，在 $O(|S'|)$ 时间内构造 S' 上的后缀树，其中任意常数 $d > 1$ 均成立。为此，我们将 S' 的字母表 $\Sigma' = \{c \in S'\}$ 约简到秩空间 $\{1, 2, \ldots, |\Sigma'|\}$，然后 Farach-Colton 等人 $[8]$ 的算法便可在线性最坏时间内构造后缀树。对于较长的字符串（$|S'| > w^{1/5}$），我们选取一个哈希函数 $\Sigma \to [0, w^c]$，它以高概率在 S' 上是单射，然后使用基数排序在线性时间内将其约简到秩空间。对于较短的字符串（$|S'| \leq w^{1/5}$），我们选取一个哈希函数 $\Sigma \to [0, w/\log w]$，它以（近乎）高概率是单射，并用它手动构造映射，在 $O(S')$ 时间内将其映射到秩空间。该映射算法还需要 $O(w \log w)$ 空间，但我们同时至多构造 $O(\log w)$ 棵后缀树，因此总空间为线性。

1.3 概述

在第 2 节中，我们介绍预备知识，包括一些关于后缀树的有用事实。在第 3 节中，我们给出及时型 SSWSI 问题的一种解法，该解法支持每个操作的期望时间为对数级

每个字符。在第4节中，我们展示如何将其推广以纳入延迟；在第5节中，我们展示如何获得良好的概率保证，并证明定理 1。

2 预备知识

给定一个长度为 n 、取自字母表 Σ 的字符串 X ，第 i 个字符记作 $X[i]$ ，从 $X[i]$ 开始到 $X[j]$ 结束的子串记作 $X[i,j]$。形式为 $X[i,n]$ 的子串是 X 的后缀。

X 的一个段是一个区间 $[i,j] = \{i, i+1, \ldots, j\}$，其中 $1 \le i \le j \le n$。我们有时也将段称为字符串，即段 $[i,j]$ 指字符串 $X[i,j]$。此定义与"子串"的区别在于它明确指定了位置；即使 $X[1,2] = X[3,4]$，我们也有 $[1,2] \neq [3,4]$。 X 的分段是将 X 分解为互不相交且覆盖它的段。例如，$x_1 = [1,i]$ 和 $x_2 = [i+1,n]$ 是将 X 分为两部分的一种分段。段 x_1 和 x_2 是相邻的，因为 x_2 紧接在 x_1 结束的位置之后开始；对于一对相邻段，我们将边界（x_1， x_2）定义为 i 与 $i+1$ 之间的隐含位置。

对于 X， [23] T 是 $X\$$ 的所有后缀构成的紧凑字典树，其中 $\$ \notin \Sigma$ 按字典序小于字母表中的任何字母。每个叶节点对应 X 的一个后缀，叶节点从左到右按字典序递增排列。 $O(n)$ 空间：通过用 X 中的两个索引隐式表示与每条边关联的字符串， [8] 可以表示后缀树。Farach-Colton 等人 [8] 证明，构建 T 的最优构建时间为 sort(n, $|\Sigma|$)，即对全集 Σ 中的 n 个元素进行排序所需的时间。对于 $\Sigma = \{0, \ldots, n^c\}$ 形式的字母表，其中 $c \ge 1$ 为常数（也就是说，整数字母表的大小关于 n 为多项式），这意味着可以使用基数排序在最坏情况下以 $O(n)$ 时间构建 T。对于更大的字母表，我们可以通过哈希在期望线性时间内将问题归约到多项式规模的情形，并在期望线性时间内构建 T （详情参见第5节）。

L 的后缀数组是一个数组，其中 $L[i]$ 是 X 中按字典序第 i 小的后缀的起始位置。注意， $L[i]$ 对应于 T 中从左到右数的第 i 个叶节点。此外，设 v 是 T 中的一个内部节点，并设 s_v 是从根节点到 v 的路径所表示的字符串。 v 的后代叶节点恰好对应于 X 中以 s_v 开头的后缀，这些叶节点在 L 中对应一个连续范围 $[\alpha,\beta]_v$。

我们对后缀树进行扩充，以支持高效的模式匹配查询，具体如下。首先，我们使用广为人知的 FKS 完美哈希方案 [11] 来存储后缀树的边，以便能够在最坏情况下的常数时间内，针对任意节点判断是否存在一条与字符 $a \in \Sigma$ 匹配的出边。注意，这种构造的期望时间为线性。此外，我们还在 L 上构建一个范围最大值查询数据结构。该数据结构支持区间最大值查询，即给定一个范围 $[\alpha,\beta]$ ，返回使 $L[j]$ 最大化的 $j \in [\alpha,\beta]$ 。它也支持以类似方式定义的区间最小值查询。该数据结构可以在线性时间内构建，并支持在常数时间内进行查询 [12]。最后，我们在线性时间内对后缀树进行预处理，使得每个内部节点 v 都存储 L 中的范围 $[\alpha,\beta]_v$ ，该范围对应于 s_v 的出现位置。

我们可以利用这种结构，在 $|P| + \mathrm{occ}$ 时间内高效地找到 P 在 O 中的所有出现位置，或者在 $|P|$ 时间内找到 P 在 O 中最靠左和最靠右的出现位置。字符串 P 的 轨迹点 是满足 P 是 s_v 前缀这一条件的最小深度节点 v 。沿后缀树向下遍历，逐个匹配 P 中的字符，即可找到轨迹点；借助字典，每个字符的匹配在最坏情况下只需常数时间。找到 v 后，我们便能在 $O(\mathrm{occ})$ 时间内报告 $[\alpha,\beta]_v$ 中的所有出现位置。或者，我们可以对范围 $[\alpha,\beta]_v$ 执行范围最大值查询，从而在常数时间内找到 P 最靠右的出现位置；该查询会返回使 字符串位置 $L[j]$ 达到最大值的 $j \in [\alpha,\beta]_v$。我们还可以通过执行范围最小值查询找到最靠左的出现位置。

最后请注意，利用均匀分配工作量这一标准技术，可以对具有 期望 运行时间的算法进行去摊销化。具体而言，如果某个算法的期望运行时间为 λn ，我们可以在 $n-1$ 步内完成最坏情况下的 λ 工作；根据期望的线性性，最后一步仅剩下期望为 λ 的工作量。

3 TimelySSWSI 问题

这里我们为及时变体提出了一种解决方案，其期望表现与定理 1 中的界相匹配。第5节将展示如何以高概率达到这些界。本节中，我们始终假设不会

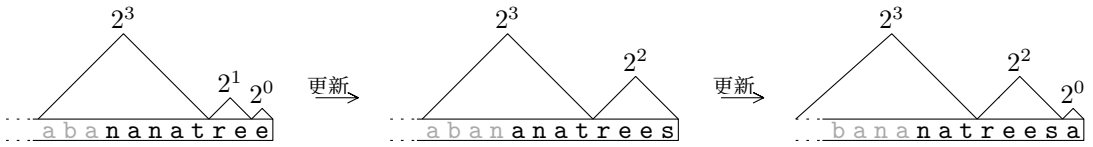


图1：窗口大小为 $w = 8$ 时更新数据结构的示例。这里用构建在各段上的后缀树来说明这些段。窗口之外的字符以灰色显示。当字符 s 到达时，我们构建一棵大小为 1 的新后缀树，然后立即将它与覆盖 e 的现有大小为 1 的后缀树合并，得到覆盖 es 的大小为 2 的后缀树；接着再将其合并，得到覆盖 rees 的最终大小为 4 的后缀树。收到 a 后，我们又有了一棵大小为 1 的后缀树。请注意，再经过三次更新后，大小为 8 的后缀树将不再与窗口重叠，因此会被丢弃。

丢失基因 rality，即 w 是 2 的幂 . 第 3 节 .3 简要提及了如何推广到任意 w。主要思路如下。我们维护 S 的一个长度至少为 w 的后缀。该后缀被划分为至多 $\log w$ 段，这些段的大小各不相同，均为 2 的幂，并从右向左递增。我们存储的后缀长度至多为 $2^0 + \ldots + 2^{\log w} = 2w - 1$。每当有新字符到达时，我们就在数据结构中追加一个大小为 1 的新段，并合并大小相同的段，直到所有段的大小再次各不相同。当最大的段不再与窗口相交时，我们也会将其丢弃。对于每一段，我们都存储一棵后缀树；对于每一对相邻段，我们还存储一棵大致覆盖这两段的边界树（见下文）。为了支持查询，我们会查询每一段对应的后缀树，以及每一棵边界树。对于大于模式的段，边界树足以找到跨越相应边界的出现位置。其余的树覆盖 S 的一个后缀，该后缀的长度为 $O(|P|)$，我们会在查询时构建一棵后缀树，以找到此后缀中的出现位置。

3.1 数据结构

在任意时刻，该数据结构都包含一个长度为 $w \le |s| \le 2w-1$ 的 S 的后缀 s，以及将s 分成至多 $\log w$ 个段的分段。具体而言，如果对整数 $b_1 < \ldots < b_k$ 有 $|s| = 2^{b_1} + \ldots + 2^{b_k}$，那么我们有如下分段：$s_1$、……、$s_k$，其中 $|s_i| = 2^{b_i}$，且 s 是字符串 s_k、s_{k-1}、……、s_1 按此顺序连接而成。集合 $\{b_1, \ldots, b_k\}$ 是唯一的，并且对应于 $|s|$ 的二进制编码中的 1 位。图 1 展示了三种不同的配置。

对于每个段 s_i，我们存储其上的后缀树 T_i，其中字符串为 s_i；此外，还存储一个针对 s_i 的后缀数组的区间最大值查询数据结构。对于每个边界 (s_{i+1}, s_i)，我们存储边界树 边界树 B_i，它是以该边界为中心、向两个方向各延伸 $|s_i|$ 个字符的子串上的后缀树。我们为 B_i 增添一个额外的数据结构，用于报告跨越边界的出现位置。令 BL_i 为对应于 B_i的后缀数组。我们将修改后的后缀数组 BL'_i定义为

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

我们在BL'_i上存储一个区间最大值查询数据结构。每个数据结构占用 $O(s_i)$ 空间，因此整个数据结构共占用 $O(s)$ $= O(w)$ 空间。

我们注意到该数据结构的几个性质。令 $S[n]$ 为最近到达的字符，$W_n = S[n-w+1, n]$ 为当前窗口。那么，由于 $|s| \ge w$，W_n 是 s 的一个后缀。最大的段（也是最靠左的段）s_k 始终具有 $2^{\log w} = w$ 大小；它不可能更大，因为 $\log w$ 位足以表示$|s| \le 2w-1$；它也始终存在，因为 $|s| \ge w$ 无法用 $\log w - 1$ 位表示。出于同样的原因，s_k 始终至少部分与 W_n相交，并且每个 s_1、……、s_{k-1} 都完全包含在 W_n中。

3.2 查询

其思路如下，图 2 给出了示例。对于完全包含在某一段中的模式 P 的任何出现位置，都可以通过该段上的后缀树找到。在最左片段中，我们必须注意，不要报告任何起始位置早于左窗口边界的出现位置。同样，任何

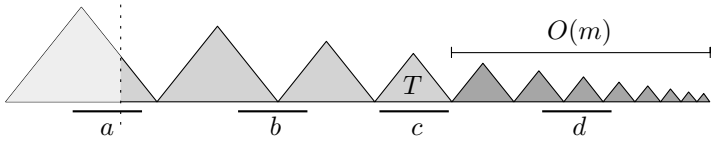


图 2：说明了我们如何查询长度为 m 的模式 P。标记为 a、b、c 和 d 的线表示 P 的出现位置。分段由各段上的树示意。最左侧的窗口边界以一条垂直虚线标出。请注意，最左片段仅有一部分与窗口相交。树 T 标记了大于 m 的最小段。T 右侧的段都小于 P，因此它们至多覆盖 $m+m/2+\ldots+1=O(m)$ 个字符。要查询 P，我们在每段上的树和每棵边界树中匹配 P，并且在查询时还会为小于 P 的段构建一棵后缀树。我们找到 b，因为相应的边界树足够大。我们找到 c，因为它完全包含在某一段中。我们在查询时构建的后缀树中找到 d。请注意，a 不在窗口内；我们通过递归使用区间最大值查询来找到 P 最右侧的出现位置，从而避免报告它。

仅跨越一个边界，且距离窗口末端足够远的出现位置，都可以在相应的边界树中找到。其余出现位置不包含在数据结构中的任何树内（原因可能是它们跨越了多个边界，也可能是它们只跨越一个边界（s_{i+1}, s_i），但起始位置比该边界向左 $>|s_i|$ 个字符）。不过，这些出现位置都位于一个大小为 $O(m)$、终止于位置 $S[n]$ 的子串中，因此我们在查询时构建一棵后缀树来查找这些出现位置。

令 P 为正在查询的长度为 m 的模式，$S[n]$ 为最新到达的字符，并令 W_n、分段 $s_1,\ldots,s_k$ 的后缀 $s_、$ 以及索引 $b_1<\ldots<b_k$ 如上所定义。如前所述，P 在 W_n 中的任何一次出现，要么完全包含在某一段内，要么跨越两个相邻段之间的边界。我们将分别说明如何处理这两种情况。

完全包含在某一段内 固定一个特定的段 s_i。随着 P 的每个字符到达，我们在 T_i中对其进行匹配。当最后一个字符到达时，我们会得到后缀数组中一个（可能为空的）区间 $[\alpha,\beta]$，该区间对应于 s_i 中 P的出现位置。如果 s_i 不是最左片段，那么它完全包含在 W_n中，我们就报告所有出现位置。否则，$s_i=s_k$ 是最左片段，它可能仅与 W_n部分重叠，并且其中可能包含不在窗口内的 P 的出现位置。不过请注意，W_n 与 s_k 的交集是 s_k的一个后缀。因此，如果 P 在 s_k开始于 W_n 内部出现，它也会在 W_n内部结束。我们按如下方式找出所有这类出现位置。令 L_k 为 s_k的后缀数组。如第 2 节所述，我们通过对 L_k中的范围 $[\alpha,\beta]$ 执行范围最大值查询，找到 P 最右侧出现位置的索引 j。如果 $L_k[j]$ 不在 W_n 内，则没有任何出现位置位于其中，我们就完成了。否则，我们在 $[\alpha,j-1]$和 $[\beta,j-1]$上递归。对所有段的树进行匹配，每个 P字符所需的总时间为 $O(\log\ w)$。报告每个出现位置需要常数时间，因为范围最大值查询可以在常数时间内完成。

跨越边界 我们展示如何报告跨越边界的 P 的出现位置。其主要思路如下，如图3所示。令 s_i 为满足 $|s_i|\geq m$ 的最小段。考虑位于 s_i 左侧的任意边界（s_{j+1},s_j），即满足 $j\geq i$ 的边界。由于这两个段的大小都至少为 $|s_i|\geq m$，边界树 B_j 会从该边界向两个方向至少延伸 m 个字符。因此，所有跨越该边界的 P 出现位置都包含在 B_j 中，并且它们都不会同时跨越其他边界。现在考虑 s 的后缀 $\mathcal{R}$，它包含 $m-1$ 个 s_i 的末尾字符，并延伸至 s 的末尾。该子串包含所有其他跨越边界的出现位置。此外，$\mathcal{R}$ 中的所有出现位置至少跨越一个边界，因为 $\mathcal{R}$ 中单个段的最长连续部分，是 s_i 中的 $m-1$ 个字符。注意，$\mathcal{R}$ 的长度至多为 $m-1+|s_{i-1}|+|s_{i-1}|/2+\ldots+1<m-1+2|s_{i-1}|<3m$，因为 $|s_{i-1}|<m$。因此，P 跨越边界的出现位置数量，等于 $\mathcal{R}$ 中的出现位置数量，加上跨越边界（s_k, s_{k-1}）、(s_{k-1},s_{k-2}) ..... (s_{i+1},s_i) 的出现位置数量。

在足够大的边界树中查找出现位置的算法如下。固定一个

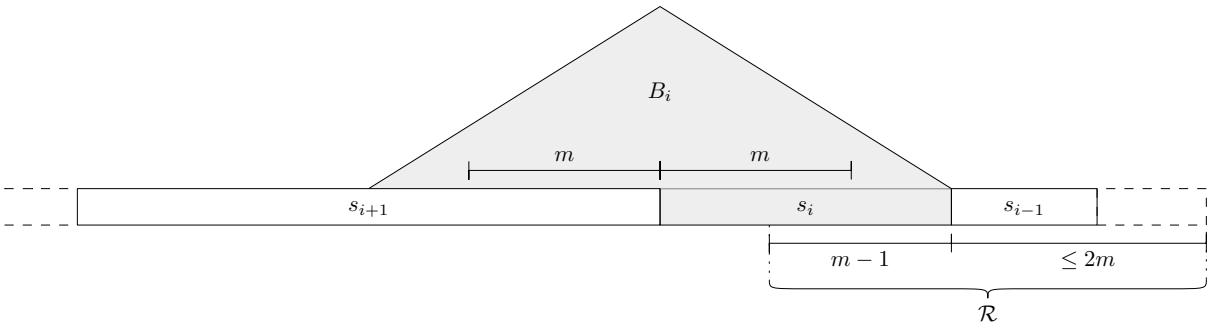


图3： s_i 是满足 $|s_i| \geq m$的最小段。对于每个边界（s_{j+1}， s_j），其中$j \geq i$，树 B_j 足够大，可以找出跨越该边界的 P 的所有出现位置。所有其他跨越边界的 P出现位置都必须位于 $\mathcal{R}$中； $\mathcal{R}$是覆盖 s_i 最右侧 $m-1$ 个字符并延伸至窗口末尾的字符串。 $\mathcal{R}$ 的长度不超过 $m-1+|s_{i-1}|+|s_{i-1}|/2+\ldots+1 < 3m$。

边界 (s_{x+1}， s_x)。我们会在 P 中的每个字符到达时，将其与 B_x 匹配。最后一个字符到达时，我们便知道 $|s_x| \geq m$，也知道边界树中与 P 在 $[\alpha, \beta]$ 中的出现位置相对应的区间。如果 $|s_x| \geq m$ （因此 $x \geq i$），则按如下方式报告这些出现位置。如上所述，我们执行范围最大值查询，以找到使 $BL'_x[j]$ 最大的 j。如果 $BL'_x[j] = -\infty$，那么 P的所有出现位置都始于 s_x，并且没有出现位置跨越边界。否则，$BL'_x[j]$ 对应于 s_{i+1}中 P 最靠右的出现位置的起始位置。由于 P 已经全部到达，现在我们知道了 m，因此当且仅当 $BL'_x[j] \geq |s_x| - m + 2$ 时，该出现位置才会跨越边界（回想一下， B_x 从边界向两个方向各延伸 $|s_x|$个字符）。如果它没有跨越边界，那么其他出现位置也都不会跨越边界。否则，我们报告 $BL'_x[j]$，并在 $[\alpha, j-1]$和 $[j+1, \beta]$ 上递归，以查找其余的出现位置。在所有边界树中匹配 P，每个字符的总体耗时为 $O(\log w)$；通过范围最大值查询报告每个出现位置则只需常数时间。

现在我们说明如何以相同的界限找出 $\mathcal{R}$ 中 P 的出现位置。假设我们知道，对于某个整数 ℓ，$2^\ell \leq m < 2^{\ell+1}$。我们针对$s$最后的 $3 \cdot 2^{\ell+1}$ 个字符构建后缀树，并在接收 P的前 $2^{\ell-1}$ 个字符期间完成去摊销化。在接下来的 $2^{\ell-1}$ 个字符中，我们在树中匹配P，每从 P读入一个新字符，就匹配两个字符。然后，当第 2^ℓ个字符到达时，我们就追上了数据流 P，并随着剩余的 $m - 2^\ell$ 个字符到达逐一进行匹配。当最后一个字符到达时，我们已在大小至少为 $3m$的树中匹配完 P，于是可以开始报告出现位置。请注意，我们高估了树的大小；树中可能包含一些位于 s_i中的 P出现位置。为避免报告这些位置，我们还在后缀数组上构建区间最大值查询数据结构，以便使用递归范围最大值查询。去摊销化后，我们以每个 P字符期望常数时间的速度构建树。匹配 P 也只需每个字符常数时间。我们知道 $m \leq w$，因此对 ℓ的 $\log\ w$ 种不同选择同时运行此算法，在 P 中每个字符耗时为期望 $O(\log\ w)$。注意，这些树总共占用 $O(w)$空间，因为各树的空间总和是几何级数，最大项为$O(w)$。

3.3 摊销更新

我们将展示如何以摊销 $O(\log w)$ 时间支持更新。设 $S[n]$ 为最新到达的字符，并按照数据结构的描述，设 $b_1 < b_2 < \ldots < b_k$ 为 $|s|$的二进制编码中值为 1 的索引位置。当新字符 $c = S[n+1]$ 到达时，我们更新 s 和分段$s_1 \ldots s_k$，以创建带有新分段 $s'_1, \ldots s'_k$的新后缀 s'。示例见图1。

如果 $|s| < 2w-1$，那么我们设置 $s' = sc$。 s'的分段对应于 $|s'| = |s| + 1$唯一的二进制编码，因此我们以类似“二进制递增”的方式更新分段。可以按如下方式进行：在 c上创建一个大小为 1 的新段。如果原本没有大小为 1 的段，那么添加这个新段即可。否则，我们将两个大小为 1 的段合并 （见下文），创建一个大小为 2 的段。这个过程会不断向后级联，直到达到分段中不存在的大小 2^b （即最小索引 $b \notin \{b_1, \ldots, b_k\}$）。此时，我们用覆盖 s'最后 2^b 个字符的 s'_1，替换所有这些段： s_{b-1}、.....、 s_1。 s'剩余的段为

与这些段相同： s_{b+1}、……、s_k。如果 $|s| = 2w - 1$，那么每种大小 2_0、2_1、……、$2^{\log w}$的段都各有一个。由于这些段从左到右逐渐变小，最右边的 $\log\ w - 1$ 个段覆盖了 s最后的 $2^0 + \ldots + 2^{\log w - 1} = w - 1$ 个字符。因此，在 c到来后，大小为 $2^{\log w} = w$的最左片段不再与窗口相交。我们将 $s' = s[w+1, |s|]c$设为相应的值以移除它，并按照上述方式更新分段。

设 a、b 和 c 是三个按此顺序排列的相邻段。要合并 b和 c，我们将它们合并为一个同时覆盖二者的新段 d，在 d 上构造后缀树，并在 d 的后缀数组上构造区间最大值查询数据结构。此外，由于 a 和 d 现在相邻，我们还会为边界 (a，d) 构造跨边界后缀树，该树在边界两侧各延伸 $|d|$ 个字符。构造所有这些数据结构的期望时间为 $O(|d|)$（见第 2 节）。因此，每当字符移入一个更大的段时，每个字符所需的期望时间为常数。每个字符离开窗口前最多包含在 $\log w$ 个段中，因此每个字符的摊销更新时间为期望 $O(\log w)$。

请注意，除了最后一次合并之外，其余合并对于实际计算 s'_1 都没有必要；在摊销情形下，我们只需确定级联将在哪里结束，并立即在相应的段上构造后缀树。不过，级联合并将在去摊销变体中发挥作用。

另请注意，如果 w 不是 2 的幂，我们可以采用类似的方案：允许同时存在两棵大小为 $2^{\lfloor \log w \rfloor}$的树，或一棵大小为 $2^{\lceil \log w \rceil}$的树。在这两种情况下，何时移除最左片段都有一些简单的边界情况需要处理。

3.4 去摊销更新

现在我们说明如何对更新进行去摊销化。遗憾的是，先前的构造无法直接去摊销化，因为 Farach-Colton 等人的后缀树构造算法 [8] 需要访问整个字符串。因此，如果一个新字符 c 引发一连串合并，并由此产生一个大小为 2^i 的新段，那么当 c 到来时，我们必须在该段上构造后缀树。

相反，我们对该结构稍作修改。当两个大小为 2^i 的段相邻时，我们暂时保留这两段，同时将合并它们的开销摊销到 S接下来的 2^i 个字符上，每处理一个字符执行期望常数次操作。请注意，查询不受影响，只有跨越边界报告出现位置这一点例外；现在可能会有两个相邻的同尺寸段 s_{i+1} 和 s_i，它们都是大小至少为 $|P|$的最小段。在这种情况下，后缀 $\mathcal{R}$ 只向右侧段 s_i延伸 $m - 1$ 个字符。(s_{i+1}，s_i) 的边界树足以报告所有跨越该边界的出现位置，因为两个段的大小都至少为 $|P|$。此外，$\mathcal{R}$的长度可能会变为两倍，因此我们会调整查询时扩展的树的常数。

为了界定更新所需的时间，我们证明在任意时刻构造的后缀树至多有 $\log w$ 个，由此可知更新所需的期望时间为 $O(\log w)$。为此，我们证明以下引理。

引理 1. 当大小为 2^i 的段构造完成时，每种大小为 $2^{i-1}, \ldots, 2_0$ 的段恰好各有一个。

证明. 证明采用关于 i 的归纳法。对于 $i = 1$，当两个大小为 1 的段相邻时，我们会在下一个字符 c 从 S 到达时将它们合并。这样会得到一个大小为 2 的段，以及一个包含 c 的大小为 1 的段，从而证明了基础情形。

归纳地，考虑两个大小为 2^i 的段首次相邻的时刻。根据归纳假设，这两个段右侧有大小分别为 $2_0, 2_1, \ldots, 2^{i-1}$ 的段各一个。要构造另一个大小为 2^i 的段，我们必须先再接收一个字符，这会触发一次合并，并最终级联经过这些 $i - 1$ 个段。要使这种情况发生，还必须从 S 收到 $1 +(2^0 + 2^1 + \ldots + 2^{i-1}) = 2^i$ 个字符，其中 1 表示下一个到达的字符，而2^j 表示第 j 次合并所摊销的字符数。不过，此时两个大小为 2^i 的段已经合并完毕，因此我们构造出了两个新段，一个大小为 2^{i+1}，另一个大小为 2^i。根据归纳假设，大小分别为 $2_0, \ldots, 2^{i-1}$ 的段也各有一个，证明得证。

□

引理 1 表明，相邻的同尺寸段最多只有两个，因此对于每种段尺寸 2_0，2_1，……，$2^{\log w}$，最多只会发生一次合并过程。为此，考虑两个尺寸为 2^i 的段 a 和 b 首次相邻的时刻。此时，$2^0 + 2^1 + \ldots + 2^{i-1} = 2^i - 1$

在 b 右侧有 个字符。当下一个尺寸为 2^i 的段 c 到达时，其右侧也有 2^i-1 个字符。但此时 b 的右侧有 $|c|+2^i-1=2^i+2^i-1$ 个字符。因此，在此期间必定到达了 2^i 个新字符，而 a 和 b 的合并也就完成了。

我们得到如下定理。

定理 2. </style}设 为一个数据流，为一个整数。我们可以使用一个 (w) 空间数据结构解决 w-*SSWSI* 问题，并支持更新和报告操作，其每个字符的期望时间为 $O(\log w)$ 。此外，报告每个出现位置所需的额外最坏情况时间为常数。

4 延迟型 SSWSI 问题

本节将说明，如果允许延迟 δ，如何改进第 3 节中的结果。主要思路如下。与之前一样，我们维护大小呈指数增长的后缀树，不过只维护其中最大的 $O(\log(w/\delta))$ 棵。因此，需要查询的树更少，但窗口中也会有一个大小为 $\Theta(\delta)$ 的未覆盖后缀，对应部分没有任何后缀树。与第 3 节一样，我们用 s 表示 S 中由后缀树覆盖的部分，并用 t表示未覆盖后缀。和上文一样， s被划分为 s_1、.....、 s_k。

我们首先说明当所有模式都是长模式时如何解决该问题，也就是说， $|P|>\delta/4$；然后说明当所有模式都是短模式时如何解决，也就是说， $|P|\le\delta/4$。最后，我们将展示如何组合这些解。当所有模式都是长模式时，我们可以在查询时构建一棵覆盖 t 的后缀树。另一方面，当所有模式都是短模式时，我们可以以离线方式同时处理更新和查询；我们将查询和更新缓冲起来，直到待处理操作约有 $\delta/2$ 个，此时便可以用去摊销化的方式为 t 构建一棵后缀树。示例见图 4。

本节中不失一般性地假设 δ 是 2 的幂。否则，我们改用更严格的 $\delta'=2^{\lfloor\log\delta\rfloor}$ 延迟，并得到相同的渐近界。

4.1 长模式

我们首先说明，当所有模式的长度为 $m>\delta/4$时，如何支持查询。我们对第 3 节中的数据结构稍作修改。现在最小树的大小为 $\delta/2$ ，而不是 1，因此有$\Theta(\log w-\log(\delta/2))=O(\log(w/\delta))$个段和边界树。未覆盖后缀 t 的长度至多为 δ。

我们以与第 3.2 节相同的方式回答查询，只做少量修改。令 P 为长度为 $m>\delta/4$的模式。和之前一样，令 s_i 为满足 $|s_i|\ge m$的最小且最靠右的段。我们像之前一样，利用每个段上的后缀树以及 s_i左侧的边界树，找出段内出现位置或跨越单个边界的出现位置。其余出现位置仍通过从右侧窗口边界开始，逐步构建大小呈指数增长的后缀树来查找。唯一的变化是，现在我们构建树的速度更快，因为还必须覆盖 t；而且，由于模式中有 $m>\delta/4$ 个字符可用于对工作进行去摊销化，因此可以让最小树的大小为 δ 。和上面一样，令 $\mathcal{R}$ 为覆盖 s_i 中最后 $m-1$ 个字符并延伸至右侧窗口边界的字符串，现在它还包含 t。由于 $|t|<\delta$， $\mathcal{R}$ 的长度为 $|\mathcal{R}|<3m+\delta<7m$。假设 $2^\ell\le m<2^{\ell+1}$，我们构建大小为 $7\cdot2^{\ell+1}$ 的后缀树，并在其中匹配 P，将代价摊销到 P的字符上。由于有 $m>\delta/4$个字符可用于对工作进行去摊销化，我们只对满足 $2^{\ell+1}\ge\delta$的每种 ℓ 选择执行此操作，由此在 P中每个字符上产生 $O(\log\ w-\log\ \delta)\ =O(\log(w/\delta))$的工作量。与第 3.2 节一样，我们使用递归范围最大值查询，避免重复报告同时也属于 s的 P 的出现位置。

由于段和边界树也只有 $O(\log(w/\delta))$ 个，我们在 P 中每个字符仅需花费 $O(\log(w/\delta))$ 时间。请注意，我们可以无延迟地回答这些查询。

更新按如下方式执行。每当收到一段包含 $\delta/2$ 个字符的数据时，我们就为其构建后缀树，并在接下来的 $\delta/2$ 个 S 字符中将构建过程去摊销化。我们像之前一样合并后缀树，也在新到达的 S 字符中将合并过程去摊销化。通过修改基础情形，第 3.4 节中的归纳证明仍然成立；合并两棵大小为 $\delta/2$ 的树需要 $\delta/2$ 个字符，而此时会构建另一棵大小为 $\delta/2$ 的树。归纳步骤基于 δ 是 2 的幂这一事实。因此，每次更新的期望时间为 $O(\log(w/\delta))$。

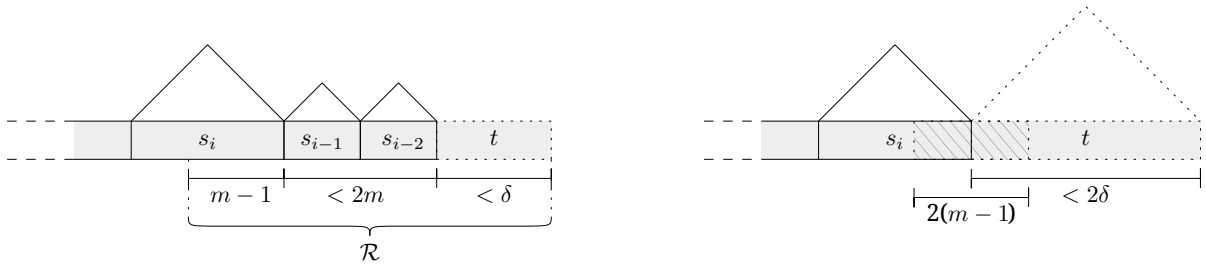


图 4：左：长模式查询示例。这里， s_i 是包含 $|s_i| \geq m$的最小且最靠右的段。注意，非索引后缀 t 的长度小于 $\delta < 4m$ 个字符。右：短模式查询示例。注意，对于短模式， s_i 始终是最靠右的段。 s 中的任何出现位置最多跨越一个边界，并通过构建的树找到。 t 中的任何出现位置都通过我们在刷新缓冲区时构建的、覆盖 t 的后缀树找到。跨越边界（s， t）的任何出现位置，都通过我们基于边界两侧各延伸 $m-1$个字符的子串构建的 KMP 自动机找到；该子串在图中以阴影标出。

4.2 短模式

现在，我们说明如何在所有模式的长度均为 $m \leq \delta$ 时支持查询。我们用一个大小为 δ 的缓冲区扩展该数据结构。此缓冲区将存放尚未回答的查询，以及尚未处理的 S 字符。总空间仍为 $O(w+\delta) = O(w)$。

每当来自 S 的一个字符到达时，我们就将其同时追加到 t 和缓冲区中。当一个模式到达时，我们将整个模式追加到缓冲区，并同时存储当前右侧窗口边界的位置。一旦缓冲区中的字符数（模式和文本合计）超过 $\delta/2$ ，我们就立即分配一个大小为 δ 的新缓冲区，并按如下方式刷新旧缓冲区。注意，此时缓冲区中的字符数严格少于 $\frac{3}{4}\delta$ ，因为每个模式都很短。

刷新缓冲区时，我们首先回答所有缓冲的查询，然后处理所有缓冲的更新。我们将这项工作去摊销化，分散到接下来从任一数据流到达的 $\delta/4$ 个字符上。

回答缓冲的查询时，我们按如下方式进行。设 P_1、.....、 P_ℓ 是缓冲区中的模式，设$m_i = |P_i|$，并设 $M = \sum_{1 \leq i \leq \ell} m_i$。我们有 $M < \delta$。首先，我们在 t上构建一棵后缀树，同时为 t的后缀数组构建一个区间最小值查询数据结构。这需要期望 $O(\delta)$时间。 P_i 的一次出现位置要么完全位于 s中，要么跨越边界(s, t)，要么完全位于 t中。由于 P_i 小于每个段 s_j ，我们可以利用各段上的后缀树以及 O中的边界树，在 $m_i\log(w/\delta)$时间内找出 s 中的所有出现位置。要找出跨越边界的出现位置，我们为 P_i构建 KMP 匹配自动机 [14]。在该自动机中，我们匹配以边界(s, t)为中心、并向两侧各延伸 $m_i - 1$ 个字符的字符串。这需要 $O(m_i)$时间。要找出 t 中的出现位置，我们在 t 上的后缀树中匹配 P_i ，这需要 $O(m_i)$时间。对于所有模式，总计需要$O(M \log(w/\delta)) = O(\delta\log(w/\delta))$时间；去摊销化后，每个字符的期望时间为 $O(\log(w/\delta))$。不过要注意，在 P_i 到达后，来自 S 的更多字符可能已经到达，并被追加到 t中。因此，我们必须注意，不要报告任何延伸到 P_i 到达时的右窗口边界之外的 P_i 的出现位置。KMP 自动机按从左到右的顺序查找出现位置；在 t 中，我们通过递归区间最小值查询来避免报告位置过于靠右的出现位置。

最后，我们按照更新到达的顺序，使用与处理长模式相同的过程来处理缓冲区中的每次更新。每次更新需要 $O(\log(w/\delta))$ 时间，总共需要 $O(\delta\log(w/\delta))$ 时间。因此，由于我们将预期的 $O(\delta\log(w/\delta))$ 工作量分摊到 $\delta/4$ 个字符上，刷新缓冲区时每个字符的预期用时为 $O(\log(w/\delta))$。由于我们一开始刷新就会立即分配一个新缓冲区，因此在下一次刷新开始之前，我们就能完成本次刷新。

4.3 长模式和短模式

现在我们展示如何结合短模式和长模式的解决方案，从而得到一种能够处理任意长度模式的方案。数据结构与上文处理小模式时相同。和上文一样，我们将每个新字符追加到缓冲区中。不过，每当开始流式处理一个模式时，我们也会按 P 是长模式的情况继续处理。如果 P 最终能放入缓冲区而不触发刷新（即使 P 是长模式，也可能出现这种情况），我们只需舍弃为长模式情况所做的工作。不过，如果将 P 添加到

缓冲区后，缓冲区中的字符数超过 $\frac{3}{4}\delta$ ，那么 P 必定是长模式。我们立即开始刷新缓冲区（忽略与 P 相关的字符），并继续将 P 作为长模式处理。请注意，由于我们可能在批处理缓冲区中的更新时，同时流式处理一个长模式，因此在其中进行匹配时，数据结构可能会发生变化。不过，只有在合并完成、用一棵更大的树替换一对后缀树时，数据结构才会变化。如果这种情况发生，我们会将旧树保留在内存中，直到模式处理完毕，然后再将其舍弃。

我们得到如下定理。

定理 3. 设 S 为一个数据流， $w \geq 1$ 和 $\delta \geq 1$ 为整数。我们可以用一个 $O(w)$ 空间数据结构解决 *(w,δ)-SSWSI* 问题，该数据结构支持更新和报告，且每处理一个字符的期望时间为 $O(\log(w/\delta))$ 。此外，每报告一个出现位置，报告操作额外使用最坏情况下的常数时间。

5 获取高概率

本节将说明，如何将时间界改进为 $O(\log(w/\delta))$，且以概率 $1 - w^{-d}$ 成立，其中 $d \geq 1$为任意常数。

第4节时间界中的期望来自后缀树的构造（回想一下，我们也会在查询时构造后缀树）。下面，在引理 2 中，我们证明：给定一个长度为 $k = O(w)$ 的字符串 $\mathcal{K}$ ，可以在 $O(k)$ 时间内以概率 $1 - 1/w^{1+\epsilon}$构造出 $\mathcal{K}$ 上的后缀树，并使用额外的 $O(w/\log w)$ 空间。我们使用该算法在更新和查询期间构造后缀树，像之前一样对其进行去摊还，并对每个到达的字符执行 $O(\log(w/\delta))$ 的工作量。当一个新字符从 S 或 P到达时，至多有 $O(\log(w/\delta))$ 个 $= O(\log w)$ 后缀树构造完成。此时，我们完成那些未能及时完成的树的构造，也就是说，完成所用时间超过分配时间的构造。根据并集界，它们中任何一个未能及时完成（从而产生额外构造开销）的概率至多为 $c\log w/w^{1+\epsilon}$，其中 c为某个常数；当 w足够大时，该概率不超过 $1/w$ 。因此，对于来自 S 或 P 的每个字符，我们以高概率在 w中花费 $O(\log(w/\delta))$ 时间。我们通过概率增强得到 $1 - 1/w^d$ 的概率界：同时运行 $d = O(1)$ 个相互独立的构造算法副本。引理 2 中的算法使用额外的 $O(w/\log w)$ 空间，但我们构造的后缀树数量从不超过 $O(\log w)$，因此总空间使用量为 $O(w)$。

此外，正如第 2 节所述，我们之前使用 FKS 字典 [11] 存储边，以支持对模式中的每个字符进行最坏情况下的常数时间报告查询。该字典的构建时间为期望线性，因此不再适用。我们改用 Dietzfelbinger 和 Meyer auf der Heide 提出的字典 [7]。如果字典中有 n 个元素，它支持最坏情况下的常数时间搜索；对于任意常数 $d' \geq 1$，任意长度为 $\frac{1}{2}n$ 的更新序列中，每次更新所需的时间均为常数，且该性质以概率 $1-1/n^{d'}$ 成立。我们将所有后缀树的所有边都存储在这样一个字典中。我们始终在字典中保留 $\Theta(w)$ 个哑元素，以确保关于 w的概率界良好，并选择足够大的 d' ，使得任意长度为 $O(w)$ 的操作序列（例如构建任意一棵后缀树）都能在 $O(w)$ 时间内完成，且该性质以概率 $1 - 1/w^{d+\epsilon}$成立。

通用哈希在证明引理 2 之前，我们重述一些关于通用哈希的基本事实。通用哈希由 Carter 和 Wegman [6] 提出。令 $M, m > 0$ 为整数， $\mathcal{H}$ 为函数集合 $[0, M] \to [0, m]$，并令 $h \in \mathcal{H}$ 从中均匀随机选取。若 $P[h(x) = h(y) \mid x \neq y] \leq 1/m$，则称 $\mathcal{H}$ 是通用的。令$R \subseteq [0, M]$，且 $|R| = r$。由并集界可知， h 在 R 上发生碰撞的概率至多为

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

特别地，如果 $m = r^c$ 对某个常数 $c \geq 1$ 成立，那么 h 在 R 上是单射（即没有碰撞）的概率至少为 $1 - 1/r^{c-2}$。Carter 和 Wegman 提出了若干类通用哈希函数，我们可以从中以常数时间均匀随机采样一个函数。

快速后缀树构造 现在我们证明引理 2，说明如何以线性时间、高概率地构造我们的后缀树。

引理 2. 给定一个长度为 $k \le 2w$ 的字符串 $\mathcal{K}$，存在一种算法，使用 $O(k + w/\log w)$ 空间，以 $O(k)$ 时间构造 $\mathcal{K}$ 上的后缀树，成功概率为 $1 - 1/w^{1+\epsilon}$，其中 $\epsilon > 0$。

证明。设 $\sigma = \{\mathcal{K}[i] \mid i \in [1,k]\} \subseteq \Sigma$ 为 $\mathcal{K}$的字母表。我们说明如何在 $O(k)$时间内找到一个函数h：$\Sigma \to [1, k^{O(1)}]$，使得 h 在 σ 上是单射，且概率至少为 $1 - 1/w^{1+\epsilon}$。如果 h 在 σ上是单射，我们就可以在时间 $O(\text{sort}(k,\ k^{O(1)})) = O(k)$内，使用基数排序构造 $\mathcal{K}'$ 上的后缀树，其中 $\mathcal{K}'[i] = h(\mathcal{K}[i])$。构造完树后，我们可以在线性时间内将原字母表替换回来。因此，构造算法以至少 $1 - 1/w^{1+\epsilon}$的概率在 $O(k)$时间内完成（否则，我们不保证构造时间，可以用任意方式构造后缀树）。

对于稍后确定的某个 m，令 $f\colon \Sigma \to [1,m]$ 从一类通用哈希函数中均匀随机选取。根据公式 1，f 在 σ 上发生碰撞的概率为

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

我们分为大规模树（$k \ge w^{1/5}$）和小规模树（$k < w^{1/5}$）两种情况。如果 k 较大，则 $w^{1/5} \le k \le 2w$，于是我们设置 $m = w^4$，使 f 发生碰撞的概率至多为

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

对于某个 $\epsilon > 0$。我们通过对集合 $\{(x, f(x))$ 按 $f(\cdot)$ 值排序，并检查是否有两个相邻元素 $(x, f(x))$ 和 $(y, f(y))$ 满足 $x \ne y$ 且 $f(x) = f(y)$，来检验 f 是否为单射。由于 $k \ge w^{1/5}$，使用基数排序可在 $O(\text{sort}(k, w^4)) = O(k)$ 时间内完成。如果 f 是单射，我们就设置 $h = f$，从而完成大规模树情况的证明。

如果 k较小，那么我们可以在常数时间内分配一个长度为 $w/\log w$ 的数组 A。为简单起见，我们假设 A 已初始化，使得对所有 i，$A[i] = 0$。使用标准的常数时间初始化方案可以避免这一假设；假设 A 中的每个条目最初都包含任意值。我们维护另外两个数组 B 和 C，使得如果我们至少向 $A[i]$ 写入过一次值，那么 $A[i]$ 就是指向某个 $B[j]$的指针，$B[j]$ 是指向 $A[i]$的指针，而 $C[j]$ 存储最近一次写入 $A[i]$的值。由此我们可以判断 $A[i]$ 是否已初始化（检查指针是否相同）；如果尚未初始化，就可以在常数时间内将其初始化。

然后我们设置 $m = w/\log w$，使得 f发生碰撞的概率不超过

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

对于 $w \ge 16$。我们通过对 $\mathcal{K}$ 中的每个字符 x，设置 $A[f(x)] = x$，并检查是否有两个不同的字符被哈希到同一索引，来判断 f 在 σ 上是否为单射。如果 f 是单射，那么我们就将值 1，...，$|\sigma|$ 任意分配给 A 中现在非零的索引，并令 $h(x) = A[f(x)]$（此时我们知道 σ，因为它等于 A 中已修改条目的数量）。为了提高成功的概率，我们使用独立选择的 f，最多运行该算法十一次。它们全部失败的概率至多为 $1/w^{11/10} \le 1/w^{1+\epsilon}$，由此完成小规模情况的证明。 □

结合定理2和定理3，即可证明定理 1。

6 结论与未来工作

我们研究了流式滑动窗口字符串索引问题的两种变体：及时变体要求立即回答查询；延迟变体则允许在收到接下来的 δ 个字符后的任意时刻回答查询，其中 δ 是给定参数。对于大小为 w 的滑动窗口，我们给出了一种空间为 $O(w)$ 的数据结构，它以高概率在 $O(\log w)$ 时间内支持更新，并以高概率在模式中的每个字符对应的 $O(\log w)$ 时间内支持查询；报告每个出现位置只需额外的常数时间。对于延迟变体，我们将这些界改进为 $O(\log(w/\delta))$，同时仍能在常数时间内报告每个出现位置。

一个尚未解决的问题是，这些界限能否进一步改进。另一个问题是，当查询可能与对数据流的新更新交错进行时，如何找到高效的解决方案。也就是说，当你正在流式处理一个模式时，S 可能会有新字符到达，从而使当前窗口发生移动。

参考资料

[1] Amihood Amir、Gianni Franceschini、Roberto Grossi、Tsvi Kopelowitz、Moshe Lewenstein 和 Noa Lewenstein。《在比较驱动数据结构中管理无界长度键及其在线索引应用》。《SIAM计算杂志》，43(4):1396–1416，2014年。doi:10.1137/110836377。[2] Amihood Amir、Tsvi Kopelowitz、Moshe Lewenstein 和 Noa Lewenstein。《迈向实时后缀树构造》。载于第12届 SPIRE 会议论文集，第3772卷，第67–78页。施普林格，2005年。doi:10.1007/ 11575832_9.[3] Amihood Amir 和 Igor Nor。《固定有限字母表上的实时索引》。载于第19届 SODA 会议论文集，第1086–1095页，2008年。网址：http://dl.acm.org/citation.cfm?id=1347082.1347201。[4] Dany Breslauer 和 Giuseppe F. Italiano。《通过边缘标记祖先问题实现近实时后缀树构造》。《离散算法杂志》，18:32–48，2013年。doi:10.1016/j.jda.2012.07. 003。[5] Andrej Brodnik 和 Matevz Jekovec。《滑动后缀树》。《算法》，11(8):118，2018年。doi:10. 3390/a11080118。[6] Larry Carter 和 Mark N. Wegman。《通用哈希函数类》。《计算机与系统科学杂志》，18(2):143–154，1979年。doi:10.1016/0022-0000(79)90044-8。[7] Martin Dietzfelbinger 和 Friedhelm Meyer auf der Heide。《一种新的通用哈希函数类及其实时动态哈希》。载于第17届 ICALP 会议论文集，第6–19页，1990年。doi:10.1007/ BFb0032018。[8] Martin Farach-Colton、Paolo Ferragina 和 S. Muthukrishnan。《后缀树构造的排序复杂度》。《ACM杂志》，47(6):987–1011，2000年。doi:10.1145/355541.355547。[9] Edward R. Fiala 和 Daniel H. Greene。《使用有限窗口进行数据压缩》。《ACM通讯》，32(4):490–505，1989年。doi:10.1145/63334.63341。[10] Johannes Fischer 和 Pawel Gawrychowski。《借助 Wexponential 搜索树进行依赖字母表的字符串搜索》。载于第26届 CPM 会议论文集，第160–171页，2005年。doi:10.1007/978-3-319-19929-0_14。[11] Michael L. Fredman、J´anos Koml´os 和 Endre Szemer´edi。《以最坏情况下 O(1) 的访问时间存储稀疏表》。《ACM杂志》，31(3):538–544，1984年。[12] Harold N. Gabow、Jon Louis Bentley 和 Robert Endre Tarjan。《几何问题的缩放及相关技术》。载于第16届 STOC 会议论文集，第135–143页。ACM，1984年。doi:10.1145/800057.808675。[13] Shunsuke Inenaga、Ayumi Shinohara、Masayuki Takeda 和 Setsuo Arikawa。《用于滑动窗口的紧凑有向无环词图》。《离散算法杂志》，2(1):33–51，2004年。doi:10.1016/ S1570-8667(03)00064-9。[14] Donald E. Knuth、James H. Morris Jr. 和 Vaughan R. Pratt。《字符串快速模式匹配》。《SIAM计算杂志》，6(2):323–350，1977年。doi:10.1137/0206024。[15] Tsvi Kopelowitz。《通过对有序列表子集进行前驱查询，实现通用字母表的在线索引》。载于第53届 FOCS 会议论文集，第283–292页，2012年。doi:10.1109/FOCS.2012.79。[16] S. Rao Kosaraju。《实时模式匹配与准实时后缀树构造（初步版本）》。载于第26届 STOC 会议论文集，第310–316页，1994年。doi:10.1145/195058.195170。[17] Gregory Kucherov 和 Yakov Nekrich。《面向常数大小字母表的完整实时索引》。《算法学》，79(2):387–400，2017年。doi:10.1007/s00453-016-0199-7。[18] N. Jesper Larsson。《字符串匹配与数据压缩的结构》。博士论文，瑞典隆德大学，1999年。网址：http://lup.lub.lu.se/record/19255。

[19] Joong Chae Na、Alberto Apostolico、Costas S. Iliopoulos 和 Kunsoo Park。《截断后缀树及其在数据压缩中的应用》。理论计算机科学，304(1-3):87–101，2003年。doi:10.1016/S0304-3975(03)00053-7。[20] Patrick E. O'Neil、Edward Cheng、Dieter Gawlick 和 Elizabeth J. O'Neil。《日志结构合并树（LSM-Tree）》。*Acta Informatica*，33(4):351–385，1996年。doi:10.1007/s002360050048。[21] M Senft。《滑动窗口的后缀树：概述》。载于*WDS* 论文集，第 5 卷，第 41–46 页，2005年。[22] Martin Senft 和 Tom´as Dvor´ak。《滑动 CDAWG 完美化》。载于第 *15* 届 *SPIRE* 论文集，第 109–120 页，2008年。doi:10.1007/978-3-540-89097-3_12。[23] Peter Weiner。《线性模式匹配算法》。载于第 *14* 届 *SWAT* 论文集，第 1–11 页，1973年。doi:10.1109/SWAT.1973.13。