LSHAlign: All-Pair Near-Duplicate Text Alignment via LSH

YUHENG ZHANG*, Rutgers University, United States
ZHENCAN PENG*, Rutgers University, United States
MIAO QIAO, University of Auckland, New Zealand
WEI ZHANG, Alibaba Cloud Computing, China
FEIFEI LI, Alibaba Cloud Computing, China
DONG DENG†, Rutgers University, United States

The all-pair near-duplicate text alignment problem aims to identify all pairs of similar subsequences (i.e., contiguous token spans) between two long texts. This problem is central to many applications, including large-scale text deduplication, plagiarism detection, and bioinformatics. Given two input texts T and S, each represented as a sequence of tokens, the naive approach that compares every pair of subsequences between T and S is computationally infeasible, requiring $O(|\mathrm{T}|^2|\mathrm{S}|^2k)$ time even when estimating Jaccard similarity via min-hash with k independent hash functions. To overcome this limitation, we propose an efficient framework based on Locality-Sensitive Hashing (LSH), which employs $m \times L$ independent hash functions organized into L hash tables of m hash functions each. Our key observation is that many subsequences in a text share identical LSH values, allowing us to group them by their LSH values. We prove that the expected number of such groups for a text of length n is $O(nmL)$. We further design a data structure to represent each group in $O(1)$ space. As a result, we develop an algorithm that reduces the expected time and space complexities of all-pair near-duplicate text alignment to $O((|\mathrm{T}| + |\mathrm{S}|)mL)$, excluding the cost of outputting the pairs. This is significantly lower than the naive approach using the same number $k = m \times L$ of hash functions. Experiments on large real-world datasets show an order of magnitude speedups over state-of-the-art baselines while maintaining comparable alignment accuracy.

CCS Concepts: • **Information systems → Near-duplicate and plagiarism detection**; **Structured text search**.

Additional Key Words and Phrases: Text Alignment, Jaccard Similarity, Locality-Sensitive Hashing

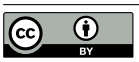
ACM Reference Format:
Yuheng Zhang, Zhencan Peng, Miao Qiao, Wei Zhang, Feifei Li, and Dong Deng. 2026. LSHAlign: All-Pair Near-Duplicate Text Alignment via LSH. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 206 (June 2026), 26 pages. https://doi.org/10.1145/3802083

1 Introduction

In this paper, we study the all-pair near-duplicate text alignment problem, which aims to identify all pairs of similar subsequences between two long texts. This problem is fundamental to a wide range of applications, including large-scale text deduplication [9, 28], test set leakage detection in large

*These authors contributed equally to this work.
†Corresponding author.

Authors' Contact Information: Yuheng Zhang, Rutgers University, New Brunswick, NJ, United States, yuheng.zhang@cs.rutgers.edu; Zhencan Peng, Rutgers University, New Brunswick, NJ, United States, zhencan.peng@cs.rutgers.edu; Miao Qiao, University of Auckland, Auckland, New Zealand, miao.qiao@auckland.ac.nz; Wei Zhang, Alibaba Cloud Computing, Hangzhou, Zhejiang, China, zwei@alibaba-inc.com; Feifei Li, Alibaba Cloud Computing, Hangzhou, Zhejiang, China, lifeifei@alibaba-inc.com; Dong Deng, Rutgers University, New Brunswick, NJ, United States, dong.deng@cs.rutgers.edu.



ACM 2836-6573/2026/6-ART206
https://doi.org/10.1145/3802083

language models (LLMs) [50], machine-generated log management [15], plagiarism detection [44], copyright protection [37], and biological sequence analysis [1].

As a motivating example, given a source document and a suspicious document, all-pair near-duplicate text alignment can reveal all potential cases of plagiarism between them [33, 34]. Similarly, in bioinformatics, given two molecular sequences, all-pair near-duplicate text alignment can identify all potential local alignments [43] between them. Finally, one can concatenate all test set examples into a single text and all training examples into another, and then apply all-pair near-duplicate text alignment to detect test set leakage [50].

A key challenge in all-pair near-duplicate text alignment lies in the combinatorial explosion of subsequence pairs. In this paper, each text is represented as a sequence of tokens (e.g., words, q-grams, or byte-pair encoding (BPE) units [20]). A subsequence refers to a consecutive span of tokens within a text. Given two texts T and S containing |T| and |S| tokens, respectively, there are $O(|\mathrm{T}|^2|\mathrm{S}|^2)$ subsequence pairs. We measure the similarity between two subsequences using the Jaccard similarity, which can be efficiently estimated via min-hash [7]. The following pseudo-code shows the "nested loop join" style end-to-end workflow:

```
for each subsequence T[i,j] in T:
    for each subsequence S[p,q] in S:
        if estimated_Jaccard(T[i,j], S[p,q]) > threshold:
            include_candidate (T[i,j], S[p,q])
```

With appropriate preprocessing [35], the Jaccard similarity between two subsequences can be estimated in $O(k)$ time using k independent hash functions, where k is a tunable parameter that trades off estimation accuracy against computational cost. Overall, this naive approach requires $O(|\mathrm{T}|^2|\mathrm{S}|^2k)$ time, which is prohibitively expensive for long texts containing millions of tokens.

LSH Framework. To address this issue, we consider a framework based on Locality-Sensitive Hashing (LSH) [26]. The following "hash-join" style pseudocode shows the overall workflow:

```
for each T[i,j] in T:
    for x in [1,L]:
        add T[i,j] to bucket no. LSH(x, T[i,j])
for each S[p,q] in S:
    for x in [1,L]:
        for each T[i,j] in bucket no. LSH(x, S[p,q]):
            include_candidate (T[i,j], S[p,q])
```

For each subsequence in T, the framework computes L hash values, each generated from a combination of m hash functions. This requires $m \times L$ hash functions in total. The parameters m and L are tunable with respect to the target similarity threshold, enabling a trade-off between hashing cost and accuracy. Each subsequence is then inserted into L corresponding hash buckets based on these values. By design, the hash functions ensure that similar subsequences (with high Jaccard similarity) are likely to fall into the same bucket, while dissimilar ones (with low Jaccard similarity) rarely collide.

The framework then performs hash table probing using the same process. Specifically, for each subsequence in S (referred to as the probing subsequence), it applies the same set of hash functions to compute L hash values and probes the corresponding L buckets. All subsequences retrieved from these buckets are considered similar to the probing subsequence. Finally, candidate pairs are generated by pairing each retrieved subsequence with the probing subsequence.

With appropriate preprocessing [35], each hash value can be computed in $O(m)$ time. Overall, if excluding the cost of generating candidate pairs, the time complexity of the framework is

$O((|\mathrm{T}|^2 + |\mathrm{S}|^2)mL)$, which is substantially lower than the naive approach when using the same number of hash functions $k = m \times L$. However, the framework still exhibits quadratic time complexity with respect to the text length, making it impractical for long texts.

In this paper, we dramatically reduce the expected cost of hash table construction and probing for a pair of texts T and S to $O((|\mathrm{T}| + |\mathrm{S}|)mL)$. Specifically, we observe that many subsequences in a text can share the same hash value under LSH. Thus, we propose to group the subsequences by their LSH values. For a text of length n, we prove that the number of such groups is $O(nmL)$ in expectation. Furthermore, we design a data structure (namely "tight intervals") to represent each group of subsequences using $O(1)$ space. We then develop an algorithm to produce these groups in $O(nmL)$ time in expectation. Consequently, our algorithm achieves $O((|\mathrm{T}| + |\mathrm{S}|)mL)$ time and space complexities for hash table construction and probing for two texts T and S, which substantially improves the quadratic cost in the LSH framework. Extensive experiments on large real-world datasets demonstrate that our approach outperforms existing baselines by an order of magnitude. In summary, this paper makes the following contributions:

(1) We propose an efficient LSH-based framework to address the all-pair near-duplicate text alignment problem.
(2) We show that in expectation a text with n tokens leads to $O(nmL)$ groups of subsequences sharing the same hash value under LSH.
(3) We develop an algorithm to generate all the groups in a text with n tokens using $O(nmL)$ time and space in expectation.
(4) We conduct extensive experiments on large real-world datasets. Experiment results show that our approach outperforms existing baselines by an order of magnitude.

2 Preliminaries

2.1 Problem Definition

We begin by introducing the key notation used throughout the paper. A text T is defined as a sequence of tokens, where $\mathrm{T}[i]$ denotes the i-th token in the sequence. The length of the text, i.e., the total number of tokens, is denoted by $|\mathrm{T}|$. Given $1 \le i \le j \le |\mathrm{T}|$, $\mathrm{T}[i, j]$ denotes the subsequence of T starting at position i and ending at position j, inclusive. A text (or a subsequence) can be mapped to a set, consisting of its distinct tokens. When the context is clear, we refer to T (or $\mathrm{T}[i, j]$) both as a text (or subsequence) and as a set.

Given two texts T and S, their *Jaccard similarity* is defined as

$$\mathrm{J}_{\mathrm{T,S}} = \frac{|\mathrm{T} \cap \mathrm{S}|}{|\mathrm{T} \cup \mathrm{S}|}.$$

Example 1. *Consider two texts* $\mathrm{T} = ABBC$ *and* $\mathrm{S} = BCD$ *where each letter is a token. The union* $\mathrm{T} \cup \mathrm{S} = \{A, B, C, D\}$*. The intersection* $\mathrm{T} \cap \mathrm{S} = \{B, C\}$*. Therefore, their Jaccard similarity is* $\mathrm{J}_{\mathrm{T,S}} = \frac{1}{2}$.

Definition 1 (All-pair near-duplicate text alignment). *Given two texts* T *and* S*, and a similarity threshold* $\theta \in [0, 1]$*, the problem of all-pair near-duplicate text alignment aims at returning all the subsequence pairs* $(\mathrm{T}[a, b], \mathrm{S}[c, d])$*, where* $1 \le a \le b \le |\mathrm{T}|$ *and* $1 \le c \le d \le |\mathrm{S}|$*, such that* $\mathrm{J}_{\mathrm{T}[a,b],\mathrm{S}[c,d]} \ge \theta$.

Example 2. *Consider two texts* $\mathrm{T} = ABBC$ *and* $\mathrm{S} = BCD$*, and the threshold* $\theta = 0.6$*. All-pair near-duplicate text alignment returns* $(\mathrm{T}[1, 4], \mathrm{S}[1, 2])$*,* $(\mathrm{T}[2, 2], \mathrm{S}[1, 1])$*,* $(\mathrm{T}[2, 3], \mathrm{S}[1, 1])$*,* $(\mathrm{T}[2, 4], \mathrm{S}[1, 2])$*,* $(\mathrm{T}[2, 4], \mathrm{S}[1, 3])$*,* $(\mathrm{T}[3, 3], \mathrm{S}[1, 1])$*,* $(\mathrm{T}[3, 4], \mathrm{S}[1, 2])$*,* $(\mathrm{T}[3, 4], \mathrm{S}[1, 3])$*, and* $(\mathrm{T}[4, 4], \mathrm{S}[2, 2])$ *as the results.*

Our goal is to design an efficient approximate algorithm for the all-pair near-duplicate text alignment problem.

2.2 Jaccard Similarity Estimation via Min-Hash

The Jaccard similarity of two texts can be estimated by their min-hash sketches.

Hash Family $\mathcal{H}$. Denote by $\mathcal{T}$ the universe of tokens. $\mathcal{H}$ denotes a universal hash family consisting of hash functions $h : \mathcal{T} \mapsto \mathbb{R}$ that map each token to a real value. Consider a universal hash function h randomly selected from hash family $\mathcal{H}$. For a text T, we abuse $h(\mathrm{T})$ to denote the min-hash of T under hash function h:

$$h(\mathrm{T}) = \min\{h(t) \mid t \in \mathrm{T}\}.$$

A commonly used universal hash function family $\mathcal{H}$ is $h(x) = (ax + b) \mod p \mod m$ where p is a large prime, $m < p$, and a, b are two randomly chosen integers modulo p with $a \neq 0$ [12]. It has been shown [7] that the min-hash collision probability of T and S is their Jaccard similarity

$$\Pr[h(\mathrm{T}) = h(\mathrm{S})] = \mathrm{J}_{\mathrm{T,S}}.$$

MinHash-Similar. Consider k independent universal hash functions $h_1, h_2, \cdots, h_k$ from universal hash family $\mathcal{H}$. For a text T, we call $(h_1(\mathrm{T}), \ldots, h_k(\mathrm{T}))$ the min-hash sketch of T under the k hash functions. Given two texts T and S, their Jaccard similarity can be estimated with $\hat{\mathrm{J}}_{\mathrm{T,S}} = \frac{1}{k}\sum_{i=1}^{k} \mathbf{1}\{h_i(\mathrm{T}) = h_i(\mathrm{S})\}$. This is an unbiased estimator of Jaccard similarity with variance $\mathrm{J}_{\mathrm{T,S}}(1 - \mathrm{J}_{\mathrm{T,S}})/k$ and can be computed in $O(k)$ time if the min-hash sketches are provided. T and S are called MinHash-Similar if their estimated Jaccard similarity is no smaller than a user-provided threshold. As discussed in Section 1, to reduce the cost of all-pair near-duplicate text alignment, we can use Locality-Sensitive Hashing (LSH) [26].

2.3 Locality-Sensitive Hashing (LSH) Framework for Jaccard Similarity

Hash Family $\mathcal{G}$. We define a hash family $\mathcal{G}$, where each hash function $g \in \mathcal{G}$ consists of m independent hash functions $h_1, h_2, \ldots, h_m$ randomly drawn from the universal hash family $\mathcal{H}$ defined in Section 2.2. For any $g \in \mathcal{G}$, the hash value of a text T is defined as $g(\mathrm{T}) = (h_1(\mathrm{T}), h_2(\mathrm{T}), \ldots, h_m(\mathrm{T}))$. For two texts T and S, define $g(\mathrm{T}) = g(\mathrm{S})$ only if $h_i(\mathrm{T}) = h_i(\mathrm{S})$ for all $i = 1, 2, \ldots, m$.

Hash Family $\mathcal{Y}$. We then define the hash family $\mathcal{Y}$ where each hash function $y \in \mathcal{Y}$ consists of L independent hash functions $g_1, g_2, \ldots, g_L$ randomly drawn from the universal hash family $\mathcal{G}$. For any $y \in \mathcal{Y}$, the hash value of a text T is defined as $y(\mathrm{T}) = (g_1(\mathrm{T}), \ldots, g_L(\mathrm{T}))$. For two texts T and S, define $y(\mathrm{T}) = y(\mathrm{S})$ if there exists $i \in [1, L]$ such that $g_i(\mathrm{T}) = g_i(\mathrm{S})$.

LSH-Similar. Given a hash function $y \in \mathcal{Y}$, we define two texts T and S as LSH-Similar if $y(\mathrm{T}) = y(\mathrm{S})$.

Under an LSH family with parameters (m, L), for two texts T and S with Jaccard similarity p, the probability that they are LSH-similar is $1 - (1 - p^m)^L$; the probability that they are not LSH-similar is $(1 - p^m)^L$.

Example 3. *Consider $m = 5$ and $L = 20$. When the Jaccard similarity is $p = 0.3$, the probability that the pair is not LSH-similar is 0.9525 and that they are LSH-similar is 0.0475; when $p = 0.8$, the probability that the pair is not LSH-similar is 0.00035 and that they are LSH-similar is 0.99965.*

A higher Jaccard similarity leads to a very small probability of not being LSH-similar, while a lower Jaccard similarity leads to a very small probability of being LSH-similar.

By choosing appropriate parameters m and L, LSH serves as a probabilistic indicator for whether the Jaccard similarity is at least a threshold θ. Figure 1 shows the probability curve of being LSH-similar with parameters $m = 8$ and $L = 16$. It provides a sharp probabilistic separation between dissimilar and similar pairs.

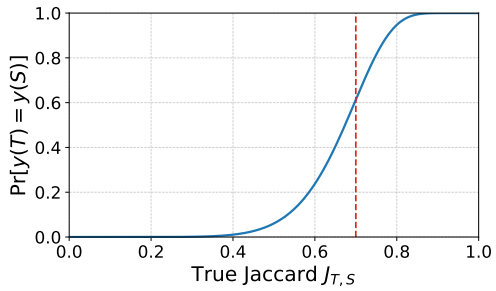


Fig. 1. The probability y that two texts with true Jaccard similarity x are reported as LSH-Similar under $m = 8$ and $L = 16$. The dashed line shows a similarity threshold $\theta = 0.7$.

As discussed in Section 1, compared to the "nested loop join" style framework which uses min-hash sketches to estimate Jaccard similarity, reporting LSH-similar text pairs offers a significant efficiency gain; however, its drawback is that the time complexity is still quadratic in |T| and |S|, which is not acceptable for long texts. Next, we propose an efficient framework to break the quadratic barrier.

3 The Framework

This section serves as an overview of our framework and a catalog of our key ideas, definitions, main algorithms, and analysis.

Given a hash function $y = (g_1, g_2, \ldots, g_L) \in \mathcal{Y}$, to find all the subsequence pairs in two texts T and S that are LSH-similar, for each hash function g_x where $x \in [1, L]$, we propose to group all subsequences $\mathrm{T}[i, j]$ of T according to their hash values $g_x(\mathrm{T}[i, j])$, and likewise, group all subsequences $\mathrm{S}[p, q]$ of S by $g_x(\mathrm{S}[p, q])$. For each pair of groups, one from T and one from S, that share the same hash value, we report all subsequence pairs, one subsequence from each group in the pair, as results.

One key observation is that neighboring subsequences of T often share the same hash value under a hash function $g \in \mathcal{G}$ and can thus be compactly represented by a construct called a rectangle.

Definition 2 (Rectangle). *Given a text* T *of length n and a hash function* $g \in \mathcal{G}$*. A rectangle in* T *under a hash function g is a set of subsequences that share the same hash value under g, represented by the tuple* $\mathrm{rec}(a, b, c, d, g) = \{\mathrm{T}[i, j] \mid i \in [a, b], j \in [c, d]\}$*, where* $1 \le a \le b \le c \le d \le n$.

A rectangle is an $O(1)$-space "compression" of neighboring subsequences sharing the same hash value under g. A natural follow-up question is that

> Can we group the subsequences by their hash values, i.e., can we use rectangles to canonically partition all the subsequences based on their hash values?

The answer is *positive*. Note that, duplication or hash collisions may introduce ties. We break ties and then assume that the universal hash function $h \in \mathcal{H}$ is collision-free, as described below.

Tie-breaking and Collision-free assumption. Given text T, for each $h \in \mathcal{H}$, we assume that $h(\mathrm{T}[p]) \neq h(\mathrm{T}[q])$ for all distinct positions $p, q \in [1, n = |\mathrm{T}|]$. In case duplicate tokens occur, ties are broken uniformly at random. This can be achieved by assigning each position p an independent random priority $\pi(p)$. Specifically, for any two distinct positions p and q in a text T such that $\mathrm{T}[p] = \mathrm{T}[q]$, we define $h(\mathrm{T}[p]) < h(\mathrm{T}[q])$ if $\pi(p) < \pi(q)$, and $(h(\mathrm{T}[p]) > h(\mathrm{T}[q])$ if otherwise. For simplicity in discussion and notation, we assume this holds throughout our analysis.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$h_1(T[i])$	82	59	22	57	39	90	94	42	32	64	91	48	99	73	53
$h_2(T[i])$	93	12	19	46	4	72	47	12	89	33	5	71	26	94	58

Fig. 2. The token hash values of a text T with $|\mathrm{T}| = 15$ under two hash functions h_1 and h_2 in $\mathcal{H}$.

This tie-breaking does not modify hash values and therefore has no effect on the behavior of LSH. Our correctness of algorithm and all theoretical guarantees do *not* require this assumption, because our analysis does not depend on which position is chosen when multiple positions share the same hash value, any tie-breaking rule yields a valid partition.

One hash function. We first consider the rectangles when g consists of only one hash function $h \in \mathcal{H}$. We observe that all subsequences of a text T with the same hash value can be represented by a single rectangle, which is uniquely determined by a kind of intervals $[l, r]$ called *tight intervals* (as formally defined below). Intuitively, a tight interval is "tight" because it cannot be extended without changing its min-hash value.

Definition 3 (Tight Interval). *Given a hash function $h \in \mathcal{H}$ and a text* T *of length n, we call an interval $[l, r]$*

- left-tight under h *if and only if (iff)* $l = 1$ *or* $h(\mathrm{T}[l-1]) < h(\mathrm{T}[i])$ *for every* $i \in [l, r]$,
- right-tight under h *iff* $r = n$ *or* $h(\mathrm{T}[r+1]) < h(\mathrm{T}[i])$ *for every* $i \in [l, r]$, *and*
- tight under h *iff it is both left-tight and right-tight under h.*

Example 4. *Consider the text* T *and hash function h_2 in Figure 2. The interval $[3, 4]$ is tight under h_2. This is because it is left-tight under h_2 as $h_2(\mathrm{T}[2]) = 12 < \min(h_2(\mathrm{T}[3]), h_2(\mathrm{T}[4]))$ and right-tight under h_2 as $h_2(\mathrm{T}[5]) = 4 < \min(h_2(\mathrm{T}[3]), h_2(\mathrm{T}[4]))$. Moreover, $\mathrm{rec}(3, 3, 3, 4, (h_2)) = \{\mathrm{T}[3, 3], \mathrm{T}[3, 4]\}$ is a rectangle.*

As we will show later, there is a one-to-one mapping between all the tight intervals in a text T and all groups of subsequences in T sharing the same hash values. Specifically, each tight interval $[l, r]$ corresponds to a group of subsequences sharing the same hash value, which can be represented by a rectangle in the form of $(l, *, *, r, g)$. The two unknown values $*$ are determined by other tight intervals. Therefore, we wish to generate all the rectangles based on all the tight intervals of T. To find all the tight intervals efficiently, we propose Algorithm 1, which takes $O(|\mathrm{T}|)$ time and space (Theorem 2). This also proves that the total number of tight intervals is $O(|\mathrm{T}|)$.

Multiple hash functions. The tight intervals discussed above apply to the case where g consists of a single hash function, i.e., $g = (h)$. We now extend the definition of tight intervals to the general case where g comprises multiple hash functions. Here, we say an interval is tight under g if it is tight under at least one hash function in g.

Definition 4. *Given a hash function $g = (h_1, h_2, \ldots, h_m) \in \mathcal{G}$ and a text* T*, an interval $[l, r]$ is*

- left-tight under g *iff there exists* $i \in [1, m]$ *such that $[l, r]$ is left-tight under h_i,*
- right-tight under g *iff there exists* $i \in [1, m]$ *such that $[l, r]$ is right-tight under h_i,*
- tight under g *iff it is both left-tight and right-tight under g.*

Example 5. *Consider text* T *and hash function $g = (h_1, h_2)$ in Figure 2. Interval $[1, 2]$ is tight under g: it is right-tight under g since $h_1(\mathrm{T}[3]) = 22 < \min(h_1(\mathrm{T}[1]), h_1(\mathrm{T}[2]))$; and left-tight under g since $l = 1$ (which indicates $[1, 2]$ is left-tight under both h_1 and h_2).*

Similarly, there is a one-to-one mapping between all tight intervals in a text T under any hash function $g \in \mathcal{G}$ and all groups of subsequences in T sharing the same hash values under g. Our Theorem 4 shows that the expected total number of tight intervals of g with m hash functions is bounded by $O(m|\mathrm{T}|)$. To find all the tight intervals efficiently, we propose Algorithm 2 which in expectation takes $O(m|\mathrm{T}|)$ time and space to produce all the tight intervals (Theorem 5).

We generate one rectangle for each tight interval – denote the set of all such rectangles by P – and the hash value of $\mathrm{T}[l, r]$ for each tight interval $[l, r]$ by Algorithm 3. Its time and space complexities are asymptotically the same as Algorithm 2, as proved in Theorem 7.

With the rectangle $(a, b, c, d, g) \in \mathrm{P}$ and the hash value v for each tight interval $[a, d]$, our end-to-end framework is outlined below.

```
for each x in [1,L]:
    use tight intervals to produce, for T, S under g_x, respectively, rectangles
    ↪ with hash values
    for each (a,b,c,d,g_x) of T with hash value v:
        add (a,b,c,d,g_x) to bucket no. v
    for each (u,w,y,z,g_x) of S with hash value v:
        for each (a,b,c,d,g_x) in bucket no. v:
            for each i in [a,b] and j in [c,d]:
                for each p in [u,w] and q in [y,z]:
                    include_candidate (T[i,j], S[p,q])
```

The detailed framework is shown in Algorithm 4 in Section 4.

To this end, we can answer the question raised at the beginning of this section: P *is* a partition of all the subsequences of T based on their hash values, as proved by Theorem 6 in Section 5. This ensures the correctness of our framework.

4 Tight Interval Generation

In this section, we discuss how to generate all the tight intervals under a hash function $g \in \mathcal{G}$. For ease of presentation, hereinafter, for any text T of length n, we add two dummy tokens $\mathrm{T}[0]$ and $\mathrm{T}[n+1]$, where $h(\mathrm{T}[0]) = h(\mathrm{T}[n+1]) = -\infty$. All positions we mention are in the range of $[0, n+1]$, while every interval $[l, r]$ satisfies $1 \le l \le r \le n$. Thus an interval $[l, r]$ is left-tight under h if and only if $h(\mathrm{T}[l-1]) < h(\mathrm{T}[l, r])$, whereas it is right-tight under h if and only if $h(\mathrm{T}[r+1]) < h(\mathrm{T}[l, r])$.

4.1 Single Hash Function

This section is a building block of our partition algorithm. Unless specified otherwise, tightness is defined under the hash function $h \in \mathcal{H}$. We observe that for an interval $[l, r]$ to be left-tight under a hash function $h \in \mathcal{H}$, the position $l-1$ plays a critical role: its hash value $h(\mathrm{T}[l-1])$ must be smaller than each of $h(\mathrm{T}[l]), h(\mathrm{T}[l+1]), \ldots, h(\mathrm{T}[r])$. Next we formalize this idea.

Definition 5 (Dominance). *A position p dominates another position q if and only if $p > q$ and $h(\mathrm{T}[p]) < h(\mathrm{T}[q])$.*

Note that by definition, position $n+1$ dominates all positions $1, 2, \ldots, n$, but not position 0. Actually, no position dominates position 0.

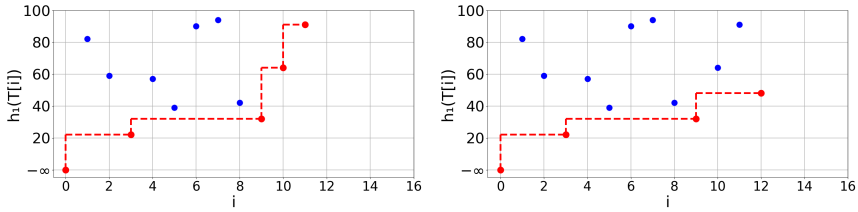
Example 6. *Figure 3 shows the token hash values of a text* T *under a hash function h_1. Position* 10 *is dominated by positions* 12, 15 *and* 16.

To characterize the left-tight intervals, we define the skyline.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$h_1(T[i])$	$-\infty$	82	59	22	57	39	90	94	42	32	64	91	48	99	73	53	$-\infty$

Fig. 3. Hash values of a text T under a hash function h_1.

Definition 6 (Skyline). *The skyline* $\mathrm{S}(r)$ *at position* r *is the set of positions in* $[0, r]$ *that are not dominated by any other position in* $[0, r]$*, i.e.,* $\mathrm{S}(r) = \{i \in [0, r] \mid \nexists\, j \in [0, r],\ j \text{ dominates } i\}$.



(a) skyline at position $r = 11$ (b) skyline at position $r = 12$

Fig. 4. Skyline derivation. The points on the skyline are highlighted in red, and the skyline itself is drawn using red dashed lines. The absence of any point below the staircase indicates that no skyline point is dominated.

Example 7. *Figure 4 illustrates two example skylines. The x-axis represents token positions, while the y-axis shows the corresponding token hash values of the text* T *under the hash function* h_1*, as defined in Figure 3. For instance, as shown in Figure 4(a), the skyline at* $r = 11$ *is* $\mathrm{S}(11) = (0, 3, 9, 10, 11)$*, whereas Figure 4(b) shows that the skyline at* $r = 12$ *becomes* $\mathrm{S}(12) = (0, 3, 9, 12)$.

Intuitively, a position p in the skyline at position r means no hash value between $p + 1$ and r is smaller than that of p. Thus $[p + 1, r]$ must be left-tight by Definition 3, as formalized below.

Lemma 1. *Let* S *be the skyline at position* r*. An interval* $[l, r]$ *is left-tight if and only if* $l - 1 \in \mathrm{S} \setminus \{r\}$.

Proof. We first prove for any left-tight interval $[l, r]$, $l - 1 \in \mathrm{S} \setminus \{r\}$. Since $[l, r]$ is an interval, $l \le r$. Thus $l - 1 \neq r$ and $l - 1 \notin \{r\}$. On the other hand, by the definition of left-tightness, $h(\mathrm{T}[l - 1]) < h(\mathrm{T}[x])$ for every $x \in [l, r]$. Therefore, by the definition of dominance, no position $x \in [l, r]$ dominates $l - 1$. Thus, by the definition of skyline, $l - 1 \in \mathrm{S}$. As a result, we have $l - 1 \in \mathrm{S} \setminus \{r\}$.

Next, we prove for every position $p \in \mathrm{S} \setminus \{r\}$, the interval $[p + 1, r]$ must be left-tight. First of all, by the definition of skyline, for every position $p \in \mathrm{S}$, $p \le r$. Thus $p + 1 \le r$ for every $p \in \mathrm{S} \setminus \{r\}$ and $[p + 1, r]$ must be a non-empty interval. Moreover, by the definition of skyline, no position in $[p + 1, r]$ dominates p. Thus $h(\mathrm{T}[p]) < h(\mathrm{T}[x])$ for every $x \in [p + 1, r]$. Therefore $[p + 1, r]$ must be left-tight by definition. □

Next we characterize the right-tight intervals. An interval $[l, r]$ is right-tight if and only if $h(\mathrm{T}[r + 1])$ is smaller than the smallest hash value of the positions in $[l, r]$. Consider the smallest position $p \in \mathrm{S}$ such that $p > l - 1$, where S is the skyline at position r. We observe that the hash value of p is the smallest among $[l, r]$. This is because, on the one hand, by the definition of skyline, $h(\mathrm{T}[p]) < h(\mathrm{T}[x])$ for every $x \in [p + 1, r]$. On the other hand, we observe that $h(\mathrm{T}[p]) < h(\mathrm{T}[x])$ for every $x \in [l, p - 1]$. This is because p dominates every $x \in [l, p - 1]$ by Lemma 2.

Lemma 2. *Let* S *be the skyline at position* r*, for any position* $q \in \mathrm{S} \setminus \{0\}$*, let* $p = \max\{p \in \mathrm{S} \mid p < q\}$*. For any position* x *such that* $p < x < q$*,* q *dominates* x*.*

Proof. Consider a position x such that $p < x < q$. Let y be the position in $[x+1, r]$ with the smallest hash value, i.e., $y = \arg\min_{z \in [x+1,r]} h(\mathrm{T}[z])$. Since $x \notin \mathrm{S}$, by Definition 6 there is some position in $[x+1, r]$ that dominates x, thus $h(\mathrm{T}[y]) < h(\mathrm{T}[x])$, i.e., y dominates x. Consider two cases. **Case 1:** $y > q$. Because $q \in \mathrm{S}$, no position $y > q$ can dominate q, so $h(\mathrm{T}[y]) > h(\mathrm{T}[q])$. Thus, $h(\mathrm{T}[x]) > h(\mathrm{T}[y]) > h(\mathrm{T}[q])$, hence q dominates x. **Case 2:** $x < y \leq q$. As y is the position in $[x+1, r]$ with the smallest hash value, $h(\mathrm{T}[y]) \leq h(\mathrm{T}[q])$. Thus, q cannot dominate y. Because all positions in $[q+1, r]$ have hash values no less than $h(\mathrm{T}[q])$ — otherwise q will not be in the skyline S, thus no position in the skyline S can dominate y. Thus $y \in \mathrm{S}$. Since there is no position in $[p+1, q-1]$ in the skyline, $y = q$. Thus q dominates x. □

Thus we have the following sufficient and necessary condition for an interval to be right-tight.

Lemma 3. *Given an interval* $[l, r]$*. Let* S *be the skyline at position* r*, i.e., all positions in* S *are no more than* r*. The interval is right-tight if and only if* $h(\mathrm{T}[r+1]) < h(\mathrm{T}[p])$ *where* $p = \min\{p \in \mathrm{S} \mid p \geq l\}$*.*

Proof. When $[l, r]$ is right-tight, $h(\mathrm{T}[r+1]) < h(\mathrm{T}[x])$ for every $x \in [l, r]$. As $p \in [l, r]$, $h(\mathrm{T}[r+1]) < h(\mathrm{T}[p])$.

Next we prove that if $h(\mathrm{T}[r+1]) < h(\mathrm{T}[p])$ then $[l, r]$ is right-tight. Since for any position $x \in [p+1, r]$, $h(\mathrm{T}[x]) > h(\mathrm{T}[p])$ because otherwise, p would not be in the skyline which contradicts that $p \in \mathrm{S}$. Based on Lemma 2, for any $x \in [l, p-1]$, x is dominated by p, i.e., $h(\mathrm{T}[x]) > h(\mathrm{T}[p])$. Therefore, $h(\mathrm{T}[r+1]) < h(\mathrm{T}[p])$ means $h(\mathrm{T}[r+1])$ is smaller than all the hash values of positions in $[l, r]$ and thus $[l, r]$ is right-tight. □

Given a position r, let S be the skyline at position r, we can visit every position $p \in \mathrm{S}$. The interval $[p+1, r]$ must be left-tight by Lemma 1. Then we can check if it is right-tight by Lemma 3. However, it is unnecessary to check all positions in the skyline.

Lemma 4. *Let* S *be the skyline at position* r*. If* $h(\mathrm{T}[r+1]) > h(\mathrm{T}[p])$ *for a position* $p \in \mathrm{S}$*, then any interval* $[l, r]$ *where* $l \leq p$ *must not be right-tight.*

Proof. By definition, $[l, r]$ is right-tight only if $h(\mathrm{T}[r+1]) < h(\mathrm{T}[x])$ for every $x \in [l, r]$. However, we have $p \in [l, r]$ and $h(\mathrm{T}[r+1]) > h(\mathrm{T}[p])$. Thus $[l, r]$ must not be right-tight. □

Example 8. *As shown in Figures 3 and 4(a), consider the skyline* $\mathrm{S}(r) = (0, 3, 9, 10, 11)$ *at position* $r = 11$*. Note that* $h(\mathrm{T}[r+1]) = h(\mathrm{T}[12]) = 48$*, and the largest position in* S *with a hash value smaller than* 48 *is* 9*. Therefore, the interval* $[10, 11]$ *is right-tight. Moreover, for any* $l \leq 9$*, the interval* $[l, 11]$ *is not right-tight, such as* $[9, 11]$*.*

Given a fixed r, our goal is to identify all tight intervals $[l, r]$. Lemma 1 shows that to meet the left-tightness it suffices to consider only those positions l such that $l - 1 \in \mathrm{S}$. Lemmas 3-4 further propose to traverse the positions in the skyline S in descending order. Once we encounter a position in the skyline that violates the right-tight condition, we can stop. To this end, we define the ordered skyline as below.

Definition 7 (Ordered Skyline). *A skyline is ordered if its positions are ordered in ascending order. We write the skyline as the tuple* $\mathrm{S} = (p_1, p_2, \ldots, p_s)$ *in ascending order, i.e.,* $p_1 < p_2 < \cdots < p_s$*.*

Lemma 5. *Let* $\mathrm{S} = (p_1, p_2, \ldots, p_s)$ *be the ordered skyline at position* r*. We have* $p_s = r$ *and* $h(\mathrm{T}[p_1]) < h(\mathrm{T}[p_2]) < \cdots < h(\mathrm{T}[p_s])$*.*

Proof. Clearly $r \in \mathrm{S}$, hence $p_s = r$. For any $1 \le i < j \le s$, by the definition of skyline, p_j does not dominate p_i. Thus, by the definition of dominance, $h(\mathrm{T}[p_i]) < h(\mathrm{T}[p_j])$. Therefore $h(\mathrm{T}[p_1]) < \cdots < h(\mathrm{T}[p_s])$. □

Finally, instead of generating the ordered skyline at a position from scratch, we propose to derive it from the ordered skyline at the previous position, as formalized below.

Lemma 6. *Let* $\mathrm{S}(r) = (p_1, p_2, \ldots, p_s)$ *be the ordered skyline at position* r *and* $i = \min\{j \in [1, s] \mid h(\mathrm{T}[p_j]) < h(\mathrm{T}[r+1])\}$. *The ordered skyline at position* $r+1$ *must be* $\mathrm{S}(r+1) = (p_1, p_2, \ldots, p_i, r+1)$.

Proof. For any position $q \notin \mathrm{S}(r)$, there exists $p \in [0, r]$ that dominates q by definition. Moreover, $p \in [0, r+1]$. Thus $q \notin \mathrm{S}(r+1)$ by definition.

Consider any position $p_x \in \mathrm{S}(r)$ where $x \le i$. By the definition of i and Lemma 5, $h(\mathrm{T}[p_x]) \le h(\mathrm{T}[p_i]) < h(\mathrm{T}[r+1])$. Thus $r+1$ does not dominate p_x. Moreover, no position in $[0, r]$ dominates p_x. Therefore $p_x \in \mathrm{S}(r+1)$.

Consider any position $p_x \in \mathrm{S}(r)$ where $i < x \le s$ (if there is any). By the definition of i and Lemma 5, $h(\mathrm{T}[r+1]) < h(\mathrm{T}[p_x])$. Thus $r+1$ dominates p_x. Therefore $p_x \notin \mathrm{S}(r+1)$.

Finally, a position p dominates q only if $p > q$. Thus $r+1 \in \mathrm{S}(r+1)$. Therefore $\mathrm{S}(r+1) = (p_1, p_2, \ldots, p_i, r+1)$. □

Example 9. *Figure 4 shows how the skyline evolves as* r *increases from* 11 *to* 12. *At* $r = 11$, *the skyline is* $\mathrm{S}(11) = (0, 3, 9, 10, 11)$. *When advancing to* $r = 12$, *we compare* $h(\mathrm{T}[12])$ *with the hash values of the positions in* $\mathrm{S}(11)$ *from right to left. Since* $h(\mathrm{T}[12]) < h(\mathrm{T}[11]) < h(\mathrm{T}[10])$, *we remove* 11 *and* 10 *from the skyline, and then append* 12. *As a result, the updated skyline becomes* $\mathrm{S}(12) = (0, 3, 9, 12)$.

Based on the discussion above, we propose the tight interval generation algorithm. Algorithm 1 shows the pseudo-code. The algorithm maintains the ordered skyline at position r in a stack S (namely, monotonic stack). To generate all the tight intervals $[l, r]$ ending with r, we visit the positions in S in reverse order. Each time we visit a position p, by Lemma 1, $[p+1, r]$ must be left-tight. Thus, we only need to test if the right-tight condition is violated based on Lemma 3. If so, by Lemma 4, no right-tight interval can be generated anymore and the process terminates (Line 3). Otherwise, we remove this position from the stack as it must not be in the skyline of $r+1$ by Lemma 6 (Line 4) and produce a tight interval (Line 6). Finally, we append $r+1$ to the stack and the S becomes the skyline of $r+1$ (Line 7).

Algorithm 1: Tight Interval Generation for One Hash Function

Input: T: text of length n; $h \in \mathcal{H}$: a universal hash function.
Output: I: all tight intervals in T.

```
// dummy token hash values h(T[0]) = h(T[n+1]) = −∞
1 S.append(0);
2 for r ← 0 to n do
3     while h(T[S.back()]) > h(T[r+1]) do
4         S.pop_back();
5         l ← S.back() + 1;
6         I ← I ∪ {[l, r]};
7     S.append(r+1);
8 return I;
```

Theorem 1. *All intervals generated by Algorithm 1 are tight intervals in* T*; all tight intervals in* T *are generated by Algorithm 1.*

Proof. We first prove by induction that, at the end of each for-loop, S is the ordered skyline of $r+1$. We assume at the beginning of the for-loop, S is the ordered skyline of r. Initially, S = (0), which is exactly the ordered skyline of $r = 0$. Thus base case $r = 0$ is true. The while-loop guarantees all the positions, and only the positions, whose hash values are greater than $h(\mathrm{T}[r+1])$, are removed from S. By Lemma 6, S must be the ordered skyline of $r+1$ after appending $r+1$ at the end of the for-loop.

Next, we prove every interval $[l, r]$ added to I must be tight interval. By Lemma 1, the interval $[l, r]$ added, where $\mathsf{S.back()} = l - 1$, must be left-tight. p, which is the position after $\mathsf{S.back()}$ in the ordered skyline, must be the smallest position in the skyline that is greater than $l = \mathsf{S.back()} + 1$. By Lemma 3, Line 3 guarantees $[l, r]$ must be right-tight. Thus $[l, r]$ must be a tight interval.

Finally, we prove that every tight interval $[l, r]$ is output by the algorithm. Fix any tight interval $[l, r]$, by Lemma 1, at the start of the iteration of r, we have $l - 1 \in \mathrm{S}(r)$. Let $p = \arg\min_{p \in \mathrm{S}} p \geq l$, by the definition of skyline, no position in $[p+1, r]$ dominates p and by Lemma 2, p dominates every position in $[l, p-1]$, thus $h(\mathrm{T}[p]) = \min_{i \in [l,r]} h(\mathrm{T}[i])$, thus p is exactly c. Since $[l, r]$ is right-tight, then $h(\mathrm{T}[r+1]) < h(\mathrm{T}[p])$. By Lemma 4, the algorithm will pop p, thus $[l, r]$ will be reported. □

Lemma 7. *Let* T *be a text of length* n *and* h *is drawn from* $\mathcal{H}$*. Let* $\mathrm{S}(r)$ *be the skyline of* T *at position* r*. Then* $\mathbb{E}[|\mathrm{S}(r)|] = O(\log r)$.

Proof. For any fixed position p, $p \in \mathrm{S}(r)$ if and only if p is not dominated by any position $x \in [p+1, r]$, i.e., $h(\mathrm{T}[p]) < h(\mathrm{T}[x])$ for all $x \in [p+1, r]$. Since each position in $[p, r]$ becomes minimum with the same probability, thus $\Pr[p \in \mathrm{S}(r)] = \frac{1}{r-p+1}$. Summing over p, we have $\mathbb{E}[|\mathrm{S}(r)|] = H_r = O(\log r)$. □

Theorem 2. *The time and space complexities of Algorithm 1 are both* $O(n)$*. The algorithm generates* $O(n)$ *tight intervals in total. Excluding the storage for the set of tight intervals* I*, the space occupied by the stack is* $O(\log n)$ *in expectation.*

Proof. Each position in $[0, n+1]$ is added to S once and popped out once. Thus the time complexity is $O(n)$ where $n = |\mathrm{T}|$. For each position popped out from S, a tight interval is produced. Thus the algorithm generates $O(n)$ tight intervals. According to Lemma 7, the auxiliary space occupied by the stack is $O(\log n)$ in expectation. □

4.2 Multiple Hash Functions

This section discusses how to generate all tight intervals of a text T under a composite hash function $g = (h_1, h_2, \ldots, h_m) \in \mathcal{G}$. Throughout, the tightness is under g unless specified otherwise. An interval $[l, r]$ is left-tight (resp. right-tight) under g if and only if it is left-tight (resp. right-tight) under at least one of $h_1, \ldots, h_m$. As r moves from 0 to n, we maintain the skyline $\mathrm{S}_i(r)$ for each hash function h_i using Algorithm 1. The left-tightness can be easily checked using these skylines: the interval $[l, r]$ is left-tight if and only if $l - 1 \in \mathrm{S}_i \setminus \{r\}$ for some $i \in [1, m]$. For right-tightness, the interval is right-tight if and only if $[l, r]$ is right-tight under at least one hash function h_i, where $i \in [1, m]$. The test of the right-tightness can be facilitated with Lemma 3. Thus, we consider a heap-based method.

Heap-based Method. For each position $r \in [0, n]$, maintain its ordered skyline by a stack per hash function as in Algorithm 1. Note that each stack stores the positions in a skyline in increasing order. We find all the tight intervals by engaging a heap to "merge-sort" all the positions over the

m skylines in decreasing order. We stop once no hash function remains right-tight. This process has two drawbacks which lead to a complexity of $O(mn \log^2 m)$ (see our technical report [51]). Different skylines may share the same position, i.e., the merge-sort may generate the same position multiple times. In other words, a heap pop operation may produce no new tight interval. 2) For a given h_i and a skyline position p, even if $[p+1, r]$ is not right-tight under h_i, the interval $[p+1, r]$ can still be right-tight under g if some other h_j is right-tight. These invalidate the simple $O(n)$ amortized argument used for Algorithm 1 and incur extra heap and duplication costs.

Global List Method. To reduce the time complexity and address the two drawbacks of the heap-based method, Algorithm 2 maintains a linked list `list`, facilitated with an array of size n, to retrieve the positions that present in at least one skyline. Specifically, each cell in `list` has a predecessor and a successor, initially linked such that $\texttt{list}[i].pred = i-1$ and $\texttt{list}[i].succ = i+1$; moreover, for each $i \in [1, n]$, `list` maintains the total number of hash functions that include i in their skylines in $\texttt{list}[i].count$, which is initially 0 (Line 6). When r varies from 0 to n (Line 7), Lines 8–26 generate all the tight intervals in the form of $[*, r]$ and maintain the linked list in three steps. (1) Lines 8–13 compute the leftmost position (`leftbound`) in the stacks such that $[\texttt{leftbound}+1, r]$ is right-tight for at least one hash function. The correctness follows from Lemma 4. (2) Using `list`, Lines 14–17 visit all the positions that are included by at least one skyline, and generate the tight intervals. The left-tightness is guaranteed by Lemma 1. (3) Lines 18–29 maintain the stack and the linked list. Specifically, Lines 19, 26, and 28 are adopted from Algorithm 1, which maintain the stack. Lines 20–25 maintain the linked list when positions are popped from the stacks, each time a position is popped from the stacks, its counter in the linked list is decremented, and once the counter reaches 0, the position is bypassed in the linked list by directly connecting its predecessor and successor. When $r+1$ is pushed onto a stack (Line 28), the counter $\texttt{list}[r+1].count$ is incremented in Line 29. The above description also establishes the correctness of Algorithm 2, which is formally stated in Theorem 3.

Theorem 3. *The set* I *reported by Algorithm 2 contains exactly all the tight intervals of* T *under* $g \in \mathcal{G}$.

Theorem 4. *Given a text* T *with length* n*, and a hash function* $g = (h_1, \ldots, h_m) \in \mathcal{G}$*, the expected number of tight intervals in* T *under* g *is* $O(nm)$.

Proof. Let $D(n, m)$ be the total number of tight intervals in a text T of length n under g. $\mathbb{E}[D(n,m)] = \sum_{1 \le l \le r \le n} \Pr([l, r] \text{ is tight})$. Observe that when $l = 1$, the interval $[l, r]$ is always left-tight; and when $r = n$, the interval $[l, r]$ is always right-tight. Therefore, we mainly consider interior intervals where $1 < l \le r < n$.

Recall in Definition 4, an interval $[l, r]$ is tight when it is left- and right-tight. For an interval $[l, r]$, we define two events. Let $\mathcal{L}$ be the event that $[l, r]$ is left-tight, and $\mathcal{R}$ be the event that $[l, r]$ is right-tight. Based on union bound, probability $\Pr[[l, r] \text{ is tight}]$ is

$$\Pr[\mathcal{L} \wedge \mathcal{R}] = \Pr[\mathcal{L}] + \Pr[\mathcal{R}] - \Pr[\mathcal{L} \vee \mathcal{R}].$$

Consider any interior interval $[l, r]$ of length $s = r - l + 1$. Under a hash function h_i, There are $s+2$ hash values, namely $h_i(\mathrm{T}[l-1]), \ldots, h_i(\mathrm{T}[r]), h_i(\mathrm{T}[r+1])$, each of which is equally likely to be the minimum. Hence the probability that $[l, r]$ is not left-tight under h_i, i.e.,, $h_i(\mathrm{T}[l-1])$ is not the minimum among the first $s+1$ hash values is $\frac{s}{s+1}$. The probability that $[l, r]$ is not tight under h_i, i.e., the minimum among the $s+2$ hash values is neither $h_i(\mathrm{T}[l-1])$ nor $h_i(\mathrm{T}[r+1])$ is $\frac{s}{s+2}$. By Definition 4, $[l, r]$ is not left-tight (or right-tight, resp.) under g if and only if $[l, r]$ is not left-tight (or right-tight, resp.) under h_i for every $i \in [m]$, and since the m hash functions are independent, $\Pr[\mathcal{L}] = \Pr[\mathcal{R}] = 1 - \left(\frac{s}{s+1}\right)^m$. Besides, the probability that $[l, r]$ is neither left-tight

Algorithm 2: Tight Interval Generation (Global List Method)

Input: T: text of length n; g: hash function $(h_1, h_2, \ldots, h_m)$.
Output: I: all tight intervals in T.
// Use one stack S[i] for one hash function, $\forall i \in [m]$, dummy tokens $h_i(T[0]) = h_i(T[n+1]) = -\infty$.

```
for i ← 1 to m do
    S[i].append(0) ;
for i ← 1 to n do
    list[i].pred ← i − 1 ;
    list[i].succ ← i + 1 ;
    list[i].count ← 0 ;
for r ← 0 to n do
    leftbound ← +∞;
    // Get the leftmost position (across all the hash functions) in the skyline such
       that [leftbound + 1, r] is right-tight.
    for i ← 1 to m do
        pointer ← S[i].size − 1;
        while h_i(T[S[i][pointer]]) > h_i(T[r + 1]) do
            pointer ← pointer − 1;
        leftbound ← min(S[i][pointer], leftbound);
    // Use the list to access all the positions to the right of leftbound that appear
       in at least one skyline. Generate tight intervals.
    pointer ← r;
    while pointer > leftbound do
        pointer ← list[pointer].pred;
        Add [pointer + 1, r] to I;
    // Maintain the skylines and the list - once a position that is not included in
       any skyline, bypass it in the linked list.
    for i ← 1 to m do
        while h_i(T[S[i].back()]) > h_i(T[r + 1]) do
            list[S[i].back()].count ← list[S[i].back()].count − 1;
            if list[S[i].back()].count = 0 then
                u ← list[S[i].back()].pred;
                v ← list[S[i].back()].succ;
                list[u].succ ← v;
                list[v].pred ← u;
            S[i].pop_back();
        if r + 1 ≤ n then
            S[i].append(r + 1);
            list[r + 1].count ← list[r + 1].count + 1;
return I;
```

nor right-tight means $[l, r]$ is neither left-tight nor right-tight under h_i for every $i \in [m]$, and thus

$1 - \Pr[\mathcal{L} \vee \mathcal{R}] = \left(\frac{s}{s+2}\right)^m$. Substitute into Equation 4.2:

$$\begin{aligned} P[\mathcal{L} \wedge \mathcal{R}] &= 1 - 2\left(\frac{s}{s+1}\right)^m + \left(\frac{s}{s+2}\right)^m \quad \text{(using Taylor expansion)} \\ &\le 1 - 2\left(1 - \frac{m}{s+1}\right) + \left(1 - \frac{2m}{s+2} + \frac{2m(m-1)}{(s+2)^2}\right) \\ &= \frac{2m}{s+1} - \frac{2m}{s+2} + \frac{2m(m-1)}{(s+2)^2} = \frac{2m}{(s+1)(s+2)} + \frac{2m(m-1)}{(s+2)^2} \le \frac{4m^2}{s^2}. \end{aligned}$$

For interior intervals with length $1 \le s \le m$, we use the trivial bound $\Pr[[l, r] \text{ is tight}] \le 1$, and for $m+1 \le s \le n-2$, we use the tail bound $\sum_{s=m+1}^{\infty} \frac{1}{s^2} \le \frac{1}{m}$. Therefore, $\mathbb{E}[D(n, m)] = \sum_{1 \le l \le r \le n} \Pr[[l, r] \text{ is tight}] \le 2n + \sum_{s=1}^{m}(n-s-1) + \sum_{s=m+1}^{n-2}(n-s-1)\frac{4m^2}{s^2} = 2n + nm + 4nm \le 7nm = O(nm)$. □

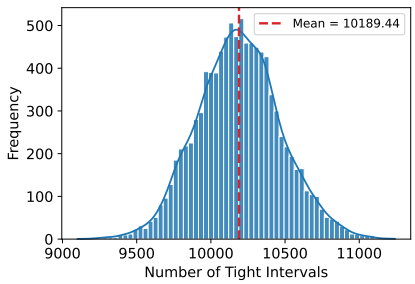


Fig. 5. Empirical distribution of the number of tight intervals over 10,000 random draws ($n = 1,000, m = 8$).

Empirical Concentration. By applying Markov's inequality to our expectation bound ($7nm$), with probability at least 0.9, the number of tight intervals is at most $70nm$. To investigate the tightness of this bound in practice, we analyzed the empirical distribution of the number of tight intervals generated by random hash functions. We conducted 10,000 independent trials using random hash functions on a text T of length $n = 1,000$ from the Arxiv dataset (see Section 7 for details of the dataset), with $m = 8$. Figure 5 presents the frequency histograms of the number of tight intervals. The distribution is highly concentrated around its mean $1.02 \times 10^4 \approx 1.25nm$ with a standard deviation of $\sigma \approx 281.4$, far below the theoretical bound of $70nm$. This result demonstrates that our method remains efficient and robust in practice.

Theorem 5. *The expected time complexity of Algorithm 2 is $O(mn)$. The space complexity consists of three parts: (1) $O(mn)$ expected for the output* I*, (2) $O(n)$ worst-case for the linked list, and (3) $O(m \log n)$ expected for the m stacks.*

Proof. The linked list is an array of size $O(n)$. By Lemma 7, each of the m skylines has an expected size of $O(\log r)$ at position r, resulting in an expected total space of $O(m \log n)$ for all stacks. The output set I has expected size $O(mn)$ according to Theorem 4. Regarding time complexity, Lines 14–17 generate one tight interval in $O(1)$ time per iteration, contributing a total of $O(mn)$ expected time. The remaining lines (Lines 7–29) maintain m independent instances of Algorithm 1, each running in $O(n)$ time by Theorem 2, yielding a total of $O(mn)$. Therefore, the overall expected time complexity of Algorithm 2 is $O(mn)$. □

5 Partitioning by Tight Intervals

This section shows how to derive rectangles from the generated tight intervals of T, and then proves that these rectangles constitute a partition of all subsequences of T.

Here we define center of (tight) interval. Intuitively, the center is simply the position with the minimum hash value under each hash function h_i. Any interval that shares the same hash value under g must include these positions. The center therefore characterizes the shortest interval attaining that hash value. We formalize this definition below.

Definition 8 (Center of Interval). *The center of the interval $[l, r]$ under a hash function $g = (h_1, \ldots, h_m) \in \mathcal{G}$ is $\mathtt{C}^{l,r} = (c_1^{l,r}, \ldots, c_m^{l,r})$ where $c_i^{l,r} = \arg\min_{x \in [l,r]} h_i(\mathrm{T}[x])$ is called the center of $[l, r]$ under h_i – the index in $[l, r]$ of which the token has the smallest hash value under h_i. We define the leftmost center of $[l, r]$ under g as $\mathtt{C}_{\min}^{l,r} = \min_{i \in [m]} c_i^{l,r}$ and rightmost center $\mathtt{C}_{\max}^{l,r} = \max_{i \in [m]} c_i^{l,r}$. We omit the superscript when the context is clear.*

Example 10. *Consider the example in Figure 2. The center of the interval $[3, 6]$ is $(3, 5)$ under the hash function $g = (h_1, h_2)$, since $h_1(\mathrm{T}[3]) = 22$ is the minimum among $h_1(\mathrm{T}[3]), \ldots, h_1(\mathrm{T}[6])$, and $h_2(\mathrm{T}[5]) = 4$ is the minimum among $h_2(\mathrm{T}[3]), \ldots, h_2(\mathrm{T}[6])$.*

Based on tie breaking and collision-free assumption, no two positions of T having tokens share the same hash value under any $h \in \mathcal{H}$, thus, the center of an interval is unique. In other words, Definition 8 is well-defined: every interval $[l, r]$ admits exactly one center.

Definition 9 (Rectangle of an Interval). *The rectangle of an interval $[l, r]$ under $g \in \mathcal{G}$ is defined as $\mathrm{rec}(l, \mathtt{C}_{\min}, \mathtt{C}_{\max}, r, g) = \{\mathrm{T}[i, j] \mid i \in [l, \mathtt{C}_{\min}], j \in [\mathtt{C}_{\max}, r]\}$ where $\mathtt{C}_{\min}$ and $\mathtt{C}_{\max}$ are the leftmost and rightmost centers of $[l, r]$, respectively.*

Example 11. *As shown in Figure 2, for the interval $[3, 6]$, its center is $\mathtt{C} = (3, 5)$ under $g = (h_1, h_2)$. Hence, $\mathtt{C}_{\min} = 3$, $\mathtt{C}_{\max} = 5$, and $\mathrm{rec}(3, 3, 5, 6, g) = \{\mathrm{T}[i, j] \mid (i, j) \in [3, 3] \times [5, 6]\} = \{\mathrm{T}[3, 5], \mathrm{T}[3, 6]\}$.*

Lemma 8 shows Definition 9 is consistent with Definition 2: all subsequences in $\mathrm{rec}(l, \mathtt{C}_{\min}, \mathtt{C}_{\max}, r, g)$ share the same hash value.

Lemma 8. *Let $[l, r]$ and $[l', r']$ be two intervals of T, then $g(\mathrm{T}[l, r]) = g(\mathrm{T}[l', r'])$ if and only if they share the same center. Furthermore, for any $i \in [l, \mathtt{C}_{\min}^{l,r}]$ and $j \in [\mathtt{C}_{\max}^{l,r}, r]$, $g(\mathrm{T}[i, j]) = (h_1(c_1^{l,r}), \ldots, h_m(c_m^{l,r}))$.*

Proof. Firstly, if $g(\mathrm{T}[l, r]) = g(\mathrm{T}[l', r'])$, then for each $i \in [m]$ we have $h_i(\mathrm{T}[l, r]) = h_i(\mathrm{T}[l', r']) \doteq v_i$. The collision-free property of h_i means that there is a unique position c_i such that $h_i(\mathrm{T}[c_i]) = v_i$, so c_i is the center of both intervals, i.e., $c_i = c_i^{l,r} = c_i^{l',r'}$.

Secondly, if both intervals share the same center, i.e., $c_i^{l,r} = c_i^{l',r'} \doteq c_i$ for all $i \in [m]$, then by definition $h_i(\mathrm{T}[l, r]) = h_i(\mathrm{T}[c_i]) = h_i(\mathrm{T}[l', r'])$ for all $i \in [m]$, so $g(\mathrm{T}[l, r]) = g(\mathrm{T}[l', r'])$.

For every $i \in [l, \mathtt{C}_{\min}^{l,r}]$, $j \in [\mathtt{C}_{\max}^{l,r}, r]$, and $k \in [m]$, because $c_k^{l,r} \in [\mathtt{C}_{\min}^{l,r}, \mathtt{C}_{\max}^{l,r}] \subseteq [i, j]$ (by the definition of the leftmost/rightmost center), we have $c_k^{l,r} = c_k^{i,j} \doteq c_k$. Thus, $h_k(\mathrm{T}[l, r]) = h_k(\mathrm{T}[i, j]) = h_k(\mathrm{T}[c_k])$ for all $k \in [m]$ and thus $g(\mathrm{T}[l, r]) = g(\mathrm{T}[i, j])$. □

Example 12. *As shown in Figure 2, for the interval $[3, 6]$, we have $\mathtt{C} = (3, 5)$ under the hash function $g = (h_1, h_2)$. Thus, $\mathtt{C}_{\min} = 3$ and $\mathtt{C}_{\max} = 5$. Therefore, $\mathrm{T}[3, 5]$ and $\mathrm{T}[3, 6]$ share the same hash value $(h_1(\mathrm{T}[3]), h_2(\mathrm{T}[5])) = (22, 4)$ under g.*

We omit the hash function g in the rectangle notation when it is clear from the context.

A set of rectangles P is a partition of all the subsequences in the text T if it satisfies the following constraints.

Definition 10 (Partition). *Let $g \in \mathcal{G}$ be a hash function and* T *a text. A set* P *of rectangles* $\mathrm{rec}(a,b,c,d,g)$ *is a partition of* T *under g if it satisfies both the disjointness condition and coverage condition.*

- *Disjointness: Any two rectangles* $\mathrm{rec}(a,b,c,d,g) \neq \mathrm{rec}(a',b',c',d',g)$ *in* P *have* $\mathrm{rec}(a,b,c,d,g) \cap \mathrm{rec}(a',b',c',d',g) = \emptyset$.
- *Coverage: Every subsequence* $\mathrm{T}[i,j]$ *of* T *lies in at least one rectangle* $\mathrm{rec}(a,b,c,d,g) \in \mathrm{P}$ *such that* $i \in [a,b]$ *and* $j \in [c,d]$.

Consider a hash function $g \in \mathcal{G}$, recall the definition of tight intervals under g in Definition 4. As we shall show in Theorem 6, the rectangles of tight intervals form a partition of T. To get an intuition, if $[l,r]$ is tight under g, no subsequences "outside" the interval can have the same hash value as $\mathrm{T}[l,r]$.

Theorem 6. *Given a text* T *of length n and a hash function $g \in \mathcal{G}$. Let* P *be the set of all rectangles of tight intervals under g in* T*. We have i)* P *is a partition of all subsequences in* T *and ii) No two different rectangles in* P *have identical hash values under g.*

Proof. (i) We first prove coverage by showing that every subsequence $\mathrm{T}[i,j]$ is in the rectangle of a tight interval under $g = (h_1, \ldots, h_m)$. Consider any subsequence $\mathrm{T}[i,j]$, denote its center under g by $\mathrm{C}^{i,j} = (c_1^{i,j}, c_2^{i,j}, \ldots, c_m^{i,j})$. Construct an interval $[l,r]$:

$$l = \max\{u \le i \mid u = 1 \text{ or } \exists x \in [1,m] \text{ s.t. } h_x(\mathrm{T}[u-1]) < h_x(\mathrm{T}[c_x^{i,j}])\},$$

$$r = \min\{u \ge j \mid u = n \text{ or } \exists x \in [1,m] \text{ s.t. } h_x(\mathrm{T}[u+1]) < h_x(\mathrm{T}[c_x^{i,j}])\}.$$

By construction, $[l,r]$ is both left- and right-tight with center $\mathrm{C}^{l,r} = \mathrm{C}^{i,j}$ under g. Since $l \le i \le \mathrm{C}^{i,j}_{\min} = \mathrm{C}^{l,r}_{\min}$ and $\mathrm{C}^{i,j}_{\max} = \mathrm{C}^{l,r}_{\max} \le j \le r$, $\mathrm{T}[i,j] \in \mathrm{rec}(l, \mathrm{C}^{l,r}_{\min}, \mathrm{C}^{l,r}_{\max}, r, g)$, i.e., the subsequence $\mathrm{T}[i,j]$ is in the rectangle of $[l,r]$ under g. Thus, every subsequence is covered.

We prove no subsequence is in the rectangles of two distinct tight intervals under g by contradiction. If a subsequence $\mathrm{T}[i,j]$ is in the rectangles of two distinct tight intervals $[l,r]$ and $[l',r']$ in P , then $l \le i \le \mathrm{C}^{l,r}_{\min} \le \mathrm{C}^{l,r}_{\max} \le j \le r$ and $l' \le i \le \mathrm{C}^{l',r'}_{\min} \le \mathrm{C}^{l',r'}_{\max} \le j \le r'$. By the definition of rectangles, we have $g(\mathrm{T}[i,j]) = g(\mathrm{T}[l,r]) = g(\mathrm{T}[l',r'])$, thus by Lemma 8, the centers of $[l,r]$ and $[l',r']$ must be the same, i.e., $\mathrm{C}^{l,r} = \mathrm{C}^{l',r'}$. By Lemma 8, this forces $[l,r] = [l',r']$, contradicting that $[l,r]$ and $[l',r']$ are distinct. Thus the rectangles of two different tight intervals are disjoint.

Combining coverage and disjointness, the set of all rectangles of tight intervals P is a partition.

(ii) We prove that no two rectangles in P have identical hash values under g by contradiction. For two different tight intervals $[l,r]$ and $[l',r']$, if $g(\mathrm{T}[l,r]) = g(\mathrm{T}[l',r'])$, then by Lemma 8, $\mathrm{C}^{l,r} = \mathrm{C}^{l',r'}$, thus $\mathrm{C}^{l,r}_{\min} = \mathrm{C}^{l',r'}_{\min} \doteq \mathrm{C}_{\min}$ and $\mathrm{C}^{l,r}_{\max} = \mathrm{C}^{l',r'}_{\max} \doteq \mathrm{C}_{\max}$, thus the rectangles of $[l,r]$ and $[l',r']$ at least overlap on $\mathrm{T}[\mathrm{C}_{\min}, \mathrm{C}_{\max}]$, contradicting that P is a partition. Thus, the rectangles in P all have different hash values. □

6 The LSHAlign Algorithm

Consider a hash function g and a text T. Section 4 generates all the tight intervals. In this section, we extend Algorithm 2 to generate not only tight intervals but also their rectangles and hash values under g and eventually integrate it into our end-to-end algorithm LSHAlign. We start with an example.

Example 13. *Let $g = (h_1, h_2, \ldots, h_m)$ be a hash function with each h_i, $i \in [m]$, having the target domain of integers. Denote by $V = (v_1, \ldots, v_m)$ the hash value of a subsequence under g, thus $V \in \mathbb{Z}^m$. Consider a function f, which we call the* ***composite hash function****, on V defined as $f(V) = (v_1 \times q + v_2 \times q^2 + \cdots + v_m \times q^m) \bmod Q$, where q is a prime number and $Q > q$ a large prime. f*

maps the hash value V to a scalar value in $\mathbb{Z}_Q$, called the ***composite hash value****. For two subsequences and their hash values V_1 and V_2, if $V_1 = V_2$, then their composite hash value $f(V_1) = f(V_2)$. Thus, we put all subsequences with the same composite hash value $f(V)$ into one bucket.*

We want to design an efficient algorithm for generating tight intervals with their rectangles and the corresponding composite hash values. To do so, the composite hash function needs to have properties similar to the function f defined in the example, i.e., having three properties below. 1) Computed from m components, each determined only by one value in V, e.g., $v_1 \times q$, $v_2 \times q^2$, $\cdots$, $v_m \times q^m$, are the m components. 2) The "addition" among these components does not require them to be computed in a specific ordering. For example, the addition in $\mathbb{Z}_Q$ is commutative. 3) When V has one value, say v_i, changed to v_i', while other values remain the same, denote the new hash value as V', one can compute $f(V')$ from $f(V)$, v_i and v_i', in $O(1)$ computation. In the example, $f(V') = (f(V) + (-v_i \times q^i) + v_i' \times q^i) \bmod Q$. We call the three properties, respectively, *component-wise form*, *abelian group structure*, and *incremental updatability*, and introduce a new notation $\oplus$ to generalize the "addition" described above. We formalize the composite hash function f on $V = (v_1, \ldots, v_m)$ below

$$f(V) = f_1(v_1) \oplus f_2(v_2) \oplus \cdots \oplus f_m(v_m),$$

where $f_1, f_2, \cdots, f_m$ are unary functions where each unary function $f_i(\cdot)$ and each group operation $\oplus$ can be evaluated in $O(1)$ time.

For any two hash values V, V' that differ in exactly one coordinate i, we can compute the composite hashvalue $f(V')$ with $f(V) \oplus f_i(v_i') \oplus (-f_i(v_i))$ in $O(1)$ time.

Generate composite hash values. To avoid imposing an extra m multiplicative factor to the complexity for computing the composite hash values, we adapt Algorithm 2 so that, for each tight interval, both its leftmost center $\mathtt{C}_{\min}$ and its corresponding composite hash value $f(g(\mathtt{C}))$ are maintained. Recall in Algorithm 2, r advances from 0 to n. During this process, we propose to maintain in $\mathtt{list}[i]$ the delta of the composite hash value when the left boundary l of the interval $[l, r]$ shifts from $i+1$ to i. Specifically, consider a hash function h_j that has i in its skyline. When the left boundary l moves from $i+1$ to i, the center under h_j will shift from a previous position, denoted as c_j, to i. This is because the hash value of i is smaller than that of c_j (according to Lemma 5). This causes the composite hash value $f(g(\mathtt{C}))$ to change with a delta of $f_i(h_j(\mathtt{T}[i])) - f_i(h_j(\mathtt{T}[c_j]))$. We record in $\mathtt{list}[i].\Delta$ the sum (as defined by $\oplus$) of these deltas across all m hash functions at position i. If a hash function h_j does not include i in its skyline, it contributes 0 to $\mathtt{list}[i].\Delta$.

Algorithm 3 shows this adaptation. For a fixed right boundary r, we traverse all positions that are left-tight with respect to r in the leftward direction. During this traversal, comphash maintains the current composite hash value, which is initialized in Line 1. Each time the pointer pointer moves left, comphash is updated by incorporating $\mathtt{list}[\mathtt{pointer}].\Delta$ using the operator $\oplus$ (Line 5). Since $\mathtt{list}[\mathtt{pointer}].\Delta$ captures the change in the composite hash value whenever a hash function h_i has its center c_i updated (i.e., when pointer belongs to the skyline of h_i), comphash always holds the correct composite hash value corresponding to the interval $[\mathtt{pointer}+1, r]$. As the skylines evolve, the associated Δ values are correspondingly updated in Lines 9, 10, 17, 18, 19, and 22.

Generate $\mathtt{C}_{\max}$ and $\mathtt{C}_{\min}$ for each tight interval. For each tight interval $[\mathtt{pointer}+1, r]$ generated in Line 4, denote by $\mathtt{C}$ its center. Its leftmost center $\mathtt{C}_{\min}$ can be obtained directly in our algorithm – it corresponds to the next position (to the right of $\mathtt{list}[\mathtt{pointer}]$) with a non-zero count, which is stored in $\mathtt{list}[\mathtt{pointer}].succ$. To compute $\mathtt{C}_{\max}$, we reverse the text $\mathtt{T}$ and execute Algorithm 3 to obtain the $\mathtt{C}_{\min}$ values of all tight intervals in the reversed text. For a tight interval $[l, r]$ in the original text, its $\mathtt{C}_{\max}$ is then given by $n - \mathtt{C}_{\min} + 1$, where $\mathtt{C}_{\min}$ is the corresponding value computed

in the reversed text. Finally, to merge the results from both directions, we build a hash table that stores all tight intervals and their associated leftmost and rightmost centers.

Algorithm 3: Extended Tight Interval Generation

```
   // Replace Lines 14-29 of Algorithm 2 with the following
1  pointer ← r; comphash ← 0;
2  while pointer > leftbound do
3      pointer ← list[pointer].pred;
       // (l, Cmin, Cmax, r, composite hash value)
4      Add (pointer + 1, list[pointer].succ, ∗, r, comphash) to I;
5      comphash ← comphash ⊕ list[pointer].Δ;
6  for i ← 1 to m do
7      while h_i(T[S[i].back()]) > h_i(T[r + 1]) do
8          list[S[i].back()].count ← list[S[i].back()].count − 1;
9          v ← f_i(h_i(T[S[i].back()]));
           // Unregister v from the being popped position's linked list.
10         list[S[i].back()].Δ ← list[S[i].back()].Δ ⊕ (−v);
11         if list[S[i].back()].count = 0 then
12             u ← list[S[i].back()].pred;
13             v ← list[S[i].back()].succ;
14             list[u].succ ← v;
15             list[v].pred ← u;
16         S[i].pop_back();
           // Unregister v from the predecessor (in stack) of the being-popped position.
17         list[S[i].back()].Δ ← list[S[i].back()].Δ ⊕ v;
18     v ← f_i(h_i(T[r + 1]));
       // Register v to the pred (in stack) of the being-appended position
19     list[S[i].back()].Δ ← list[S[i].back()].Δ ⊕ (−v);
20     if r + 1 ≤ n then
21         S[i].append(r + 1);
22         list[r + 1].Δ ← list[r + 1].Δ ⊕ v;
23         list[r + 1].count ← list[r + 1].count + 1;
```

Theorem 7. *The time and space complexity of Algorithm 3 is asymptotically the same as those of Algorithm 2.*

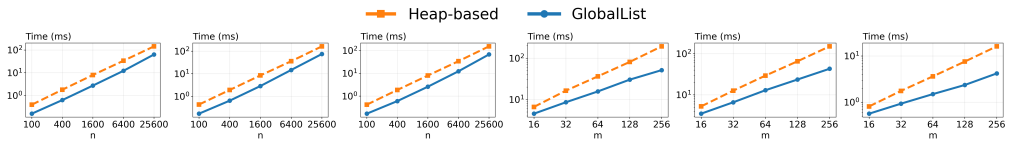
Proof. We maintain the array `list` without the effort of initialization due to Line 10, Algorithm 3. The update cost is identical to that of generating the skylines in Algorithm 2. The computation of $\mathsf{C}_{\min}$ incurs no additional cost, whereas computing $\mathsf{C}_{\max}$ effectively doubles the total cost of the algorithm – we execute the same procedure on the reversed text T to obtain the $\mathsf{C}_{\min}$ of the reversed interval and then use hash table to assemble it back to the original interval. Therefore, the overall complexity of Algorithm 3 is asymptotically the same as that of Algorithm 2. □

LSHAlign. Algorithm 4 outlines the end-to-end procedure LSHAlign for identifying all pairs of subsequences of two texts T and S that are LSH-similar (as defined in Section 2.3). The algorithm performs L independent trials (Line 1). In each trial, it first samples a hash function

Algorithm 4: LSHAlign (T, S, m, L)

Input: T and S: two texts; m and L: two positive integers.
Output: $\mathcal{R}$: all LSH-similar subsequence pairs of T and S.

```
1  for i ← 1 to L do
2      randomly draw m hash functions h_1, ..., h_m from H;
3      g_i = (h_1, h_2, ..., h_m);
4      P_T ← PARTITION(T, g_i) returned by Algorithm 3;
5      P_S ← PARTITION(S, g_i) returned by Algorithm 3;
6      foreach (l, p, q, r, comphash) ∈ P_T do
7          append (l, p, q, r) to HT[comphash];
8      foreach (l, p, q, r, comphash) ∈ P_S do
9          foreach (l', p', q', r') ∈ HT[comphash] do
10             add (T[a, b], S[c, d]) to R for every a ∈ [l', p'], b ∈ [q', r'], c ∈ [l, p], and d ∈ [q, r];
11 return R;
```



(a) Arxiv (m=64) (b) PAN11 (m=64) (c) PAN25 (m=64) (d) Arxiv (n=1600) (e) PAN11 (n=1600) (f) PAN25 (n=1600)

Fig. 6. Evaluating partition generation time. Varying text length n and the # of hash functions m, fixing $L = 1$.

$g_i = (h_1, h_2, \ldots, h_m) \in \mathcal{G}$ (Line 3) and partitions both T and S into rectangles using Algorithm 3 (Lines 4–5). Each rectangle is represented as a tuple $(l, p, q, r, \text{comphash} = f(g(\mathtt{C})))$, where $[l, r]$ is the corresponding tight interval, C is the center of $[l, r]$, $p = \mathtt{C}_{\min}$, $q = \mathtt{C}_{\max}$, and comphash is the composite hash value.

Next, we construct a hash table HT indexed by comphash, which stores all rectangles generated from T (Lines 6- 7). For each rectangle generated from S with hash value comphash, we probe the bucket HT[comphash] (Lines 8- 9). Finally, we enumerate all LSH-similar subsequence pairs by combining (a, b) from the rectangle in T (retrieved from the bucket) with every (c, d) from the corresponding rectangle in S. Each pair $(\mathrm{T}[a, b], \mathrm{S}[c, d])$ is then inserted into the result set $\mathcal{R}$ (Line 10).

7 Experiment

Environment. We implemented all methods and baselines in C++ and compiled them using GCC 11.4.0 with -O3 optimization. All experiments were conducted on a single thread of a server equipped with an AMD EPYC 7352 CPU@2.3GHz and 80GB of RAM.

Datasets. We used three datasets for evaluation. (1) Arxiv[1]: a corpus of one million uncopyrighted scientific papers from the Pile-ArXiv dataset on Hugging Face. We extracted the introduction section from each paper and selected 100 most similar pairs using SPECTER [10] embeddings for evaluation. (2) PAN11 and (3) PAN25[2]: two benchmarks from the PAN plagiarism detection series [34]. PAN11

[1]https://huggingface.co/datasets/timaeus/pile-arxiv
[2]https://zenodo.org/records/3250095 and https://zenodo.org/records/14969012

is a classic benchmark containing manually inserted plagiarism cases created via crowdsourcing, whereas PAN25 is a recent benchmark featuring plagiarism generated through paraphrasing by large language models (LLMs). We use the document pairs annotated with plagiarism cases from both benchmarks for evaluation. All texts were tokenized using NLTK (https://www.nltk.org/).

Parameters and Settings. We vary four parameters in evaluation: n, m, L and θ. Here, n denotes the token length of each text. LSH employs a total of $m \times L$ hash functions, organized into L bands of m hash functions each (see Section 2.3). θ is the Jaccard similarity threshold. For a fixed n, texts longer than n tokens are truncated to the first n tokens, while shorter texts are concatenated until they reach at least n tokens and then truncated to exactly n. Given two texts, the ground truth includes all subsequence pairs whose Jaccard similarity exceeds θ; an output pair is considered a true positive (TP) if it appears in the ground truth. To evaluate a method, for *accuracy*, we report precision = $TP/(TP + FP)$, recall = $TP/(TP + FN)$ and F1 score = $2TP/(2TP + FP + FN)$. For *efficiency*, we report query time, peak memory usage and the number of tight intervals. Note that for evaluating accuracy, 1) computing the ground truth set needs expensive enumeration of $O(n^4)$ subsequence pairs, thus we can only afford small n, so we set $n = 100$; 2) similar short subsequences pairs are likely to be not meaningful, so we only consider subsequence pairs with length ≥ 32 as candidates.

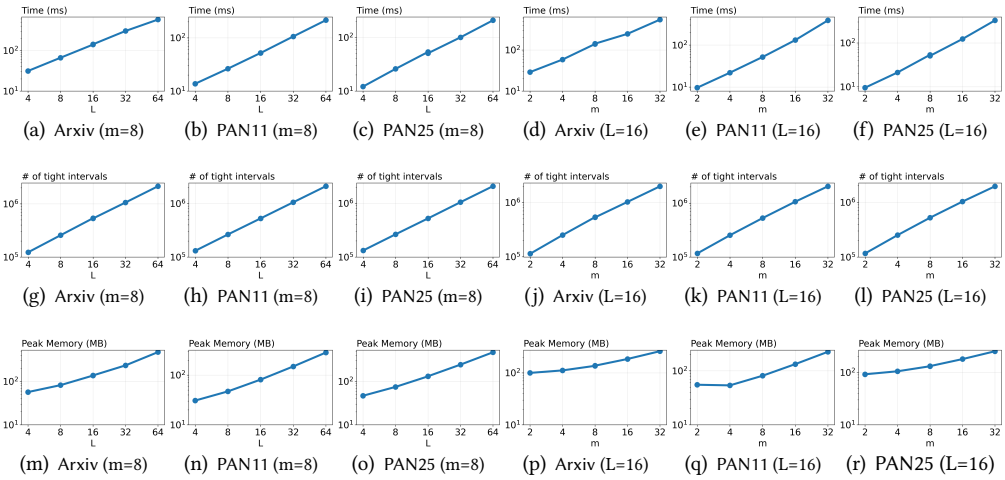
Compared Methods We compare the following four methods.

- `Heap-based` Method (proposed in Section 4.2, detailed in our technical report [51]: Heap-based method maintains the skyline for each hash function and identifies all tight intervals in a "merge-sort" manner using a heap. The expected time for hash table construction and for probing is $O(m(|\mathrm{T}| + |\mathrm{S}|)L \log^2 m)$ for texts T and S.
- `GlobalList`: Algorithm 2 generates tight intervals and Algorithm 3 adds rectangles and hash values to the tight intervals. The overall time complexity is $O((|\mathrm{T}| + |\mathrm{S}|)mL)$ expected.
- AllAlign [16]: It uses k independent hash functions. Given a text of length n, for each hash function, it merges adjacent subsequences of the text with the same min-hash value into a single group (namely, a compact window), thereby assigning each subsequence to nk compact windows in $O(kn \log n)$ time. Then, compact windows from two texts T and S are matched by identifying subsets of "collided" compact windows that share at least $\lceil \theta k \rceil$ min-hashes using segment trees. The worst-case time complexity of AllAlign is $O(k^2 n^4 \log^3 n)$.
- OphAlign [36]: Given a text and a query, the near-duplicate text alignment problem aims to find all subsequences in the text that are similar to the query. To extend this problem to all-pair near-duplicate text alignment between two texts T and S, we enumerate all subsequences in T, treat each subsequence as an individual query, and perform near-duplicate text alignment against S. OphAlign [36] is a state-of-the-art near-duplicate text alignment algorithm. Its per-query time complexity is $O\big((|\mathrm{S}| + k)(\log(|\mathrm{S}| + k) + k \log k)\big)$, leading to a total runtime of $O(|\mathrm{T}|^2(|\mathrm{S}| + k)(\log(|\mathrm{S}| + k) + k \log k))$ when all subsequences are considered.

7.1 Evaluating Partition Generation Time

We compare the partition generation time of our two methods, `GlobalList` (Section 4) and `Heap-based` (technical report [51]), on three datasets, Arxiv, PAN11, and PAN25. For partition generation, the two methods differ only in the partition generation process, whose time complexities are $O(mn)$ for `GlobalList` and $O(mn \log^2 m)$ for `Heap-based`, given a text of length n for each LSH band. Thus we vary both n and m and report the tight interval generation time.

Figures 6(a)–(c) show the results on the three datasets when $m = 64$ and $L = 1$, while varying the text length n from 100 to 25,600. The partition generation time of both methods increased

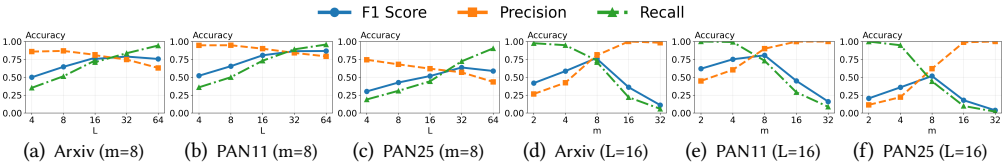


(a) Arxiv (m=8) (b) PAN11 (m=8) (c) PAN25 (m=8) (d) Arxiv (L=16) (e) PAN11 (L=16) (f) PAN25 (L=16)

(g) Arxiv (m=8) (h) PAN11 (m=8) (i) PAN25 (m=8) (j) Arxiv (L=16) (k) PAN11 (L=16) (l) PAN25 (L=16)

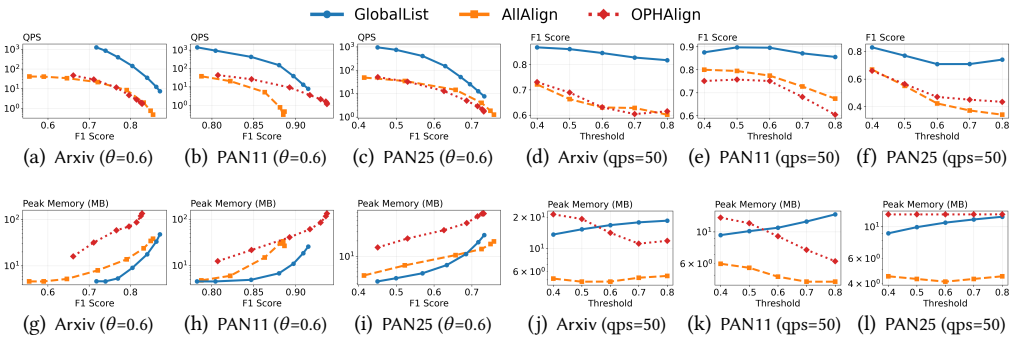
(m) Arxiv (m=8) (n) PAN11 (m=8) (o) PAN25 (m=8) (p) Arxiv (L=16) (q) PAN11 (L=16) (r) PAN25 (L=16)

Fig. 7. Sensitivity of `GlobalList`'s Efficiency to LSH Parameters. $n = 1600$ in all experiments.



(a) Arxiv (m=8) (b) PAN11 (m=8) (c) PAN25 (m=8) (d) Arxiv (L=16) (e) PAN11 (L=16) (f) PAN25 (L=16)

Fig. 8. Sensitivity of `GlobalList`'s Accuracy to LSH Parameters. $n = 100$ in all experiments.



(a) Arxiv (θ=0.6) (b) PAN11 (θ=0.6) (c) PAN25 (θ=0.6) (d) Arxiv (qps=50) (e) PAN11 (qps=50) (f) PAN25 (qps=50)

(g) Arxiv (θ=0.6) (h) PAN11 (θ=0.6) (i) PAN25 (θ=0.6) (j) Arxiv (qps=50) (k) PAN11 (qps=50) (l) PAN25 (qps=50)

Fig. 9. Comparison with baselines at Jaccard threshold $\theta = 0.6$. $n = 100$ in all experiments.

linearly with n. We observe that `GlobalList` consistently outperformed `Heap-based`, achieving a nearly constant 3× speedup across all datasets, indicating that the optimization yields stable improvements regardless of n, which aligns with the theoretical analysis.

Figures 6(d)–(f) show the results when $n = 1600$ and $L = 1$ with a varying number m of hash functions. Both methods exhibited linear growth in partition generation time with respect to m. Across all datasets, `GlobalList` was faster than `Heap-based`, and the performance gap widened as

m increases, consistent with the additional $\log^2 m$ factor in the time complexity of the `Heap-based` method. Thus, in the following experiments, we focus on `GlobalList` only.

7.2 Sensitivity Study

This section evaluates how the performance of `GlobalList` varies with the LSH parameters L and m.

Figure 7 shows the partition time, the number of tight intervals generated, and the peak memory usage of `GlobalList` when varying L and m. When varying L, we fixed $m = 8$; when varying m, we fixed $L = 16$. We observe that both the partition time and the number of tight intervals grow nearly linearly with either L or m, which aligns with the expected $O(nmL)$ time and space complexities. For example, as shown in Figure 7(b), when L increased from 4 to 64, latency increased from 13ms to 214ms. Similarly, Figure 7(j) shows that, when m increased from 2 to 32, the number of tight intervals increased from 114K to 2M. As for peak memory, we observe that it grows slightly super-linearly with increasing L or m, primarily due to a fixed startup overhead. However, as L or m becomes larger, the growth trend gradually approaches linear behavior.

Figure 8 shows the accuracy of `GlobalList` when varying L (with default value $m = 8$) and m (with default value $L = 16$) under a fixed text length $n = 100$ and Jaccard threshold $\theta = 0.6$. As L increases with m fixed, precision gradually decreases, while recall increases, leading the F1 score to first rise and then fall. This is because, as L increases, the condition that two subsequences are matched becomes looser, causing the number of FP to increase and the number of FN to decrease. Consequently, recall improves while precision declines. When L is fixed and m increases, precision increases, recall decreases, and again the F1 score first rises and then falls. This is because when m increases, the condition that two subsequences are matched becomes stricter. Consequently, the number of FP decreases, while the number of FN increases. As a result, precision improves but recall declines. Note that one can also choose optimal LSH parameters (m, L) by iterating candidate values and computing the expected F1 score, following `datasketch` [52]; we defer the details to the technical report [51].

7.3 Comparing with the State-of-the-Art

This subsection compares the performance of `GlobalList` with two state-of-the-art baselines: AllAlign and OphAlign. We fix the text length at $n = 100$ and vary the number of hash functions used, i.e., the parameters m and L for `GlobalList` and k for AllAlign and OphAlign. For each method, we perform a grid search over different parameter combinations and report the best performance obtained. The resulting points are connected to get a smooth curve that approximates the Pareto frontier of accuracy versus efficiency.

Figure 9(a)-(c) report the query throughput (QPS) versus F1 score at a fixed Jaccard threshold $\theta = 0.6$ across the three datasets. `GlobalList` consistently outperforms both baselines, achieving one to two orders of magnitude higher query throughput under the same F1 score. For example in Figure 9(a), when the F1 score was fixed at 0.8, `GlobalList` attained 176.0 QPS, while AllAlign and OphAlign achieved only 7.01 and 4.32 QPS, respectively.

Figure 9(d)-(f) present the F1 score as a function of the Jaccard threshold θ under a fixed $QPS = 50$. For each threshold, `GlobalList` consistently achieves the highest F1 score among all methods. Moreover, `GlobalList` exhibits the most stable performance as θ increases, while both AllAlign and OphAlign degrade sharply. For example, in Figure 9(d), when θ increased from 0.4 to 0.8, the F1 score of `GlobalList` changed only slightly, from 0.87 to 0.82, whereas those of AllAlign and OphAlign dropped significantly—from 0.72 to 0.60 and from 0.73 to 0.61, respectively.

Figures 9(g)-(i) report peak memory usage versus F1 score at a fixed Jaccard threshold of $\theta = 0.6$. Across all datasets, `GlobalList` consistently requires the least memory among all methods. For

example, as shown in Figure 9(g), when the F1 score was fixed at 0.8, `GlobalList` required 7.60 MB, whereas AllAlign and OphAlign required 13.35 MB and 70.34 MB, respectively.

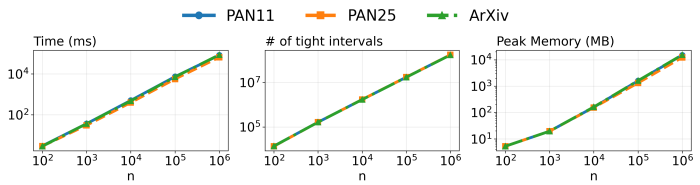
Figures 9(j)-(l) present the peak memory usage as a function of the Jaccard threshold θ under a fixed query throughput of $QPS = 50$. Overall, `GlobalList` consumes more memory than AllAlign, but remains comparable to OphAlign. As θ increases, the memory usage of `GlobalList` grows mildly, whereas that of OphAlign decreases noticeably and AllAlign remains nearly constant.

These results confirm that `GlobalList` achieves the best trade-off among accuracy, speed, and memory usage across all datasets. While we limited $n = 100$ here for accuracy verification, we further evaluate the QPS and peak memory usage of `GlobalList` against baselines on larger texts ($n = 1,000$ and $10,000$) in our technical report [51]. The results show that `GlobalList` maintains high throughput while baselines suffer from prohibitive overheads at larger n.

7.4 Scalability

In this subsection, we evaluate the scalability of `GlobalList` with respect to the text length n, which is the primary scalability bottleneck in our setting. Fixing the parameters at $m = 8, L = 8, \theta = 0.6$, we report the query latency, the number of tight intervals generated, and the peak memory usage across all three datasets. The number of similar subsequence pairs grows quadratically with n, reaching impractically large magnitudes for long texts (e.g., increasing from 1,429 at $n = 100$ to over 1.5×10^{16} at $n = 10^6$ on PAN25). Such a massive volume of pairs would dominate the peak memory usage and is rarely required in real-world scenarios. Therefore we modified the procedure to accumulate the count of reported pairs rather than storing them.

We observe that the number of tight intervals grows almost linearly with n, confirming the expected $O(nmL)$ space complexity of `GlobalList`. For example, as shown in Figure 10(b), when n increased from 100 to 10^6, the number of tight intervals increased from about 14K to 171M. Similarly, both the query latency and peak memory usage exhibit linear growth with respect to n.



(a) $m = 8, L = 8$ (b) $m = 8, L = 8$ (c) $m = 8, L = 8$

Fig. 10. Scalability of `GlobalList` across datasets.

8 Related Work

Min-Hash. Min-Hash was first introduced by Brewer as a coordinated sampling technique [5] and later adopted by Flajolet and Martin for probabilistic counting in database systems [18]. Broder et al. [8] applied Min-Hash to detect near-duplicate web pages, and Cohen [11] extended it for estimating set similarities. Given a text of n tokens, constructing a Min-Hash sketch of size k requires $O(nk)$ time. To improve efficiency, several variants have been proposed, including the bottom-k sketch [48], one-permutation hashing (OPH) [29], the fast similarity sketch [13], and C-MinHash [30]. More recent works further densify OPH [39, 41, 42] to fill empty hash bins and guarantee that each sketch contains exactly k valid hash values, thereby improving estimation stability. Beyond sketch construction, Min-Hash has also been widely adopted within the LSH framework for large-scale set similarity search and similarity joins. Representative systems include

LSHEnsemble for set containment search [54], Josie for adaptive similarity joins [53], and LAZO for high-throughput similarity indexing [17].

Locality-Sensitive Hashing. Locality-Sensitive Hashing (LSH) [2, 26] provides a general framework for similarity search, ensuring that objects close to each other under a given metric have a high probability of being mapped to the same hash bucket, while objects far away from each other have a low probability of colliding. Specialized LSH families have been developed for various distance metrics, including Hamming distance [22], Euclidean distance [14], and angular similarity [3]. Subsequent research has enhanced retrieval efficiency and adaptivity: C2LSH [21] uses dynamic collision counting; QALSH [25] introduces query-aware bucket boundaries; LSB-tree and its improved variant [46, 47] integrate LSH with B-tree indexing; Multi-Probe LSH [31] explores nearby buckets during queries; and SRS [45] achieves competitive accuracy with compact index structures. Further extensions such as ALSH [40] adapt LSH to maximum inner product search (MIPS).

Near-Duplicate Text Alignment. Most existing methods for near-duplicate text alignment are heuristic and follow the classic seed–extend–filter paradigm [4, 6, 8, 23, 24, 27, 32, 38]. Such methods depend heavily on heuristic thresholds [19] and lack theoretical guarantees. To address these issues, recently, Min-Hash-based approaches [16, 35, 36, 49] have introduced the concept of compact windows to efficiently group subsequences sharing the same Min-Hash value, avoiding explicit subsequence enumeration. Feng et al. [16] first formalized this idea and proved that when identical tokens share the same hash value, the $O(kn^2)$ MinHashes in a text of length n can be compressed into $O(kn)$ compact windows, each taking $O(1)$ space, in $O(kn \log n)$ time, where k is the sketch size. Wang et al. [49] showed that the $O(n^2)$ bottom-k sketches of all subsequences can be compressed into compact windows using $O(n \log n + nk)$ time and $O(nk^2)$ space. Subsequent work [36] improved efficiency to $O(n+k)$ space and $O(n \log n+k)$ time via one-permutation hashing (OPH) [29]. Peng et al. [35] later extended this framework to study memorization in large language models, showing that up to 10% of generated texts by LLMs contain near-duplicates in their training corpus. It also proved that Min-Hash sketches of all subsequences with length at least t can be grouped into $O(\frac{n}{t})$ compact windows on average. However, these methods primarily address the text-to-subsequence alignment problem, where a single text is compared against all subsequences of another. A straightforward way to extend them to the all-pairs setting is to enumerate all subsequences of one text and invoke the original algorithm repeatedly, but this naive approach still incurs quadratic complexity.

9 Conclusion

In this paper, we study the all-pair near-duplicate text alignment problem, which aims to identify all similar subsequences between two input texts T and S. We propose an efficient framework based on Locality-Sensitive Hashing (LSH) that employs $m \times L$ independent hash functions, organized into L hash tables of m hash functions each. We propose to group the subsequences in a text by their LSH values. We prove that a text of length n yields $O(nmL)$ such groups in expectation. We further design a constant-space representation for each group and develop an algorithm with $O((|\mathrm{T}| + |\mathrm{S}|)mL)$ expected time and space complexity for all-pair near-duplicate text alignment of two texts T and S (excluding result generation time). Experiments on large real-world datasets show order-of-magnitude speedups while maintaining comparable alignment accuracy.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. This material is based upon work supported by the National Science Foundation under Grant No. 2212629.

References

[1] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. 1990. Basic local alignment search tool. *Journal of molecular biology* 215, 3 (1990), 403–410.

[2] Alexandr Andoni and Piotr Indyk. 2006. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. In *FOCS*. 459–468.

[3] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya P. Razenshteyn, and Ludwig Schmidt. 2015. Practical and Optimal LSH for Angular Distance. In *NIPS*. 1225–1233.

[4] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. 1999. *Modern Information Retrieval*. ACM Press / Addison-Wesley. http://www.dcc.ufmg.br/irbook/

[5] K R W Brewer, L J Early, and S F Joyce. 1972. Selecting several samples from a single population. *Australian Journal of Statistics* 14, 3 (1972), 231–239.

[6] Sergey Brin, James Davis, and Hector Garcia-Molina. 1995. Copy Detection Mechanisms for Digital Documents. In *SIGMOD*. ACM Press, 398–409.

[7] Andrei Z. Broder. 1997. On the resemblance and containment of documents. In *SEQUENCES*. IEEE, 21–29.

[8] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. 1997. Syntactic Clustering of the Web. *Comput. Networks* 29, 8-13 (1997), 1157–1166. doi:10.1016/S0169-7552(97)00031-7

[9] Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramèr, and Chiyuan Zhang. 2022. Quantifying Memorization Across Neural Language Models. *CoRR* abs/2202.07646 (2022). arXiv:2202.07646 https://arxiv.org/abs/2202.07646

[10] Arman Cohan, Sergey Feldman, Iz Beltagy, Doug Downey, and Daniel S Weld. 2020. Specter: Document-level representation learning using citation-informed transformers. *arXiv preprint arXiv:2004.07180* (2020).

[11] Edith Cohen. 2016. Min-Hash Sketches.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3 ed.). MIT Press, Cambridge, MA.

[13] Søren Dahlgaard, Mathias Bæk Tejs Knudsen, and Mikkel Thorup. 2017. Fast similarity sketching. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 663–671.

[14] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG*. 253–262.

[15] Hailun Ding, Shenao Yan, Juan Zhai, and Shiqing Ma. 2021. {ELISE}: A storage efficient logging system powered by redundancy reduction and representation learning. In *30th USENIX Security Symposium (USENIX Security 21)*. 3023–3040.

[16] Weiqi Feng and Dong Deng. 2021. Allign: Aligning All-Pair Near-Duplicate Passages in Long Texts. In *SIGMOD*. ACM, 541–553. doi:10.1145/3448016.3457548

[17] Raul Castro Fernandez, Jisoo Min, Demitri Nava, and Samuel Madden. 2019. Lazo: A cardinality-based method for coupled estimation of jaccard similarity and containment. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1190–1201.

[18] Philippe Flajolet and G Nigel Martin. 1985. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences* 31, 2 (1985), 182–209.

[19] Tomás Foltýnek, Norman Meuschke, and Bela Gipp. 2020. Academic Plagiarism Detection: A Systematic Literature Review. *ACM Comput. Surv.* 52, 6 (2020), 112:1–112:42.

[20] Philip Gage. 1994. A New Algorithm for Data Compression. *C Users J.* 12, 2 (feb 1994), 23–38.

[21] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2012. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD*. 541–552.

[22] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *VLDB*. 518–529.

[23] Ossama Abdel Hamid, Behshad Behzadi, Stefan Christoph, and Monika Rauch Henzinger. 2009. Detecting the origin of text segments efficiently. In *WWW*. ACM, 61–70.

[24] Timothy C. Hoad and Justin Zobel. 2003. Methods for Identifying Versioned and Plagiarized Documents. *J. Assoc. Inf. Sci. Technol.* 54, 3 (2003), 203–215. doi:10.1002/asi.10170

[25] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-Aware Locality-Sensitive Hashing for Approximate Nearest Neighbor Search. *PVLDB* 9, 1 (2015), 1–12.

[26] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC*. 604–613.

[27] Jong Wook Kim, K. Selçuk Candan, and Jun'ichi Tatemura. 2009. Efficient overlap and content reuse detection in blogs and online news articles. In *WWW*. ACM, 81–90.

[28] Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. 2022. Deduplicating Training Data Makes Language Models Better. In *ACL*. 8424–8445.

[29] Ping Li, Art B. Owen, and Cun-Hui Zhang. 2012. One Permutation Hashing. In *NIPS*. 3122–3130.
[30] Xiaoyun Li and Ping Li. 2021. C-MinHash: Practically Reducing Two Permutations to Just One. arXiv:2109.04595 [cs.DS] https://arxiv.org/abs/2109.04595
[31] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search. In *VLDB*. 950–961.
[32] Udi Manber. 1994. Finding Similar Files in a Large File System. In *USENIX Winter 1994 Technical Conference, San Francisco, California, USA, January 17-21, 1994, Conference Proceedings*. USENIX Association, 1–10. https://www.usenix.org/conference/usenix-winter-1994-technical-conference/finding-similar-files-large-file-system
[33] PAN. 2014. PAN 2014 Text Alignment Task. https://pan.webis.de/clef14/pan14-web/text-alignment.html.
[34] PAN. 2025. PAN Datasets. https://pan.webis.de/data.html.
[35] Zhencan Peng, Zhizhi Wang, and Dong Deng. 2023. Near-Duplicate Sequence Search at Scale for Large Language Model Memorization Evaluation. *Proc. ACM Manag. Data* 1, 2 (2023), 179:1–179:18. doi:10.1145/3589324
[36] Zhencan Peng, Yuheng Zhang, and Dong Deng. 2024. Near-Duplicate Text Alignment with One Permutation Hashing. *Proc. ACM Manag. Data* 2, 4 (2024), 200:1–200:26. doi:10.1145/3677136
[37] Saul Schleimer, Daniel Shawcross Wilkerson, and Alex Aiken. 2003. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, Alon Y. Halevy, Zachary G. Ives, and AnHai Doan (Eds.). ACM, 76–85. doi:10.1145/872757.872770
[38] Jangwon Seo and W. Bruce Croft. 2008. Local text reuse detection. In *SIGIR*. ACM, 571–578.
[39] Anshumali Shrivastava. 2017. Optimal densification for fast and accurate minwise hashing. In *International Conference on Machine Learning*. PMLR, 3154–3163.
[40] Anshumali Shrivastava and Ping Li. 2014. Asymmetric LSH (ALSH) for Sublinear Time Maximum Inner Product Search (MIPS). In *NIPS*. 2321–2329.
[41] Anshumali Shrivastava and Ping Li. 2014. Densifying one permutation hashing via rotation for fast near neighbor search. In *International Conference on Machine Learning*. PMLR, 557–565.
[42] Anshumali Shrivastava and Ping Li. 2014. Improved densification of one permutation hashing. *arXiv preprint arXiv:1406.4784* (2014).
[43] Temple F Smith, Michael S Waterman, et al. 1981. Identification of common molecular subsequences. *Journal of molecular biology* 147, 1 (1981), 195–197.
[44] Benno Stein, Sven Meyer zu Eissen, and Martin Potthast. 2007. Strategies for retrieving plagiarized documents. In *SIGIR*. ACM, 825–826. doi:10.1145/1277741.1277928
[45] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. 2014. SRS: Solving c-Approximate Nearest Neighbor Queries in High Dimensional Euclidean Space with a Tiny Index. *PVLDB* 8, 1 (2014), 1–12.
[46] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2009. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*. 563–576.
[47] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2010. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Trans. Database Syst.* 35, 3 (2010), 20:1–20:46.
[48] Mikkel Thorup. 2013. Bottom-k and priority sampling, set similarity and subset sums with minimal independence. In *STOC*. ACM, 371–380. doi:10.1145/2488608.2488655
[49] Zhizhi Wang, Chaoji Zuo, and Dong Deng. 2022. TxtAlign: Efficient Near-Duplicate Text Alignment Search via Bottom-k Sketches for Plagiarism Detection. In *SIGMOD*. ACM, 1146–1159. doi:10.1145/3514221.3526178
[50] Shuo Yang, Wei-Lin Chiang, Lianmin Zheng, Joseph E. Gonzalez, and Ion Stoica. 2023. Rethinking Benchmark and Contamination for Language Models with Rephrased Samples. arXiv:2311.04850 [cs.CL] https://arxiv.org/abs/2311.04850
[51] Yuheng Zhang, Zhencan Peng, Miao Qiao, Wei Zhang, Feifei Li, and Dong Deng. 2025. Near-Duplicate Text Alignment under Weighted Jaccard Similarity. https://github.com/rutgers-db/KMINSLSH/raw/main/All-Pair_Near-Duplicate_Text_Alignment_via_LSH.pdf. Technical Report.
[52] Eric Zhu. 2025. datasketch: MinHash LSH Implementation. https://github.com/ekzhu/datasketch GitHub repository.
[53] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In *SIGMOD*. 847–864.
[54] Erkang Zhu, Fatemeh Nargesian, Ken Q Pu, and Renée J Miller. 2016. LSH ensemble: internet-scale domain search. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1185–1196.

Received October 2025; revised January 2026; accepted February 2026