Near-Duplicate Text Alignment under Weighted Jaccard Similarity

Yuheng Zhang
Rutgers University
New Brunswick, NJ, United States
yuheng.zhang@cs.rutgers.edu

Miao Qiao
University of Auckland
Auckland, New Zealand
miao.qiao@auckland.ac.nz

Zhencan Peng
Rutgers University
New Brunswick, NJ, United States
zhencan.peng@cs.rutgers.edu

Dong Deng*
Rutgers University
New Brunswick, NJ, United States
dong.deng@cs.rutgers.edu

ABSTRACT

Near-duplicate text alignment is the task of identifying all subsequences (i.e., substrings) in a collection of texts that are similar to a given query. Traditional approaches rely on seeding–extension–filtering heuristics, which lack accuracy guarantees and require many hard-to-tune parameters. More recent methods leverage min-hash techniques. They propose to group all the subsequences in each text by their min-hash and index the groups. When a query arrives, they can use the index to find all the min-hash sketches that are similar to the query's sketch and then return the corresponding subsequences as the results efficiently. Thus these methods guarantee to identify all subsequences whose estimated Jaccard similarity with the query exceed a user-provided threshold. However, these methods only support unweighted Jaccard similarity, which cannot capture token importance or frequency, limiting their effectiveness in real-world scenarios where tokens carry weights, such as TF-IDF.

In this paper, we address this limitation by supporting weighted Jaccard similarity using consistent weighted sampling. We design an algorithm `MonoActive` to group all subsequences in a text by their consistent weighted sampling. We analyze the complexity of our algorithm. For raw count term frequency (where a token's weight is proportional to its frequency in the text), we prove `MonoActive` generates $O(n + n \log f_T)$ groups (each group occupies $O(1)$ space) in expectation for a text with n tokens, where f_T is the maximum token frequency in the text. We further prove that our algorithm is optimal, meaning that any algorithm must generate $\Omega(n + n \log f_T)$ groups in expectation. Extensive experiments show that `MonoActive` outperforms the state-of-the-art by up to 4.7× in speed and reduces index size by up to 30%, with superior scalability.

PVLDB Reference Format:
Yuheng Zhang, Miao Qiao, Zhencan Peng, and Dong Deng. Near-Duplicate Text Alignment under Weighted Jaccard Similarity. PVLDB, 19(7): 1600 - 1613, 2026.
doi:10.14778/3801059.3801072

*Corresponding author.
This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 7 ISSN 2150-8097.
doi:10.14778/3801059.3801072

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/rutgers-db/WeightAlign.

1 INTRODUCTION

This paper studies the near-duplicate text alignment problem [15, 34, 35, 38, 39, 53]. Given a collection of data texts, the task takes a short query text and returns all subsequences (i.e., substrings) of the data texts that are similar to the query. Near-duplicate text alignment has become increasingly important in the era of large language models (LLMs), with key applications in test set leakage (also known as data contamination) detection [30], training data deduplication [28], and memorization analysis [9, 51]. Beyond LLMs, this problem also plays a crucial role in domains such as bioinformatics [2], log analysis [13], and plagiarism detection [36, 40].

Due to the high computational cost of near-duplicate text alignment, previous methods often adopt the seeding–extension–filtering heuristic [3, 5, 7, 23, 24, 27, 32, 36, 43, 44, 52]. However, these methods lack accuracy guarantees and often involve many hard-to-tune hyperparameters [17]. To address these limitations, recent studies have proposed to use min-hash techniques [6, 29] for near-duplicate text alignment. These approaches guarantee to retrieve all subsequences whose estimated Jaccard similarities with the query exceed a user-defined threshold [15, 34, 35, 53]. However, a key limitation is that they only support the unweighted Jaccard similarity, which treats all tokens equally—regardless of their frequency or importance (note that depending on the tokenizer, a token can be a word [54], a q-gram [18], a byte-pair encoding [19], etc.).

To highlight the issue of unweighted Jaccard similarity, consider Q = *AAAAAATTTTTTCCCCCC*, T = *AAAAAATTTTTGCCCCCC*, and S = *AATTGCC*. Intuitively, Q is much more similar to T than to S. Yet, under unweighted Jaccard similarity (with each token as a 2-gram), both Q and T, as well as Q and S yield the same similarity score: 4/7. In contrast, weighted Jaccard similarity correctly reflects the difference: when each token in a text is weighted by its frequency in the text, Q and T have a similarity of 0.8, while Q and S only score 0.2. Moreover, unweighted Jaccard similarity fails to distinguish between stop words and content words. For example, consider Q = "*I read about Einstein in a book*", T = "*I studied Einstein through a book*", S = "*I roamed about in a castle*".

Semantically, Q is clearly closer to T than to S. However, the unweighted Jaccard similarity is the same for both: 4/9 (with each token as a word). If a meaningful token weighting scheme such as TF-IDF [49] is applied, then stop words, e.g., *I, in, a, about, through*, could have near-zero weight and the other words may have near-one weight, and thereby drawing clear distinctions. Under this scheme, the weighted Jaccard similarity between Q and T is around 0.5, while that between Q and S drops to nearly zero.

Existing Methods. Existing methods propose to group, as the indexing process, the subsequences in each data text based on their min-hash values. This way, when a query arrives, they use the index to identify all the min-hash sketches that are similar to the query's min-hash sketch and then return the corresponding subsequences as the results efficiently [15, 34, 35]. A key observation made by these methods is that the nearby subsequences of a text tend to share the same min-hash (this is because appending a token to a subsequence most likely would not change the min-hash of the subsequence). Given a text with n tokens, they group the $O(n^2)$ subsequences in the text (since each subsequence is defined by a valid pair of start and end positions, the number of such valid position pairs is $\sum_{i=1}^{n}(n-i+1) = O(n^2)$ [22]) into $O(n)$ groups while representing each group with a tuple of $O(1)$ space.

Limitations of Existing Methods. In many real-world scenarios, tokens are associated with weights, such as TF-IDF (term frequency-inverse document frequency) weights [33]. Estimating weighted Jaccard similarity requires consistent weighted sampling (CWS) [25], as standard min-hash[6] applies only to the unweighted case. Existing methods do not support CWS or weighted Jaccard similarity. The only exception is `AllAlign` [15] which supports multi-set Jaccard, a special case of weighted Jaccard where each token's weight is its frequency in the text. `AllAlign` is a greedy algorithm which groups the subsequences by their "multi-set min-hash". However, it lacks a formal complexity analysis, including an upper bound on the number of groups generated and its time complexity. Due to its recursive nature, analyzing its complexity is particularly difficult.

Our Approach. In this paper, we propose an efficient algorithm `MonoActive` to group the subsequences of a text by their CWS or multi-set min-hash. We rigorously analyze the complexity for `MonoActive`. Specifically, for multi-set Jaccard similarity, we prove that `MonoActive` produces $O(n + n\log f_{\mathrm{T}})$ groups for a text with n tokens in expectation, where f_{T} is the maximum token frequency in the text. Each group occupies only $O(1)$ space. Furthermore, we prove that `MonoActive` is *worst-case optimal* by presenting a lower bound analysis on the number of groups. That is, there exists text instances for which any algorithm under the hash-based framework must generate at least $\Omega(n + n\log f_{\mathrm{T}})$ groups in expectation. We further develop an optimization that improves the time complexity of our algorithm from $O(nf_{\mathrm{T}}\log n)$ to $O(n\log n + n\log n\log f_{\mathrm{T}})$ and space complexity from $O(nf_{\mathrm{T}})$ to $O(n + n\log f_{\mathrm{T}})$. Finally, we show our algorithm can be generalized to group the subsequences in a text based on their consistent weighted samplings, as long as the weight of a token in a text is monotonically increasing with its frequency in the text and is independent of other properties of the text. For example, for logarithmic term frequency (where the weight of a token t with frequency f_t is proportional to $\log(f_t+1)$), `MonoActive` generates $O(n + n\log\log f_{\mathrm{T}})$ groups in expectation.

In summary, we make the following contributions in this paper.

- We develop `MonoActive`, the first algorithm for near-duplicate text alignment under weighted Jaccard similarity.
- We rigorously analyze the complexity of `MonoActive` and design optimizations to reduce its time and space complexities.
- For the special case of multi-set Jaccard similarity, we prove `MonoActive` is optimal, while the greedy algorithm `AllAlign` lacks theoretical guarantees.
- For multi-set Jaccard similarity, experimental results show that `MonoActive` outperforms `AllAlign` by up to 26× in index construction time, reduces index size by up to 30%, and improves query latency by up to 3×. The performance gain increases as the text length n grows, exhibiting superior scalability.

The rest of the paper is organized as follows. We introduce preliminary knowledge in Section 2 and the framework in Section 2.3. Section 4 presents our grouping algorithm and Section 5 extends it to weighted Jaccard similarity. Section 6 shows experimental results, Section 7 reviews related work, and Section 8 concludes the paper.

2 PRELIMINARIES

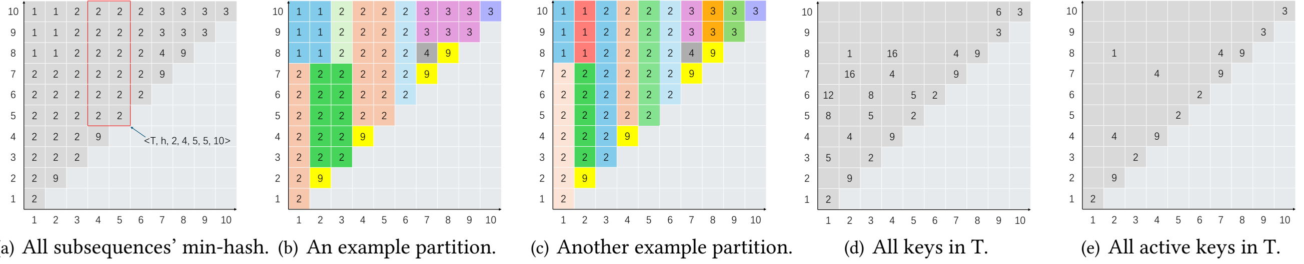
Table 1: Summary of symbols and notations.

Symbol	Meaning
D	A collection of data texts.
T, S	A text, viewed as a sequence of tokens.
Q	A query text.
n	Length of a text (number of tokens).
$f(t, T)$	Frequency of token t in text T.
f_T	Maximum token frequency in T.
$\theta \in [0, 1]$	Jaccard similarity threshold.
k	Number of independent hash functions.
h	A universal hash function.
$\mathrm{J}_{T,S}$	Multi-set Jaccard similarity between texts T and S.
$\mathrm{J}^{w}_{T,S}$	Weighted Jaccard similarity between texts T and S under a weight function w.

We first consider a special case where the tokens are weighted by their frequencies in the text (i.e., term frequency). In this case, the weighted Jaccard similarity degrades to multi-set Jaccard similarity.

2.1 Multi-Set Jaccard Similarity

We first define notations that will be used in the paper. Table 1 summarizes the symbols and notations that will be used throughout the paper. A text T is a sequence of tokens, where $\mathrm{T}[i]$ is the i-th token in the text. We define $f(t, \mathrm{T})$ as the frequency of a token t in T, i.e., the number of occurrences of t in T. A text T can be uniquely mapped to a set which exclusively contains all the distinct tokens of T as elements. For ease of presentation, we refer to T both as a text and a set. We use $|\mathrm{T}|$ to denote the length of the text T. Furthermore, we use $\mathrm{T}[i, j]$ to represent the subsequence of T from the i-th token to the j-th token (inclusive), where $1 \le i \le j \le |\mathrm{T}|$. Note all the definitions naturally extend to subsequences. Thus $\mathrm{T}[i, j]$ is both a subsequence and a token set. Given two texts T and



(a) All subsequences' min-hash. (b) An example partition. (c) Another example partition. (d) All keys in T. (e) All active keys in T.

Figure 1: A running example used throughout the paper with a text $\mathrm{T} = ABABAABBCC$ **and a hash function** h **defined as follows:** $h(A,1) = 2$, $h(A,2) = 5$, $h(A,3) = 8$, $h(A,4) = 12$, $h(B,1) = 9$, $h(B,2) = 4$, $h(B,3) = 16$, $h(B,4) = 1$, $h(C,1) = 3$, **and** $h(C,2) = 6$**.**

S, their *multi-set Jaccard similarity* is

$$\mathrm{J}_{\mathrm{T,S}} = \frac{\sum_{t \in \mathrm{T} \cup \mathrm{S}} \min(f(t,\mathrm{T}), f(t,\mathrm{S}))}{\sum_{t \in \mathrm{T} \cup \mathrm{S}} \max(f(t,\mathrm{T}), f(t,\mathrm{S}))}.$$

Example 1. *Consider two texts* $\mathrm{T} = ABBC$ *and* $\mathrm{S} = BCD$ *where each letter denotes a token. The union* $\mathrm{T} \cup \mathrm{S} = \{A, B, C, D\}$*. The token frequencies are* $f(A,\mathrm{T}) = 1$*,* $f(B,\mathrm{T}) = 2$*,* $f(C,\mathrm{T}) = 1$*,* $f(D,\mathrm{T}) = 0$*,* $f(A,\mathrm{S}) = 0$*,* $f(B,\mathrm{S}) = 1$*,* $f(C,\mathrm{S}) = 1$*, and* $f(D,\mathrm{S}) = 1$*. Therefore, their multi-set Jaccard similarity is* $\mathrm{J}_{\mathrm{T,S}} = \frac{2}{5}$*.*

2.2 Min-Hash for Multi-Set Jaccard Similarity

The multi-set Jaccard similarity of two texts can be efficiently and accurately estimated by their *min-hash* sketches. Specifically, let $h(t, x)$ be a random universal hash function that takes a token t and a positive integer x as input and outputs a non-negative integer hash value. The (multi-set) min-hash of a text T is

$$h(\mathrm{T}) = \min\{h(t,x) \mid t \in \mathrm{T}, 1 \le x \le f(t,\mathrm{T})\}. \quad (1)$$

Example 2. *Consider the text* T *and the hash function* h *from the running example in the caption of Fig. 1, we have* $h(\mathrm{T}) = h(B,4) = 1$*.*

Given a random universal hash function h and two texts T and S, the probability that they share the same min-hash is equivalent to their multi-set Jaccard similarity. Formally, we have

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

This is because there are $\Sigma_{t \in \mathrm{T} \cup \mathrm{S}} \max(f(t,\mathrm{T}), f(t,\mathrm{S}))$ unique hash values and $\Sigma_{t \in \mathrm{T} \cup \mathrm{S}} \min(f(t,\mathrm{T}), f(t,\mathrm{S}))$ common hash values in the two texts T and S. T and S share the same min-hash if and only if the smallest hash value of all unique hash values is among their common hash values. The probability of the latter is exactly $\mathrm{J}_{\mathrm{T,S}}$.

Min-Hash Sketch. With k independent random universal hash functions $h_1, \cdots, h_k$, the min-hash sketch of a text T consists of k min-hash values $h_1(\mathrm{T}), \cdots, h_k(\mathrm{T})$. The multi-set Jaccard similarity of two texts T and S can be unbiasedly estimated by

$$\hat{\mathrm{J}}_{\mathrm{T,S}} = \frac{1}{k} \sum_{i=1}^{k} \mathbf{1}\{h_i(\mathrm{T}) = h_i(\mathrm{S})\} \quad (2)$$

where $\mathbf{1}$ is an indicator function [6].

Implementation Details. The hash function $h(t, x)$ can be drawn from a family $\mathcal{H}$ of universal hash functions $h(t,x) = (a_1 t + a_2 x + b) \bmod p$ where p is a large prime and a_1, a_2, b are randomly chosen integers module p with $a_1 \neq 0, a_2 \neq 0$. We assume there is no hash collision under the universal hash function throughout the paper.

2.3 Near-Duplicate Text Alignment

Near-duplicate text alignment is formally defined as below.

Definition 1. *Given a collection of texts* D*, a query* Q*, and a similarity threshold* $\theta \in [0,1]$*, the near-duplicate text alignment problem returns all the subsequences* $\mathrm{T}[i,j]$*, where* $\mathrm{T} \in \mathrm{D}$ *and* $1 \le i \le j \le |\mathrm{T}|$*, such that* $\hat{\mathrm{J}}_{\mathrm{Q},\mathrm{T}[i,j]} \ge \theta$*.*

Example 3. *Consider a collection of two texts* $\mathrm{D} = \{\mathrm{T}, \mathrm{S}\}$*, where* $\mathrm{T} = ABBCDE$ *and* $\mathrm{S} = BCCDEF$*, and a query* $\mathrm{Q} = ACE$*. Let the similarity threshold be* $\theta = 0.5$*. Suppose the similarity estimation is accurate. The near-duplicate text alignment problem returns three subsequences* $\mathrm{T}[1,6]$*,* $\mathrm{T}[4,6]$*, and* $\mathrm{S}[3,5]$*. Note that* $\mathrm{S}[2,5] = CCDE$ *is not a result as* $\mathrm{J}_{\mathrm{Q},\mathrm{S}[2,5]} = \frac{2}{5} < \theta$*.*

3 BACKGROUND

This section provides background on the framework of near-duplicate text alignment, largely following `AllAlign`, and is included to make the paper self-contained. To improve the query performance of near-duplicate text alignment, it is natural to index the min-hash sketches of all subsequences within each text in D. This way, when a query arrives, one shall use the index to identify all the min-hash sketches that are similar to the query's sketch, then return the corresponding subsequences as the results efficiently.

The main challenge of indexing the min-hash sketches is the cost – the number of min-hash sketches in a text grows quadratically with the length of the text (as there are $O(n^2)$ subsequences in a text with n tokens). To reduce the index size, we have a key observation: the min-hash of adjacent subsequences of a text T, such as $\mathrm{T}[i,j]$ and $\mathrm{T}[i,j+1]$, tend to be the same. Thus, the min-hash of adjacent subsequences can be grouped and compactly represented. For this purpose, we define the compact window.

3.1 Compact Window and Partition

Below, we formally define the compact window [15], a structure used to represent the groups of subsequences where the subsequences in each group share the same min-hash.

Definition 2 (Compact Window [15]). *Given a text* T *and a random universal hash function h, a compact window in* T *is a tuple* $\langle \mathrm{T}, h, v, a, b, c, d \rangle$ *satisfying that*

- $1 \le a \le b \le c \le d \le |\mathrm{T}|$ *and*
- $h(\mathrm{T}[i,j]) = v$ *for every integers* $i \in [a,b]$ *and* $j \in [c,d]$.

EXAMPLE 4. *Consider the running example in Figure 1. Figure 1(a) shows the min-hash of all the subsequences of the text* T. *Specifically, the integer in the cell* (i,j) *is the min-hash of the subsequence* $\mathrm{T}[i,j]$. *For example, the min-hash of* $\mathrm{T}[1,10]$ *is shown in the cell* $(1,10)$, *which is* 1. *As one can verify, both* $\langle \mathrm{T}, h, 1, 1, 2, 8, 10\rangle$ *and* $\langle \mathrm{T}, h, 2, 4, 5, 5, 10\rangle$ *(the highlighted rectangle) are compact windows.*

Hereinafter, for two integers p, q, we abuse $[p,q]$ to denote the integer set $\{p, p+1, \cdots, q\}$. Let $[a,b] \times [c,d]$ be the set of all integer pairs (i,j) where $i \in [a,b]$ and $j \in [c,d]$. A compact window $\langle \mathrm{T}, h, v, a, b, c, d\rangle$ represents all the subsequences $\mathrm{T}[i,j]$ where $(i,j) \in [a,b] \times [c,d]$. We aim to generate a set of compact windows such that each subsequence in T is represented by one and only one compact window in the set (i.e., the set of compact windows is a lossless compression of the min-hash of all subsequences in a text). Formally, we define the concept of *partition* as below.

DEFINITION 3 (PARTITION). *Given a text* T *and a hash function* h, *a partition* $\mathcal{P}(\mathrm{T}, h)$ *is a set of compact windows* $\{W_1, W_2, \cdots, W_l\}$, *where* $l = |\mathcal{P}(\mathrm{T}, h)|$ *and* $W_x = \langle \mathrm{T}, h, v_x, a_x, b_x, c_x, d_x\rangle$, *that satisfies*

- *Disjointness: For any two compact windows* W_x *and* W_y *in* $\mathcal{P}(\mathrm{T}, h)$,

$$([a_x, b_x] \times [c_x, d_x]) \cap ([a_y, b_y] \times [c_y, d_y]) = \phi.$$

- *Coverage: The union of all compact windows in* $\mathcal{P}(\mathrm{T}, h)$ *covers all the subsequences in* T, *i.e.,*

$$\{(i,j) \mid 1 \le i \le j \le |\mathrm{T}|\} \subseteq \cup_{x=1}^{l} ([a_x, b_x] \times [c_x, d_x]).$$

EXAMPLE 5. *Consider the running example in Figure 1. Figures 1(b) and 1(c) show two example partitions with 13 and 17 compact windows respectively, as outlined by the colors. Each colored rectangle* $[a,b] \times [c,d]$ *in the figures corresponds to a compact window* $\langle \mathrm{T}, h, v, a, b, c, d\rangle$. *All the subsequences in this rectangle share the same min-hash* v.

3.2 Indexing and Query Processing

Algorithm 1 shows the pseudo-code of indexing in our framework. It takes a collection D of data texts and an integer k as input and produces k inverted indexes of compact windows. For this purpose, it first randomly selects k independent hash functions $h_1, h_2, \cdots, h_k$ (Line 1). Then, for each text T in D, and each hash function h_i, it generates a partition $\mathcal{P}(\mathrm{T}, h_i)$ (Line 4). The partition generation algorithm will be described in the next section. For each compact window $\langle \mathrm{T}, h_i, v, a, b, c, d\rangle$ in the partition, it is appended to the inverted list $I_i[v]$ (Line 6). Finally, k inverted indexes $I_1, \cdots, I_k$ are returned (Line 7).

Algorithm 2 presents the pseudo-code of query processing in our framework. The input consists of a query text Q, a multi-set Jaccard similarity threshold θ, the integer k, the same k random hash functions $h_1, h_2, \ldots, h_k$ used during indexing, and the k inverted indexes $I_1, \cdots, I_k$. It first calculates the k min-hash $v_1, \cdots, v_k$ of Q (Line 1) and then retrieves the k corresponding inverted lists $I_1[v_1], \cdots, I_k[v_k]$ from the k inverted indexes. The near-duplicate subsequences can be identified by a simple plane sweep algorithm over the compact windows in the k inverted lists (Line 2). Specifically, the plane sweep algorithm produces all the cells (x,y) where there are at least $\lceil k\theta \rceil$ compact windows $\langle \mathrm{T}, h, v, a, b, c, d\rangle$ in the

Algorithm 1: Indexing(D, k)

```
Input: D: a collection of data texts; k: an integer.
Output: I_1, I_2, ..., I_k: k inverted indexes.
1 randomly select k hash functions h_1, h_2, ..., h_k from H;
2 foreach T ∈ D do
3     foreach i ∈ [1, k] do
4         P ← PartitionGeneration(T, h_i);
5         foreach ⟨T, h_i, v, a, b, c, d⟩ ∈ P do
6             append ⟨T, h_i, v, a, b, c, d⟩ to I_i[v];
7 return I_1, I_2, ..., I_k ;
```

Algorithm 2: QueryProcessing(Q, θ, k, $h_1, \cdots, h_k, I_1, \cdots, I_k$)

```
Input: Q: a query text; θ: a similarity threshold; k: an integer;
       h_1, ..., h_k: hash functions; I_1, ..., I_k: inverted indexes.
Output: All subsequences T[x, y] in D where Ĵ_{Q,T[x,y]} ≥ θ.
1 foreach i ∈ [1, k] do v_i ← h_i(Q);
2 return PlaneSweep(k, θ, I_1[v_1], ..., I_k[v_k]);
```

k inverted lists satisfying $(x,y) \in [a,b] \times [c,d]$. This is because the estimated multi-set Jaccard similarity $\hat{\mathrm{J}}_{\mathrm{Q},\mathrm{T}[x,y]} \ge \theta$. The simple plane sweep algorithm is detailed in [34], while an optimized version is presented in [35]. Due to space constraints, we omit the details in this paper.

4 PARTITION GENERATION

Given a text T and a hash function h, there exist many possible partitions. In our framework, both the indexing and query costs scale with the partition size. This raises a natural question: how can we generate a small partition, and what is the smallest possible partition? In this section, we study the partition generation problem.

DEFINITION 4 (PARTITION GENERATION). *Given a text* T *and a random universal hash function* h, *the partition generation problem is to generate a partition* $\mathcal{P}(\mathrm{T}, h)$.

4.1 Monotonic Partitioning

In this section, we present our partition generation algorithm. To this end, we first define the hash value set of a subsequence, which contains all the hash values of the subsequence.

DEFINITION 5 (HASH VALUE SET). *Given a random universal hash function* h, *the hash value set of a subsequence* $\mathrm{T}[i,j]$ *is*

$$\mathrm{H}(\mathrm{T}[i,j], h) = \{h(t,x) \mid t \in \mathrm{T}[i,j], 1 \le x \le f(t, \mathrm{T}[i,j])\}.$$

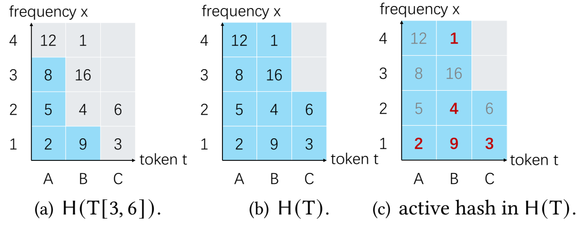
Based on Equation 1, the *min-hash of a subsequence* is the smallest hash value in the hash value set of the subsequence, i.e.,

$$h(\mathrm{T}[i,j]) = \min \mathrm{H}(\mathrm{T}[i,j], h). \tag{3}$$

We omit the hash function h in the hash value set for brevity.

EXAMPLE 6. *Consider the* T *and* h *from the running example. Figures 2(a) and 2(b) show the hash value sets* $\mathrm{H}(\mathrm{T}[3,6])$ *and* $\mathrm{H}(\mathrm{T})$. *In the figures, each integer in the highlighted cell* (t,x) *represents the hash value* $h(t,x)$ *in the corresponding hash value set. Specifically, we have* $\mathrm{H}(\mathrm{T}[3,6]) = \{2,5,8,9\}$. *Thus* $h(\mathrm{T}[3,6]) = \min \mathrm{H}(\mathrm{T}[3,6]) = 2$.

The min-hash of any subsequence of a text T must belong to the hash value set $\mathrm{H}(\mathrm{T})$ of T, as formalized below.



(a) $\mathsf{H}(\mathsf{T}[3,6])$. (b) $\mathsf{H}(\mathsf{T})$. (c) active hash in $\mathsf{H}(\mathsf{T})$.

Figure 2: Examples of hash value sets and active hash values.

Lemma 1. *$h(\mathsf{T}[i,j]) \in \mathsf{H}(\mathsf{T})$ for any $1 \le i \le j \le |\mathsf{T}|$.*

Proof. Based on Definition 5, $\mathsf{H}(\mathsf{T}[i,j]) \subseteq \mathsf{H}(\mathsf{T})$. Based on Equation 3, $h(\mathsf{T}[i,j])$=min $\mathsf{H}(\mathsf{T}[i,j])$. Thus $h(\mathsf{T}[i,j]) \in \mathsf{H}(\mathsf{T})$. □

Based on Lemma 1, the min-hash of any subsequence of T belongs to H(T). Thus, for each hash value $v = h(t,x) \in \mathsf{H}(\mathsf{T})$, we propose to generate a few disjoint compact windows to represent all the subsequences whose min-hash is v. The compact windows generated along the way must form a partition.

Specifically, we observe that if the min-hash of a subsequence $\mathsf{T}[i,j]$ comes from a token t and a frequency x such that $h(t,x) =$ min $\mathsf{H}(\mathsf{T}[i,j])$, then $x \le f(t, \mathsf{T}[i,j])$, and thus there must exist a subsequence $\mathsf{T}[p,q]$ of $\mathsf{T}[i,j]$ (i.e., $i \le p \le q \le j$) such that $\mathsf{T}[p,q]$ starts and ends with token t and has exactly x occurrences of t; otherwise $h(t,x) \notin \mathsf{H}(\mathsf{T}[i,j])$ and cannot be the min-hash of $\mathsf{T}[i,j]$ based on Lemma 1. Next, we formalize our observation.

Definition 6 (Key). *Given a text T, a key s is a pair of integers (p,q) with $1 \le p \le q \le |\mathsf{T}|$ such that $\mathsf{T}[p] = \mathsf{T}[q]$. We refer to (p,q) as the coordinates of s, define $s.x = p$ and $s.y = q$. The hash value of s is $h(\mathsf{T}[q], f(\mathsf{T}[q], \mathsf{T}[p,q]))$, abbreviated as $h(p,q,\mathsf{T})$.*

Definition 7 (Key Set). *The key set of a subsequence $\mathsf{T}[i,j]$, denoted as $\mathsf{K}(\mathsf{T}[i,j])$ is $\{(p,q) \mid [p,q] \subseteq [i,j]$ and (p,q) is a key $\}$.*

Example 7. *Consider the running example in Figure 1. Figure 1(d) shows all the keys in K(T) and their hash values, where the integer in cell (p,q) represents the hash value of key $(p,q) \in \mathsf{K}(\mathsf{T})$ (in contrast, Figure 1(a) shows the min-hash of the subsequence $\mathsf{T}[p,q]$ in cell (p,q)). In total, there are 23 keys in K(T). The x-value and y-value of the key $(1,3)$ are respectively 1 and 3. The hash value of $(1,3)$ is $h(1,3,\mathsf{T}) = h(A, f(A, \mathsf{T}[1,3])) = h(A,2) = 5$.*

The min-hash of a subsequence is exactly the smallest hash value of all its keys, as formalized below.

Lemma 2. *$h(\mathsf{T}[i,j]) = \min\{h(p,q,\mathsf{T}) \mid (p,q) \in \mathsf{K}(\mathsf{T}[i,j])\}$.*

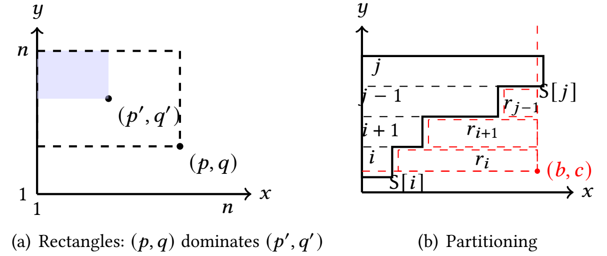
Proof. Based on Definitions 5, 6, and 7, we have $\mathsf{H}(\mathsf{T}[i,j]) = \{h(p,q,\mathsf{T}) \mid (p,q) \in \mathsf{K}(\mathsf{T}[i,j])\}$. Based on Equation 1, we have $h(\mathsf{T}[i,j]) = \min\{h(p,q,\mathsf{T}) \mid (p,q) \in \mathsf{K}(\mathsf{T}[i,j])\}$. □

Example 8. *In Figure 1(d), the key set $\mathsf{K}(\mathsf{T}[i,j])$ consists of all keys on the bottom-right of the cell (i,j), i.e., (p,q) with $p \ge i$ and $q \le j$. For example, $\mathsf{K}(\mathsf{T}[1,3]) = \{(1,1),(1,3),(2,2),(3,3)\}$. By Lemma 2, the min-hash of $\mathsf{T}[1,3]$ is $h(\mathsf{T}[1,3]) = \min\{5,2,9,2\} = 2$.*

We say a subsequence $\mathsf{T}[i,j]$ *contains* a key (p,q) if and only if $(p,q) \in \mathsf{K}(\mathsf{T}[i,j])$. Lemma 3 follows directly from Lemma 2.

Lemma 3. *A subsequence has min-hash v if and only if it contains a key with hash value v and no key with a smaller hash value.*

Lemma 3 motivates us to visit all keys in K(T) in ascending order of their hash values (breaking ties arbitrarily). When visiting a key (p,q) with hash value v, we identify all subsequences that contain (p,q) but no visited key. The min-hash of all these subsequences must be v based on Lemma 3. We then group them into disjoint compact windows. Next, we characterize these subsequences.



(a) Rectangles: (p,q) dominates (p',q') (b) Partitioning

Figure 3: Monotonic partitioning.

Definition 8 (Rectangle of a key). *Given a key (p,q) in a text T, where $n = |\mathsf{T}|$, its rectangle $\mathsf{rec}(p,q) = [1,p] \times [q,n] = \{(i,j) \mid i \in [1,p], j \in [q,n]\}$. For a set L of keys, rec(L) is the union of $\mathsf{rec}(p,q)$ with $(p,q) \in \mathsf{L}$.*

Figure 3(a) shows the rectangles of two keys (p,q) (outlined by dashed lines) and (p',q') (the shaded rectangle). By Definitions 7 and 8, a subsequence $\mathsf{T}[i,j]$ contains a key (p,q) if and only if $(i,j) \in \mathsf{rec}(p,q)$, i.e., the rectangle $\mathsf{rec}(p,q)$ precisely characterizes all subsequences that contain the key (p,q). To exclude subsequences that contain visited keys, we maintain a "skyline" to track the union of the rectangles of visited keys, as formalized below.

Definition 9 (Dominance). *Consider two keys (p,q) and (p',q'). (p,q) dominates (p',q') if and only if $[p,q] \subset [p',q']$.*

Informally, a key dominates all keys located on its top-left side, as illustrated in Figure 3(a), regardless of their hash values. Note that a key does not dominate itself based on the definition. In Example 7, the key $(1,1)$ dominates another key $(1,3)$ as $[1,1] \subset [1,3]$.

Definition 10 (Skyline). *The skyline of a set L of keys, denoted as S(L), is the subset of keys in L that are not dominated by any key in L: $\mathsf{S}(\mathsf{L}) = \{(p',q') \in \mathsf{L} \mid \nexists (p,q) \in \mathsf{L}, (p,q)$ dominates $(p',q')\}$.*

Lemma 4. *For a key set L and its skyline S(L), rec(L) = rec(S(L)).*

Proof. By definitions, a key (p,q) dominates another key (p',q') if and only if $\mathsf{rec}(p',q') \subset \mathsf{rec}(p,q)$. Furthermore, by the definition of skyline, for any key s of L not in the skyline S(L), there must be another key s' in the skyline that dominates s. Thus $\mathsf{rec}(s) \subset \mathsf{rec}(s')$. Therefore rec(L) = rec(S(L)). □

When visiting a key $(p,q) \in \mathsf{K}(\mathsf{T})$ with hash value v, let L be the set of all visited keys. Based on the discussion above, all subsequences in $\mathsf{rec}(p,q) \setminus \mathsf{rec}(\mathsf{S}(\mathsf{L}))$ must have min-hash v. As shown in Figure 3 (the red dot (b,c) is (p,q) and the indented black line is the skyline), these subsequences collectively form a "staircase

shape". We generate one compact window for "each step of the staircase" (red dashed rectangles).

In summary, our partitioning algorithm works as follows. Given a text T, we traverse all keys in K(T) in ascending order of their hash values, maintaining a dynamic skyline of the visited keys. When visiting a key (p, q) with hash value v, let L be the set of visited keys. For each key in S(L) that is dominated by (p, q), we generate a compact window. We then update the skyline by adding (p, q) if it is not dominated by any key in the skyline, and removing all keys in the skyline that are dominated by (p, q). As a result, the skyline is updated to $\mathsf{S}(\mathsf{L} \cup \{(p, q)\})$. The following sections elaborate on these details.

Example 9. *Consider the running example. There are 23 keys in* T *in total. The first 8 of them in the order of hash values are* $(2, 8)$, $(1, 1)$, $(3, 3)$, $(5, 5)$, $(6, 6)$, $(9, 9)$, $(10, 10)$, *and* $(2, 4)$. *As illustrated in Figure 4, we first visit* $(2, 8)$. *The skyline is empty. A compact window* $\langle \mathrm{T}, h, 1, 1, 2, 8, 10 \rangle$ *is generated and* $(2, 8)$ *is added to the skyline. Then we visit* $(1, 1)$. *A compact window is generated and* $(1, 1)$ *is added to the skyline. Next, we visit* $(3, 3)$. *Notice that two compact windows are generated. Since the key* $(2, 8)$ *in the skyline is dominated by* $(3, 3)$, *it is removed from the skyline, while* $(3, 3)$ *is added. Later we visit* $(5, 5)$ *and one compact window is generated. The process is repeated. When we visit the key* $(2, 4)$, *we find* $(2, 4)$ *is dominated by the key* $(3, 3)$ *on the skyline. Thus no compact window will be generated and the skyline remains the same. In the end, 13 compact windows are generated as shown in Figure 1(b).*

4.2 The Monotonic Partitioning Algorithm

Before presenting the pseudo-code of our algorithm, we first discuss how to efficiently find all keys in a skyline that are dominated by a key. This can be achieved by binary search, as formalized next.

Definition 11 (Coordinate Order). *A skyline* S *is in coordinate order if its keys are ordered lexicographically by their coordinates. We denote* $\mathsf{S}[i]$ *as the* i*-th key in* S *under this order.*

The following lemmas show a skyline is a totally ordered set.

Lemma 5. *Let* S *be a skyline of any set of keys. For any two keys* (p, q) *and* (p', q') *in* S, *if* $p \leq p'$, *then* $q \leq q'$.

Proof. Suppose that $p \leq p'$ and $q > q'$. Then (p, q) dominates (p', q'), since $p \leq p'$ and $q > q'$ implies $[p', q'] \subset [p, q]$. This contradicts the definition of the skyline, where no key is dominated by another. Therefore, we must have $q \leq q'$. □

Lemma 6. *Let* S *be the skyline of a set of keys in coordinate order. Then* $\mathsf{S}[1].x \leq \cdots \leq \mathsf{S}[l].x$ *and* $\mathsf{S}[1].y \leq \cdots \leq \mathsf{S}[l].y$ *where* $l = |\mathsf{S}|$.

Proof. Consider any two consecutive keys $\mathsf{S}[i]$ and $\mathsf{S}[i+1]$ for $1 \leq i < l$. Since the keys in S are sorted lexicographically by their coordinates, we have $\mathsf{S}[i].x \leq \mathsf{S}[i+1].x$. By Lemma 5, this implies $\mathsf{S}[i].y \leq \mathsf{S}[i+1].y$. This completes the proof. □

Lemma 7 shows that by a binary search, one can determine if a key is dominated by any key in a skyline under coordinate order.

Lemma 7. *Given a skyline* S *in coordinate order and a key* (b, c), *let* j *be the largest index such that* $\mathsf{S}[j].y \leq c$. *There exists a key in* S *that dominates* (b, c) *if and only if* j *exists and* $\mathsf{S}[j]$ *dominates* (b, c).

Proof. Suppose there exists a key $\mathsf{S}[i]$ in S that dominates (b, c), i.e., $b \leq \mathsf{S}[i].x$ and $\mathsf{S}[i].y \leq c$. Then j exists by the definition of j. Since j is the largest index with $\mathsf{S}[j].y \leq c$, we have $\mathsf{S}[i].y \leq \mathsf{S}[j].y$, which by Lemma 6, implies $\mathsf{S}[i].x \leq \mathsf{S}[j].x$. Thus, $b \leq \mathsf{S}[i].x \leq \mathsf{S}[j].x$ and $\mathsf{S}[j].y \leq c$. As $\mathsf{S}[i]$ dominates (b, c), $\mathsf{S}[j]$ dominates (b, c). On the other hand, if j exists and $\mathsf{S}[j]$ dominates (b, c), then clearly S contains a key (namely $\mathsf{S}[j]$) that dominates (b, c). □

Algorithm 4 shows the pseudo-code our monotonic partitioning algorithm. It takes a text T and a hash function h as input and produces a partition $\mathcal{P}(\mathrm{T}, h)$. It first generates all the keys in the text using the procedure GenerateKeys (Line 1). GenerateKeys (Algorithm 3) first builds an inverted index to keep track of the positions of each distinct token in T (Lines 1-4). For each inverted list (t, A) (Line 5), it enumerates every pair of positions $A[i]$ and $A[j]$ where $i \leq j$ (Lines 6-7). Then $(A[i], A[j])$ must be a key. The key, along with its hash value $h(t, j - i + 1)$, is added to the key set K (Line 8). Finally, the procedure returns the set K of all keys in T, along with their hash values (Line 9).

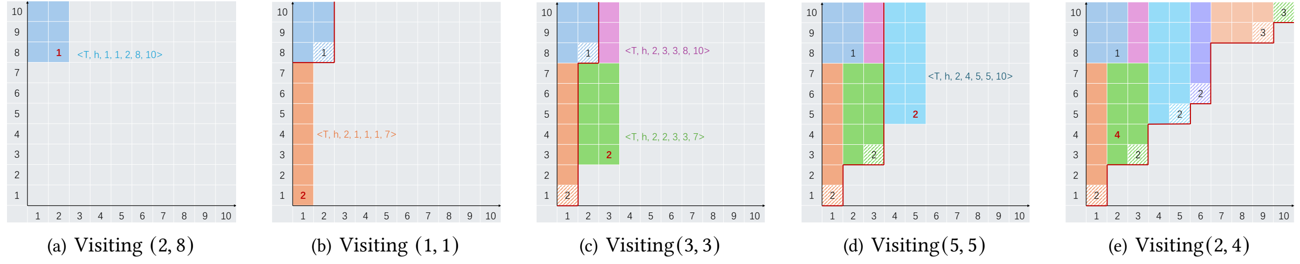
Once the key set K is generated, Algorithm 4 initializes the skyline with two guard keys $(0, 0)$ and $(n + 1, n + 1)$ in Line 2. The guard keys help simplify the algorithm by avoiding a few corner cases. The keys in S will always be kept in coordinate order using a self-balanced binary search tree. After that, we visit the keys in K in ascending order of their hash values; let (b, c) be the key currently being visited and v be its hash value (Line 3), denote by L the set of keys that have already been visited. S maintains the skyline on L upon the insertion of (b, c). Specifically, if (b, c) has been dominated by any key in S, then $\mathrm{rec}(b, c) \setminus \mathrm{rec}(\mathsf{L})$ is empty and no subsequence in $\mathrm{rec}(b, c)$ would have min-hash v. It means all subsequences in $\mathrm{rec}(b, c)$ have been represented by compact windows generated earlier. In this case, we generate no compact window: skip (b, c) and jump to the next key (Line 5).

Based on Lemma 7, we can efficiently determine whether (b, c) is dominated by any key in S via binary search. Note that if the result is $j' = 0$ (Line 4), then $\mathsf{S}[j']$ refers to the guard key $(0, 0)$, which does not dominate (b, c). This case corresponds to the scenario in Lemma 7 where no such index j exists.

When (b, c) is not dominated by any key in S, we first find all the keys in S that are dominated by (b, c). In the technical report [56], we prove that with two binary searches on S (Lines 6-7), $\mathsf{S}[i + 1]$ to $\mathsf{S}[j - 1]$ are exactly the keys in S dominated by (b, c). Lines 8-13 generate up to $j - i$ compact windows (red dashed rectangles in Figure 3(b)). As formally proved in the technical report [56], these compact windows are disjoint and cover all the subsequences $\mathrm{T}[i, j]$ where $(i, j) \in \mathrm{rec}(b, c) \setminus \mathrm{rec}(\mathsf{L})$, whose min-hash must be v (by Lemma 3 and Lemma 4). We update S by removing these dominated keys (Line 14) and inserting the new key (b, c) (Line 15).

In practice, Line 4 and Line 6 can be combined in one binary search. We use two binary searches for ease of presentation later.

Example 10. *Consider the running example. As shown in Figure 4(c), when visiting* $(b = 3, c = 3)$, *we have* $v = 2$ *and* $\mathsf{S} = \{(0, 0), (1, 1), (2, 8), (11, 11)\}$. *The binary search would find* $j' = 2$ *as* $\mathsf{S}[2].y = 1 \leq c$ *and* $\mathsf{S}[3].y = 8 > c$. *Clearly,* $\mathsf{S}[j'] = (1, 1)$ *does not dominate* $(3, 3)$. *Then the next two binary searches would find* $i = 2$ *and* $j = 4$. *Next, when* $k = 2$, *a compact window* $\langle \mathrm{T}, h, v = 2, a = 2, b = 3, c' = 3, d = 7 \rangle$ *is generated and* c' *becomes 8. When* $k = 3$,



(a) Visiting $(2, 8)$ (b) Visiting $(1, 1)$ (c) Visiting $(3, 3)$ (d) Visiting $(5, 5)$ (e) Visiting $(2, 4)$

Figure 4: An example of monotonic partitioning.

another compact window $\langle \mathrm{T}, h, 2, 3, 3, 8, 10\rangle$ *is produced. Finally, the key* $\mathrm{S}[3] = (2, 8)$ *is removed from the skyline, while* $(3, 3)$ *is added.*

Let f_T be the *maximum token frequency* in T, i.e., $f_\mathrm{T} = \max_{t\in \mathrm{T}} f(t, \mathrm{T})$.

Theorem 1. *Given a text* T *and a hash function h, Algorithm 4 generates a partition* $\mathcal{P}(\mathrm{T}, h)$.

Theorem 2. *In expectation, the partition* $\mathcal{P}(\mathrm{T}, h)$ *generated by Algorithm 4 has* $O(n + n \log f_\mathrm{T})$ *compact windows, where* $n = |\mathrm{T}|$.

Theorem 3. *When given the length and the maximum token frequency of a text, Algorithm 4, the Monotonic Partitioning algorithm, is optimal in terms of the expected partition size in the worst case.*

Due to space limitations, the proof of Theorem 1 is provided in the technical report [56], while the proof of Theorem 2 appears in Section 4.3. We emphasize that the "expectation" in Theorem 2 arises solely from the randomness of the hash functions, rather than from any distributional assumption on the input text T. In particular, the bound holds for arbitrary inputs and explicitly captures the worst-case input characteristics through the parameter f_T, the maximum token frequency in T. We also provide a matching lower bound in Theorem 3, establishing the optimality of our analysis with respect to f_T, with the proof included in the extended version for interested readers.

4.3 Active Keys and Complexity Analysis

The total number of keys generated by Algorithm 3 is bounded by $O(\sum_{t\in \mathrm{T}} f^2(t, \mathrm{T}))$, which can go up to $O(n^2)$ – when all the tokens are duplicated (i.e., $f_\mathrm{T} = n$). This leads to a running time of $O(n^2 \log n)$ and space $O(n^2)$ just for producing, sorting and storing the keys.

We observe that not all keys in $\mathrm{K}(\mathrm{T})$ need to be enumerated. Specifically, in Algorithm 4, when visiting a key $(b, c) \in \mathrm{K}(\mathrm{T})$, the key is skipped if it is dominated by a key in the skyline of visited keys (Line 5). Although it is non-trivial to determine all the skipped keys in advance, we identify a subset of them – referred to as *non-active keys* – which, as formalized below, are guaranteed to be dominated and can thus be safely skipped during key generation.

Definition 12 (Active Hash Value). *Given a text* T *and a hash function h, a hash value* $h(t, x)$ *for a token* $t \in \mathrm{T}$ *is said to be active if and only if* $h(t, x) < h(t, x')$ *for all positive integers* $x' < x$.

That is to say, a hash value $h(t, x)$ is active if and only if it is the smallest among $h(t, 1), h(t, 2), \cdots, h(t, x-1), h(t, x)$. For example, Figure 2(c) shows all the active hash values in the hash value set $\mathrm{H}(\mathrm{T})$ in Figure 2(b). The bold, red integers are active hash values, while the rest (i.e., gray ones) are non-active hash values.

Definition 13 (Active Key). *A key* (p, q) *is active if and only if its hash value* $h(p, q, \mathrm{T})$ *is active with respect to token* $\mathrm{T}[p]$ *(note that* $\mathrm{T}[p] = \mathrm{T}[q]$ *for a key). The set of active keys in a text* T *is denoted as* $\mathrm{X}(\mathrm{T})$.

For example, Figure 1(e) shows the active key set $\mathrm{X}(\mathrm{T})$ of T under the hash function h from the running example. Although T contains 23 keys in total, only 14 of them are active keys.

Note that a key is added to the skyline in Algorithm 4 if and only if it is not skipped, i.e., the condition in Line 5 evaluates to false.

Lemma 8. *No non-active key is added to the skyline in Algorithm 4.*

Proof. Let $(p, q) \in \mathrm{K}(\mathrm{T})$ be a non-active key. Then its hash value $h(t, x)$ must be non-active, where $t = \mathrm{T}[p] = \mathrm{T}[q]$ and $x = f(t, \mathrm{T}[p, q])$. Let $p = q_1 < q_2 < \cdots < q_x = q$ be the positions such that $\mathrm{T}[q_i] = t$ for all $1 \le i \le x$. Since $h(t, x)$ is non-active, by Definition 12, there exists some $1 \le i < x$ such that $h(t, i) < h(t, x)$. Then the corresponding key (p, q_i) has a smaller hash value than (p, q) and satisfies $[p, q_i] \subset [p, q]$, hence it dominates (p, q). Moreover, since $h(t, i) < h(t, x)$, (p, q_i) is visited before (p, q).

Now, consider two cases. (1) If (p, q_i) is in the skyline at the time (p, q) is visited, then (p, q) will be skipped by Lemma 7. (2) If (p, q_i) is not in the skyline, it must be pruned by a key that dominates (p, q_i). By Lemma 9 (dominance is transitive), there exists a key in the skyline that dominates (p, q_i) and thus also dominates (p, q). In this case, (p, q) will also be skipped. In either case, (p, q) will not be added to the skyline. Therefore, only active keys can enter the skyline in Line 15, and all non-active keys are skipped. □

Lemma 9. *Dominance is transitive, i.e., if* (p, q) *dominates* (p', q') *and* (p', q') *dominates* (p'', q'')*, then* (p, q) *dominates* (p'', q'').

Proof. This is because $[p, q] \subset [p', q'] \subset [p'', q'']$. □

Lemma 10. *The total number of compact windows generated by Algorithm 4 is at most* $2|\mathrm{X}(\mathrm{T})|$.

Proof. Consider Lines 9-13. The algorithm produces at most $j-i$ compact windows. Note that inserting the key (b, c) results in the removal of $j - i - 1$ other keys from the skyline, and these removed keys will not be reinserted. Thus, among the $j-i$ compact windows generated, one corresponds to the insertion of (b, c) and one to

Algorithm 3: GenerateKeys(T, h)

```
   Input: T : a text of length n; h: a universal hash function.
   Output: K: the set of keys in T.
1  map ← ∅, a map that maps a token to an array of integers;
2  K ← ∅, the set of generated keys in T;
3  foreach 1 ≤ i ≤ n do
4      append i to the array of token T[i], i.e., map[T[i]];
5  foreach token t in map (denote by A the array of map[t]) do
6      foreach i ∈ [1, |A|] do
7          foreach j ∈ [i, |A|] do
8              add (A[i], A[j], h(t, j − i + 1)) to K;
9  return K;
```

Algorithm 4: MonotonicPartitioning(T, h)

```
   Input: T: a text of length n; h: a universal hash function.
   Output: A partition P(T, h).
1  K ← GenerateKeys(h, T);
2  S ← {(0,0), (n+1, n+1)}, a skyline kept in coordinate order;
3  foreach (b, c, v) ∈ K in ascending order of the hash value v do
4      j' ← binary search the largest index such that S[j'].y ≤ c;
5      if S[j'] dominates (b, c) then continue;
6      i ← binary search the largest index such that S[i].y < c;
7      j ← binary search the smallest index such that S[j].x > b;
8      c' ← c;
9      foreach i ≤ k < j do
10         a ← S[k].x;  d ← S[k+1].y − 1 ;
           // this is to avoid the case when S[i+1].y = c
11         if a ≤ b and c' ≤ d then
12             add ⟨T, h, v, a, b, c', d⟩ to P(T, h);
13         c' ← S[k+1].y;
14     remove S[i+1], S[i+2], ··· , S[j−1] from S;
15     add (b, c) to S;
16 return P(T, h);
```

Algorithm 5: GenerateActiveKeys(T, h)

```
   // Replace Lines 5-8, Algorithm 3 with:
1  foreach entry (t, A) ∈ map do
2      minkey ← +∞;
3      foreach 1 ≤ x ≤ |A| do
4          if minkey > h(t, x) then
5              minkey ← h(t, x);
6              foreach 1 ≤ i ≤ |A| − x + 1 do
7                  add (A[i], A[i + x − 1], h(t, x)) to K;
```

the removal of each key. Thus one compact window is produced upon the insertion of a key and one compact window is produced upon the removal of a key. Moreover, by Lemma 8, only active keys can be added to the skyline. Hence, the total number of compact windows generated by Algorithm 4 is at most $2|\mathsf{X}(\mathrm{T})|$. □

LEMMA 11. *In expectation,* $|\mathsf{X}(\mathrm{T})| = O(n + n\log f_{\mathrm{T}})$*, where* $n = |\mathrm{T}|$.

PROOF. For any $t \in \mathrm{T}$ and $i \in [f(t,\mathrm{T})]$, the probability that $h(t,i)$ is active, i.e., it is smaller than all the hash values $h(t,j)$ for all $j < i$ is $\frac{1}{i}$. Therefore, the total number of active hash values among $h(t,1),\cdots,h(t,f(t,\mathrm{T}))$ in expectation is $\sum_{i\in[f(t,\mathrm{T})]}\frac{1}{i} = O(1+\ln(f(t,\mathrm{T})))$, seeing that even when $f(t,\mathrm{T}) = 1$, the summation is still 1. Moreover, there are $O(f(t,\mathrm{T}))$ keys with hash value $h(t,i)$ for any $i \in [f(t,\mathrm{T})]$. Therefore, in expectation, the total number of active keys in T, i.e., $|\mathsf{X}(\mathrm{T})|$, is $O(\sum_{t\in\mathrm{T}}(f(t,\mathrm{T})(1+\ln(f(t,\mathrm{T}))))) = O(n + \sum_{t\in\mathrm{T}}(f(t,\mathrm{T})\ln(f(t,\mathrm{T})))) = O(n + n\log f_{\mathrm{T}})$. □

Combining Lemmas 10 and 11, in expectation, the total number of compact windows in the partition $\mathcal{P}(\mathrm{T},h)$ generated by Algorithm 4 is $O(|\mathsf{X}(\mathrm{T})|) = O(n + n\log f_{\mathrm{T}})$, which proves Theorem 2.

To incorporate this optimization in our algorithm, we replace procedure GENERATEKEYS with a similar procedure GENERATEACTIVEKEYS (Algorithm 5). It generates active keys by enumerating, for each token t, every possible frequency x of t: the algorithm maintains the smallest hash value encountered in *minkey* and only when the current hash value $h(t,x) < minkey$ (Line 1-4), it produces all the keys whose hash values are $h(t,x)$ (Line 6-7) and updates *minkey* as $h(t,x)$ (Line 5).

An additional optimization is to sort the active hash values prior to generating the corresponding active keys, rather than generating and subsequently sorting all active keys. This approach is justified by two key observations: (1) active keys associated with the same hash value can be ordered arbitrarily, and (2) the number of active hash values does not exceed the total number of active keys.

THEOREM 4. *Algorithm 4 with* GENERATEACTIVEKEYS *in Line 1 has time complexity* $O(|\mathsf{X}(\mathrm{T})|\log n)$ *and space complexity* $O(|\mathsf{X}(\mathrm{T})|)$*. In expectation, the time complexity is* $O(n\log n + n\log n\log f_{\mathrm{T}})$*, and space complexity is* $O(n\log f_{\mathrm{T}} + n)$.

PROOF. Because for any $i \in [n]$, (i,i) is an active key, thus $|\mathsf{X}(\mathrm{T})| \geq n$. The size of the skyline S is $O(n)$ as each $b \in [n]$ will have at most one key in S. The binary search and each update to the skyline take $O(\log n)$. GENERATEACTIVEKEYS only produces active keys $\mathsf{X}(\mathrm{T})$. Each active key costs up to three binary searches of the skyline and two updates and corresponds to up to two compact windows. In addition, GENERATEACTIVEKEYS takes $O(|\mathsf{X}(\mathrm{T})| + n) = O(|\mathsf{X}(\mathrm{T})|)$ time; sorting the active hash values takes $O(n\log n)$ time as there are at most n active hash values. Thus the time complexity is $O(|\mathsf{X}(\mathrm{T})|\log n)$. The space complexity is $O(|\mathsf{X}(\mathrm{T})|)$. By Lemma 10, in expectation, $|\mathsf{X}(\mathrm{T})| = O(n + n\log f_{\mathrm{T}})$. Therefore, the time complexity is $O(n\log n + n\log n\log f_{\mathrm{T}})$; the space complexity is $O(n\log f_{\mathrm{T}} + n)$ in expectation. □

Due to space limitations, the proof of the correctness of this optimization is provided in the technical report [56]. Note that there are $O(nf_{\mathrm{T}})$ keys in K(T). Thus, the time and space complexities of vanilla Algorithm 4 (without the active key optimization) are respectively $O(nf_{\mathrm{T}}\log n)$ and $O(nf_{\mathrm{T}})$.

5 GENERALIZATION TO WEIGHTED JACCARD SIMILARITY

Consider a text T where each distinct token t is associated with a weight. The weight is determined by a weight function $w(t,\mathrm{T})$ where $w(t,\mathrm{T}) > 0$ for $t \in \mathrm{T}$ and $w(t,\mathrm{T}) = 0$ for $t \notin \mathrm{T}$. The weighted Jaccard similarity of two texts T and S is defined as $\mathrm{J}^{w}_{\mathrm{T,S}} = \frac{\sum_{t\in\mathrm{T}\cup\mathrm{S}}\min(w(t,\mathrm{T}),w(t,\mathrm{S}))}{\sum_{t\in\mathrm{T}\cup\mathrm{S}}\max(w(t,\mathrm{T}),w(t,\mathrm{S}))}$. To estimate weighted jaccard similarity, one can use **improved consistent weighted sampling** [26]. In

Algorithm 6: Improved Consistent Weighted Sampling [26]

```
1  Class ICWS:
2    def init(self, Σ: the vocabulary of tokens):
3      foreach token t ∈ Σ do
4        self.r_t ~ Gamma(2, 1)
5        self.c_t ~ Gamma(2, 1)
6        self.β_t ~ Uniform(0, 1)
7    def h(t: a token, weight: a positive real value):
8      y = exp(r_t(⌊ln weight / r_t + β_t⌋ − β_t));
9      a = 1/y · c_t / exp(r_t);
10     return HashValue(t, y, a);
11 Class HashValue:
12   def init(self, t, y, a): self.t = t, self.y = y, self.a = a;
13   def operator=(v1, v2): return v1.t = v2.t and v1.y = v2.y;
14   def operator<(v1, v2): return v1.a < v2.a;
```

a nutshell, one designs a hash family $\mathcal{H}$. Given a text T, for each distinct token t in T, a hash function $h \in \mathcal{H}$ takes the token t and its weight $w(t, \mathrm{T})$ as input and outputs a hash value, denoted as $h(t, w(t, \mathrm{T}))$. With hash function h, we define the *weighted min-hash* of a text T as the smallest hash value among all distinct tokens in T, denoted as $h(\mathrm{T}, w)$. Formally,

$$h(\mathrm{T}, w) = \min\{h(t, w(t, \mathrm{T})) \mid t \in \mathrm{T}\}. \tag{4}$$

The hash family $\mathcal{H}$ designed in improved consistent weighted sampling scheme guarantees that for any two texts T and S and weight function w, $\mathbf{Pr}[h(\mathrm{T}, w) = h(\mathrm{S}, w)] = \mathrm{J}^w_{\mathrm{T,S}}$.

Thus the weighted Jaccard similarity of two texts T and S can be accurately and unbiasedly estimated by k independent hash functions $h_1, \cdots, h_k$ randomly drawn from the hash family $\mathcal{H}$ as $\hat{\mathrm{J}}^w_{\mathrm{T,S}} = \frac{1}{k}\sum_{i=1}^{k} \mathbf{1}\{h_i(\mathrm{T}, w) = h_i(\mathrm{S}, w)\}$.

Implementation Details. Algorithm 6 implements the improved consistent weighted sampling scheme [26]. Drawing a random hash function $h \in \mathcal{H}$ is the same as instantiating a new object o of ICWS class (Lines 1-10). The constructor (Lines 2-6) samples three independent random variables for every token t in the vocabulary Σ from the ICWS specified distributions. The hash computation (Lines 7-10) $o.h(t, w(t, \mathrm{T}))$ of o returns the hash value based on token t and the weight of t in T. Note that an object is initialized with a random seed; the set of possible objects form the hash family.

A hash value (t, y, a) is an instance of the HashValue, where the comparators = and < are defined. For simplicity, we assume there are no hash collisions: for any two hash values (t_1, y_1, a_1) and (t_2, y_2, a_2), if $t_1 \neq t_2$, then $a_1 \neq a_2$. Comparator < provides hash values a total order, i.e., for any two hash values $v_1 = (t_1, y_1, a_1)$ and $v_2 = (t_2, y_2, a_2)$, we say $v_1 < v_2$ if and only if $a_1 < a_2$. Besides, for a token t, r_t, c_t and β_t depend solely on t. Thus the weight determines both y and a, i.e., there is a one-to-one mapping between y and a in a hash value (t, y, a). Thus, $v_1 = v_2$ iff. $t_1 = t_2$ and $y_1 = y_2$.

Consistency and Uniformity. The weighted min-hash under improved consistent weight sampling has the nice properties below:

- *Uniformity*: Denote by (t, y, a) the weighted min-hash of a text T, then (t, y) is distributed uniformly over $\cup_{t' \in \Sigma}\{t'\} \times [0, w(t', \mathrm{T})]$.
- *Consistency*: Let $v = (t, y, a)$ be the weighted min-hash of a text T. Given a text S with $w(t', \mathrm{S}) \le w(t', \mathrm{T})$ for all $t' \in \Sigma$. If $y \le w(t, \mathrm{S})$, then v must also be the weighted min-hash of S.

Table 2: A snippet of TF and IDF weights we support. Here $N = |\mathrm{D}|$ is the number of texts in the corpus D and $N_t = |\{\mathrm{S} \in \mathrm{D} \mid t \in \mathrm{S}\}|$ is the number of texts in the corpus containing t. N is a global constant while N_t is a constant local to the token t.

tf weight functions		idf weight functions	
binary:	$\mathbf{1}\{f(t, \mathrm{T}) > 0\}$	unary:	1
raw count:	$f(t, \mathrm{T})$	standard:	$\log \frac{N}{N_t}$
logarithmic:	$\log(f(t, \mathrm{T}) + 1)$	smooth:	$\log(\frac{N+N_t}{N_t}) + 1$
squared:	$f(t, \mathrm{T})^2$	probabilistic:	$\log \frac{N-N_t}{N_t}$

Assumption on the Weight Function (AoW): *Given a text* T*, we assume that for any token t in* T*, its weight is a monotonically increasing function of its frequency* $f(t, \mathrm{T})$ *and is independent of other properties of* T. With AoW, we can simplify the notation of the weight of a token t in T as $w(t, x)$ where $x = f(t, \mathrm{T})$. We can also simplify the hash function as $h_w(t, x) \doteq h(t, w(t, x))$ where x is a positive integer. The AoW, i.e., for any $1 \le x \le x'$, $w(t, x) \le w(t, x')$, is reasonable especailly when it comes to TF-IDF weights.

Term Frequency–Inverse Document Frequency (TF-IDF) [49]. TF-IDF is a widely used weighting scheme that captures both the importance of a token within a text and its rarity across a corpus D. The TF-IDF weight function is defined as $w(t, \mathrm{T}) = \mathrm{tf} \cdot \mathrm{idf}$ where tf is the TF weight and idf is the IDF weight. A few examples TF and IDF weights we support (but not limited to) are listed in Table 2.

Hash Value Set. We omit the weight function w in the notation of a hash value $h_w(t, x)$ when the context is clear and abbreviate the hash value as $h(t, x)$. We can define the hash value set of a subsequence in the same way as Definition 5.

Definition 14. *Given a random hash function h from* $\mathcal{H}$ *and a weight function w, the hash value set of a subsequence* $\mathrm{T}[i, j]$ *is*

$$\mathrm{H}(\mathrm{T}[i, j], h) = \{h(t, x) \mid t \in \mathrm{T}[i, j], 1 \le x \le f(t, \mathrm{T}[i, j])\}.$$

Next we show that, under the assumption of AoW, the hash values under consistent weighted sampling have the property below.

Lemma 12. *Given a random hash function h from* $\mathcal{H}$ *and a weight function w under the assumption of* AoW*, we have* $h(t, 1) \ge h(t, 2) \ge \cdots \ge h(t, x)$ *for any token t in any text* T *where* $x = f(t, \mathrm{T})$.

Proof. Under the assumption of AoW, we have $w(t, 1) \le w(t, 2) \le \cdots \le w(t, x)$. Now consider texts $\mathrm{T}_1, \mathrm{T}_2, \cdots, \mathrm{T}_x$ where T_i consists of exactly i copies of token t. Then T_i has only one hash value $h(t, i)$, which must be its weighted min-hash by Equation 4. Consider any $j > i$. Let $h(t, i) = (t, y, a)$ and $h(t, j) = (t, y', a')$, which are the weighted min-hash of T_i and T_j. By uniformity, y distributes uniformly over $[0, w(t, i)]$ and so does y' over $[0, w(t, j)]$. By consistency, because $w(t, i) \le w(t, j)$, if $y' \le w(t, i)$, we have $h(t, i) = h(t, j)$. Otherwise, i.e., $y' > w(t, i)$, because $y \le w(t, i)$, we have $y < y'$. Under the same t, i.e., r_t, c_t, β_t are fixed, we have $a \propto \frac{1}{y}$ (Line 9, Algorithm 6). Thus we have $a > a'$, which indicates $h(t, i) > h(t, j)$. In both cases, we have $h(t, i) \ge h(t, i)$. Thus $h(t, 1) \ge h(t, 2) \ge \cdots \ge h(t, x)$ for any t and T. ◻

By Lemma 12 and Equation 4, we have

$$h(\mathrm{T}[i, j]) = \min \mathrm{H}(\mathrm{T}[i, j], h). \tag{5}$$

Theorem 5. *Given a text* T *and a hash function* h *with weight* w *under AoW, Algorithm 4 embedded with Algorithm 6 generates a partition* $\mathcal{P}(\mathrm{T}, h)$.

Proof. By Definition 14 and Lemma 12, the weighted setting induce the hash value set structure as the unweighted case. In particular, Equation 5 plays the same role as Equation 3, i.e., for any subsequence $\mathrm{T}[i, j]$, its hash value equals the minimum over a finite, totally ordered set of candidate hash values. The remainder of the correctness proof of Algorithm 4 only relies on comparisons between hash values, and does not depend on how hash values are generated. Therefore, replacing the unweighted hash function $h(t, x)$ with the weighted hash function under AoW $h_w(t, x)$ leaves the definitions, lemmas and theorems unchanged. □

Lemma 13. *In expectation,* $|\mathrm{X}(\mathrm{T})|$ *is* $O(n)$ *for binary TF,* $O(n + n \log f_{\mathrm{T}})$ *for raw count TF,* $O(n + \log\log f_{\mathrm{T}})$ *for logarithmic TF, and* $O(n+n \log f_{\mathrm{T}})$ *for squared TF, each combined with any IDF in Table 2.*

Proof. Given a text with n tokens and a weight function w. Consider a token $t \in \mathrm{T}$. Let $x = f(t, \mathrm{T})$. Clearly, $h(t, 1)$ must be an active hash value. For any $i \in [2, x]$, based on the proof of Lemma 12, $h(t, i)$ is active if and only if $h(t, i) < h(t, i-1)$. Let $h(t, i) = (t, y_i, a_i)$ and $h(t, i-1) = (t, y_{i-1}, a_{i-1})$. $h(t, i) < h(t, i-1)$ if and only if $y_i \in (w(t, i-1), w(t, i)]$. By uniformity, y_i distributed uniformly over $[0, w(t, i)]$. Thus the probability that $h(t, i)$ is active is $\frac{w(t,i)-w(t,i-1)}{w(t,i)}$. There are $x - i + 1 = O(x)$ keys corresponding to the hash value $h(t, i)$ (imagine a sliding window of size i over a string of length x). Thus in expectation, the number of active keys in T is $\mathbb{E}[\mathrm{X}(\mathrm{T})] = O(\sum_{t\in \mathrm{T}} f(t, \mathrm{T}) \sum_{i=1}^{f(t,\mathrm{T})} \frac{w(t,i)-w(t,i-1)}{w(t,i)})$. Note that $w(t, 0) = 0$. Clearly, for $x \geq 1$, when $w(t, x) = x \cdot \mathsf{IDF}$, we have $\mathbb{E}[|\mathrm{X}(\mathrm{T})|] = O(n + n \log f_{\mathrm{T}})$. When $w(t, x) = 1 \cdot \mathsf{IDF}$, $\mathbb{E}[|\mathrm{X}(\mathrm{T})|] = O(n)$. For $w(t, x) = \log(x + 1) \cdot \mathsf{IDF}$, $\mathbb{E}[|\mathrm{X}(\mathrm{T})|] = O(n+n \log\log f_{\mathrm{T}})$ based on taylor expansion. For $w(t, x) = x^2 \cdot \mathsf{IDF}$, $\mathbb{E}[|\mathrm{X}(\mathrm{T})|] = O(n + n \log f_{\mathrm{T}})$. □

A caveat is that, instead of breaking ties arbitrarily for keys with the same hash value when visiting the key set of a text, we need to order the keys firstly by their hash values and secondly by their frequencies. That is to say for two keys (p, q) and (p', q') with the same hash value, if $f(\mathrm{T}[p], \mathrm{T}[p, q]) > f(\mathrm{T}[p'], \mathrm{T}[p', q'])$, we should visit (p, q) before (p', q'). If $f(\mathrm{T}[p], \mathrm{T}[p, q]) = f(\mathrm{T}[p'], \mathrm{T}[p', q'])$, we can break tie arbitrarily.

6 EXPERIMENT

Due to space constraints, we report representative results in this paper. Complete results and additional discussions (including query latency and effectiveness) are provided in the technical report [56].

Environment. We implemented our proposed method `MonoAll`, `MonoActive` and the `AllAlign` baseline in C++. For the `SeedExtension` baseline, we used the open-source implementation in python [1]. All C++ programs were compiled with GCC 11.4.0 and optimized using the `-O3` flag. All experiments were conducted on a machine equipped with an Intel Xeon Gold 6212 CPU @ 2.40GHz and 1 TB of memory, running Ubuntu 18.04.5.

[1]https://github.com/CubasMike/plagiarism_detection_pan2015

Table 3: Datasets statistics and the number of queries.

Dataset	# Texts	Avg. Text Length	# Queries
PAN	11,093	57,908.60	10
OWT	8,013,769	1,127.06	10
NEWS	2,688,878	661.31	10

Datasets. We conducted experiments on three datasets, PAN [37], OWT [20], and NEWS. All datasets were tokenized using the GPT-2 byte-pair encoding (BPE) tokenizer [41], with a vocabulary size of 50,257. PAN [2] is a benchmark for external plagiarism detection [37]. Each text in PAN is a book. The dataset contains 11,093 texts and 642,380,109 tokens after tokenization. OWT consists of 8 million web texts highly ranked on Reddit. Each document corresponds to a web page. NEWS is a large-scale news corpus collected from publicly available online news sources. Each document corresponds to a news article. Table 3 summarizes the dataset sizes, the average text lengths and the number of queries.

Parameters and Settings. There are three parameters, the text length n, the maximum token frequency f_{T}, and the min-hash sketch size k (see Section 2.2). We report the compact window generation time (i.e., partition time) and the number of compact windows generated (i.e., partition size) per text. To avoid text-dependent bias, for each configuration of parameters, we randomly choose 10 texts under the configuration and report the average partition time/size.

To fix the text length n, texts with more than n tokens were truncated to their first n tokens. Texts with fewer than n tokens were concatenated until they reached or exceeded n tokens, after which they were truncated to the first n tokens as a single text. This preserves token distribution without introducing artificial tokens.

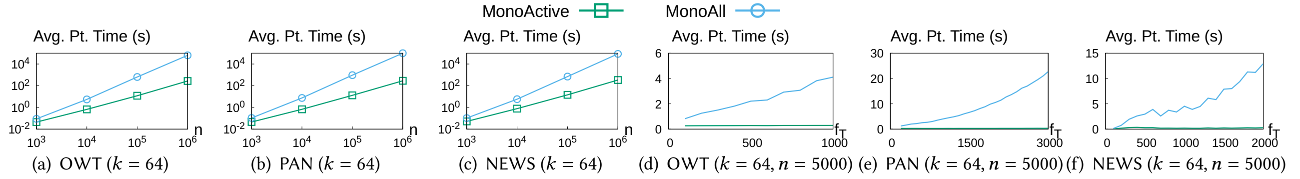
To evaluate the impact of the maximum token frequency f_{T}, we fixed the text length n and used the first 100,000 texts in OWTand NEWS, as well as the first 10,000 texts in PAN. These texts were grouped based on their maximum token frequencies using intervals of 100. For example, texts with f_{T} in the range $[1, 100]$ formed one group, while those in $[101, 200]$ formed another.

`MonoAll`, `MonoActive` and `AllAlign` all exhibit linear runtime growth with k in our experiments because the partition generation process is independent across the k universal hash functions. Thus, we skip evaluating the impact of the parameter k.

Compared Methods. We consider 4 methods in our experiment.

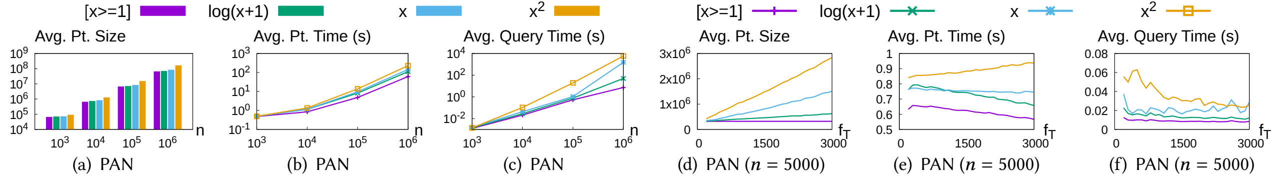
- `MonoAll`: Our vanilla monotonic partitioning, Algorithm 4.
- `MonoActive`: Our monotonic partitioning with active hash optimization (Section 4.3). It calls Algorithm 5 to generate keys.
- `AllAlign`: A greedy partitioning algorithm for multi-set Jaccard similarity [15]. `AllAlign` generates compact windows in recursion. In each iteration, it takes a rectangle of the shape $[a, b] \times [a, c]$ as input and partitions all the subsequences $\mathrm{T}[i, j]$ in this rectangle into a few compact windows and one or more smaller rectangles of that shape. The smaller rectangles are recursively partitioned until no rectangles left. At the beginning, the input rectangle is $[1, n] \times [1, n]$. However, `AllAlign` is greedy and its time and space complexities are unknown.

[2]https://pan.webis.de/data.html



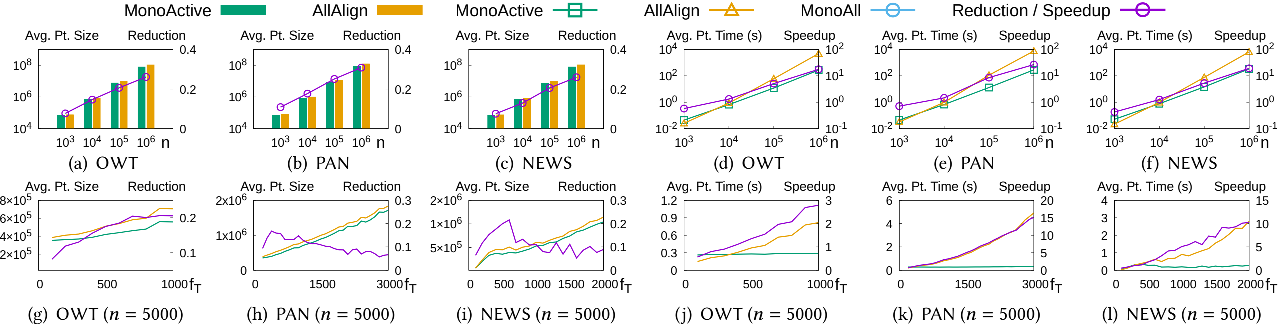
(a) OWT ($k = 64$) (b) PAN ($k = 64$) (c) NEWS ($k = 64$) (d) OWT ($k = 64, n = 5000$) (e) PAN ($k = 64, n = 5000$) (f) NEWS ($k = 64, n = 5000$)

Figure 5: Evaluating the active hash optimization. k: sketch size, n: text length, f_T: maximum token frequency.



(a) PAN (b) PAN (c) PAN (d) PAN ($n = 5000$) (e) PAN ($n = 5000$) (f) PAN ($n = 5000$)

Figure 6: Evaluating weighted Jaccard similarity under various weight functions. $k = 64$ in all experiments.



(a) OWT (b) PAN (c) NEWS (d) OWT (e) PAN (f) NEWS (g) OWT ($n = 5000$) (h) PAN ($n = 5000$) (i) NEWS ($n = 5000$) (j) OWT ($n = 5000$) (k) PAN ($n = 5000$) (l) NEWS ($n = 5000$)

Figure 7: Comparing with the state-of-the-art for multi-set Jaccard similarity. $k = 64$ in all experiments.

- `SeedExtension`: A widely used approach in plagiarism detection. It enumerates sentence pairs as candidate seeds based on a similarity threshold, and applies multiple extension and filtering stages to generate final alignments. We use the open-source implementation with the recommended parameter settings.

6.1 Evaluating Active Hash Optimization

This section evaluates our two methods `MonoActive` and `MonoAll` on three datasets OWT, PAN, and NEWS. Since the optimization in `MonoActive` does not change the compact windows, the partition size of `MonoActive` and `MonoAll` are the same, we only consider the partition time. Recall that in Theorem 4, the time complexities of `MonoActive` and `MonoAll` are respectively $O(n \log n \log f_T)$ and $O(n f_T \log n)$ when the maximum frequency $f_T > 1$.

Figures 5(a)-5(c) show that the partition time of two methods increases with the text length. `MonoActive` consistently outperforms `MonoAll` because `MonoActive` avoids unnecessary computations for non-active keys. The ratio between the partition time of `MonoAll` and that of `MonoActive` echoes our complexity analysis.

Figures 5(d)-5(f) show that with $k = 64$ and $n = 5000$, as the maximum frequency f_T increases, the runtime of `MonoActive` remained nearly constant whereas that of `MonoAll` grew almost linearly with f_T. For example, on PAN, when $f_T \in [901, 1000]$, $f_T \in [1901, 2000]$, and $f_T \in [2901, 3000]$, `MonoActive` took 0.27, 0.29, 0.33 seconds for partition generation on average, while `MonoAll` took 3.98, 10.56, 22.59 seconds. When f_T increased around threefold, the runtime of `MonoActive` and `MonoAll` respectively increased 1.2× and 5.7×. This observation is consistent with our complexity analysis.

6.2 Evaluating Weighted Jaccard Similarity

This subsection evaluates `MonoActive` under weighted Jaccard similarity. We use four TF weights for $w(t, x)$: binary $[x \geq 1]$, logarithmic $\log(x + 1)$, raw count x, and squared x^2 (all with unary IDF).

Figure 6(a) shows that the partition size increases approximately linearly with text length n under all four weights. Squared TF yields the largest partitions, raw count is second, and binary TF is smallest. Figure 6(b) shows the same ordering for partitioning time. These trends are consistent with Lemma 13: binary scales as $O(n)$, logarithmic as $O(n + n \log \log f_T)$, and raw-count/squared as $O(n + n \log f_T)$ (with a larger constant for squared TF).

Figure 6(d) shows that as f_T grows, the partition size is nearly constant for binary TF, increases mildly for logarithmic TF, and increases sublinearly for raw-count and squared TF. These results are consistent with our theoretical analysis. Figure 6(e) shows that partitioning time remains stable or slightly decreases, because a substantial part of `MonoActive`'s cost is sorting active hash values, and the number of distinct tokens decreases as f_T increases.

6.3 Comparing with State-of-the-Art

This section compares our method `MonoActive` with `AllAlign`, the state-of-the-art method for multi-set Jaccard similarity.

Figures 7(a)-7(c) compare partition size when varying the text length n from 10^3 to 10^6 on OWT, PAN, and NEWS. The reduction ratio, defined as $1 - \frac{\texttt{MonoActive}}{\texttt{AllAlign}}$, is overlaid as a purple line. `MonoActive` consistently generates fewer compact windows than `AllAlign` (up to 30.82%), and the gap widens as n increases. For example, the reduction grows from 10.86% to 30.82% on PAN and from 7.69% to 26.08% on OWT. This trend matches our analysis: `AllAlign`'s recursive partitioning introduces early boundaries that may split one compact window into multiple windows.

Figures 7(d)-7(f) compare partition time under the same settings. `MonoActive` is consistently faster than `AllAlign`, and the speedup increases with n. On PAN with $n = 10^6$, `AllAlign` takes 7,581s while `MonoActive` takes 284s (26.7× speedup). Similar trends are observed on OWT and NEWS.

We also evaluate the impact of the maximum token frequency f_T. Figures 7(g)–7(l) report partition size and partition time with fixed text length $n = 5000$, and the relative improvements are shown by the same purple line. As f_T increases, `MonoActive` remains stable while `AllAlign` becomes significantly slower. For example, on PAN, the speedup increases from 1.41 to 16.26 when f_T grows from 600 to 3000. The partition-size gap first increases and then decreases as f_T approaches n, where both methods approach worst-case behavior.

Scalability. As shown in Figures 7(a)-7(i), the performance gain of `MonoActive` over `AllAlign` increased as the text length n grew in terms of partition size, partition generation time, and query latency. Thus `MonoActive` scales better than `AllAlign`. This is attributed to the complexity guarantees of our algorithm.

Effectiveness. We omit effectiveness analysis in the main paper.

7 RELATED WORK

Near-Duplicate Text Alignment. Most of existing methods for near-duplicate text alignment rely on rule-based heuristics and the "seeding-extension-filtering" framework [3, 5, 7, 23, 24, 27, 32, 44]. They first find "seed matches" between the data texts and the query. Various kinds of seeds have been proposed, such as fingerprints [36], super-shingles [8], 0 mod p [32], fixed-length windows [52], q-grams [43], and sentences [42]. Then they extend the seed matches as far as possible to form candidates. Finally, candidates failing predefined criteria (e.g., length or overlap thresholds) are filtered. However, these heuristics are highly sensitive to the hard-to-tune hyper-parameters [17] such as the granularity of the seeds and various kinds of thresholds [1, 42]. They also lack accuracy guarantees.

A few recent works propose to use the min-hash techniques [6] for near-duplicate text alignment [15, 34, 35, 53]. They introduce the concept of "compact windows" to group nearby subsequences sharing the same min-hash. It has been shown that when duplicate tokens have the same hash value, the $O(kn^2)$ min-hashes in a text with n tokens can be compressed in compact windows using $O(kn)$ space and $O(kn \log n)$ time, where k is the sketch size [15]. Along this line, an algorithm is developed to group the $O(n^2)$ bottom-k sketches in a text with n tokens into compact windows using $O(nk^2)$ space and $O(n \log n + nk)$ time [53]. To further reduce the space cost, another algorithm is designed to group the $O(kn^2)$ one-permutation hashing [29] min-hash sketches into compact windows using $O(n+k)$ space and $O(n \log n + k)$ time [35]. These algorithms are used to evaluate the memorization behaviour in large language models (LLMs), revealing that up to 10% of texts generated by GPT-2 [41] had near-duplicates in its training data. It also shows that the min-hash sketches of all subsequences no shorter than t tokens in a text with n tokens can be grouped into $O(\frac{n}{t})$ compact windows on average [34]. However, none of these works can deal with weighted min-hash (i.e., consistent weighted sampling [26]).

Min-Hash Sketch. Min-hash was originally introduced in statistics for coordinated sampling [4] and later adapted for database applications by Flajolet and Martin [16]. Broder [7] employed the min-hash sketch to detect near-duplicate web pages. Variants of the min-hash sketch have been proposed to improve the sketching time of the classic min-hash sketch, including the bottom-k sketch [50], one-permutation hashing (OPH) [29], and fast similarity sketch [12]. The number of min-hashes generated by one-permutation hashing is not fixed. A few works aim to address this issue [46–48].

Weighted Jaccard Similarity Estimation. Many techniques have been proposed to estimate the weighted Jaccard similarity [14, 26, 31, 55]. Specifically, [21] extends the classic min-hash to estimate the multi-set Jaccard similarity. A method dealing with integer weights is mentioned in [10]. It was extended by [11] to support a more general weight function. Consistent weighted sampling (CWS) is first proposed in [31] to estimate weighted Jaccard similarity. Ioffe proposes improved consistent weighted sampling, which simplified CWS and guarantees worst-case constant time for each non-zero weight [26]. Shrivastava proposes to use rejected sampling to estimate the weigthed Jaccard Similarity [45].

8 CONCLUSION

In conclusion, this paper extends near-duplicate text alignment to support weighted Jaccard similarity by leveraging consistent weighted sampling. We introduce `MonoActive`, an efficient and theoretically optimal algorithm for grouping subsequences based on their consistent weighted samplings. Our analysis establishes tight bounds on the number of groups generated, and our experiments demonstrate substantial improvements over state-of-the-art methods in both index time and index size. These results highlight the practicality and scalability of our approach for real-world text alignment tasks where token weights matter.

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] Eneko Agirre, Carmen Banea, Daniel M. Cer, Mona T. Diab, Aitor Gonzalez-Agirre, Rada Mihalcea, German Rigau, and Janyce Wiebe. 2016. SemEval-2016 Task 1: Semantic Textual Similarity, Monolingual and Cross-Lingual Evaluation. In *SEMEVAL*. The Association for Computer Linguistics, 497–511.

[2] 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.

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

[4] 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.

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

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

[7] 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. https://doi.org/10.1016/S0169-7552(97)00031-7

[8] Andrei Z Broder, Steven C Glassman, Mark S Manasse, and Geoffrey Zweig. 1997. Syntactic clustering of the web. *Computer networks and ISDN systems* 29, 8-13 (1997), 1157–1166.

[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] Moses S Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. 380–388.

[11] Ondrej Chum, James Philbin, Andrew Zisserman, et al. 2008. Near duplicate image detection: Min-hash and TF-IDF weighting.. In *Bmvc*, Vol. 810. 812–815.

[12] 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.

[13] Hailun Ding, Juan Zhai, Dong Deng, and Shiqing Ma. 2023. The Case for Learned Provenance Graph Storage Systems. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 3277–3294. https://www.usenix.org/conference/usenixsecurity23/presentation/ding-hailun-provenance

[14] Otmar Ertl. 2018. Bagminhash-minwise hashing algorithm for weighted sets. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1368–1377.

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

[16] 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.

[17] 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.

[18] Alex Franz and Thorsten Brants. 2006. All our n-gram are belong to you. *Google Machine Translation Team* 20 (2006).

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

[20] Aaron Gokaslan and Vanya Cohen. 2019. OpenWebText Corpus. http://Skylion007.github.io/OpenWebTextCorpus.

[21] Sreenivas Gollapudi and Rina Panigrahy. 2006. Exploiting asymmetry in hierarchical topic extraction. In *Proceedings of the 15th ACM international conference on Information and knowledge management*. 475–482.

[22] Dan Gusfield. 1997. Algorithms on stings, trees, and sequences: Computer science and computational biology. *Acm Sigact News* 28, 4 (1997), 41–60.

[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. https://doi.org/10.1002/asi.10170

[25] Sergey Ioffe. 2010. Improved Consistent Sampling, Weighted Minhash and L1 Sketching. In *ICDM*. IEEE Computer Society, 246–255.

[26] Sergey Ioffe. 2010. Improved consistent sampling, weighted minhash and l1 sketching. In *2010 IEEE international conference on data mining*. IEEE, 246–255.

[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] Inbal Magar and Roy Schwartz. 2022. Data Contamination: From Memorization to Exploitation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*. Association for Computational Linguistics, 157–165. https://doi.org/10.18653/V1/2022.ACL-SHORT.18

[31] Mark Manasse, Frank McSherry, and Kunal Talwar. 2010. Consistent weighted sampling. *Unpublished technical report) http://research. microsoft. com/en-us/people/manasse* 2 (2010).

[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] Gabriella Pasi and Gloria Bordogna. 2023. Introduction to the Minitrack on Intelligent Information Access and Retrieval. In *56th Hawaii International Conference on System Sciences, HICSS 2023, Maui, Hawaii, USA, January 3-6, 2023*, Tung X. Bui (Ed.). ScholarSpace, 4169–4170. https://hdl.handle.net/10125/103140

[34] 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. https://doi.org/10.1145/3589324

[35] 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. https://doi.org/10.1145/3677136

[36] Martin Potthast, Alberto Barrón-Cedeño, Andreas Eiselt, Benno Stein, and Paolo Rosso. 2010. Overview of the 2nd International Competition on Plagiarism Detection. In *CLEF 2010 LABs and Workshops, Notebook Papers (CEUR Workshop Proceedings)*, Vol. 1176. CEUR-WS.org.

[37] Martin Potthast, Andreas Eiselt, Alberto Barrón-Cedeño, Benno Stein, and Paolo Rosso. 2011. Overview of the 3rd International Competition on Plagiarism Detection. In *CLEF 2011 Labs and Workshop, Notebook Papers (CEUR Workshop Proceedings)*, Vol. 1177. CEUR-WS.org.

[38] Martin Potthast, Tim Gollub, Matthias Hagen, Johannes Kiesel, Maximilian Michel, Arnd Oberländer, Martin Tippmann, Alberto Barrón-Cedeño, Parth Gupta, Paolo Rosso, and Benno Stein. 2012. Overview of the 4th International Competition on Plagiarism Detection. In *CLEF 2012 Evaluation Labs and Workshop (CEUR Workshop Proceedings)*, Vol. 1178. CEUR-WS.org.

[39] Martin Potthast, Matthias Hagen, Anna Beyer, Matthias Busse, Martin Tippmann, Paolo Rosso, and Benno Stein. 2014. Overview of the 6th International Competition on Plagiarism Detection. In *Working Notes for CLEF 2014 Conference (CEUR Workshop Proceedings)*, Vol. 1180. CEUR-WS.org, 845–876.

[40] Martin Potthast, Matthias Hagen, Tim Gollub, Martin Tippmann, Johannes Kiesel, Paolo Rosso, Efstathios Stamatatos, and Benno Stein. 2013. Overview of the 5th International Competition on Plagiarism Detection. In *Working Notes for CLEF 2013 Conference (CEUR Workshop Proceedings)*, Vol. 1179. CEUR-WS.org.

[41] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[42] Miguel A. Sanchez-Perez, Alexander F. Gelbukh, and Grigori Sidorov. 2015. Adaptive Algorithm for Plagiarism Detection: The Best-Performing Approach at PAN 2014 Text Alignment Competition. In *6th International Conference of the CLEF Association (Lecture Notes in Computer Science)*, Vol. 9283. Springer, 402–413.

[43] Saul Schleimer, Daniel Shawcross Wilkerson, and Alexander Aiken. 2003. Winnowing: Local Algorithms for Document Fingerprinting. In *SIGMOD*. ACM, 76–85.

[44] Jangwon Seo and W. Bruce Croft. 2008. Local text reuse detection. In *SIGIR*. ACM, 571–578.

[45] Anshumali Shrivastava. 2016. Simple and efficient weighted minwise hashing. *Advances in Neural Information Processing Systems* 29 (2016).

[46] Anshumali Shrivastava. 2017. Optimal densification for fast and accurate minwise hashing. In *International Conference on Machine Learning*. PMLR, 3154–3163.

[47] 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.

[48] Anshumali Shrivastava and Ping Li. 2014. Improved densification of one permutation hashing. *arXiv preprint arXiv:1406.4784* (2014).

[49] Amit Singhal et al. 2001. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.* 24, 4 (2001), 35–43.

[50] Mikkel Thorup. 2013. Bottom-k and priority sampling, set similarity and subset sums with minimal independence. In *STOC*. ACM, 371–380. https://doi.org/10.1145/2488608.2488655

[51] Thuy-Trang Vu, Xuanli He, Gholamreza Haffari, and Ehsan Shareghi. 2023. Koala: An Index for Quantifying Overlaps with Pre-training Corpora. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023 - System Demonstrations, Singapore, December 6-10, 2023*, Yansong Feng and Els Lefever (Eds.). Association for Computational Linguistics, 90–98. https://doi.org/10.18653/V1/2023.EMNLP-DEMO.7

[52] Pei Wang, Chuan Xiao, Jianbin Qin, Wei Wang, Xiaoyang Zhang, and Yoshiharu Ishikawa. 2016. Local Similarity Search for Unstructured Text. In *SIGMOD*. ACM, 1991–2005.

[53] 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. https://doi.org/10.1145/3514221.3526178

[54] Jonathan J Webster and Chunyu Kit. 1992. Tokenization as the initial phase in NLP. In *COLING 1992 volume 4: The 14th international conference on computational linguistics*.

[55] Wei Wu, Bin Li, Ling Chen, Chengqi Zhang, and S Yu Philip. 2018. Improved consistent weighted sampling revisited. *IEEE Transactions on Knowledge and Data Engineering* 31, 12 (2018), 2332–2345.

[56] Yuheng Zhang, Miao Qiao, Zhencan Peng, and Dong Deng. 2025. Near-Duplicate Text Alignment under Weighted Jaccard Similarity. https://github.com/rutgers-db/WeightAlign/raw/main/Near_Duplicate_Text_Alignment_under_Weighted_Jaccard_Similarity.pdf. Technical Report.