Near-Duplicate Text Alignment with One Permutation Hashing

ZHENCAN PENG, Rutgers University, USA
YUHENG ZHANG, Rutgers University, USA
DONG DENG*, Rutgers University, USA

This paper studies the near-duplicate text alignment problem under the constraint of Jaccard similarity. Specifically, given a collection of long texts and a short query text, this problem finds all the *subsequences* in each text whose Jaccard similarities to the query are no smaller than a given threshold. Near-duplicate text alignment is computationally intensive. This is because there are $O(n^2)$ subsequences in a text with n tokens. To remedy this issue, a few recent studies propose to first generate the min-hash sketch of every subsequence in each text and then find all the subsequences whose min-hash sketches are similar to that of the query. They introduce the concept of "compact windows" and show that the $O(n^2k)$ min-hashes in a text with n tokens can be losslessly compressed in compact windows using $O(nk)$ space, where k is the sketch size. However, the space cost $O(nk)$ is still too high for long texts, especially when the sketch size k is large. To address this issue, we propose to use One Permutation Hashing (OPH) to generate the min-hash sketch and introduce the concept of "OPH compact windows". Although the size of each sketch remains the same, which is $O(k)$, we prove that all the $O(n^2k)$ min-hashes generated by OPH in a text with n tokens can be losslessly compressed in OPH compact windows using only $O(n + k)$ space. Note the generation of OPH compact windows does not necessitate the enumeration of the $O(n^2k)$ min-hashes. Moreover, we develop an algorithm to find all the sketches in a text similar to that of the query directly from OPH compact windows, along with three optimizations.We conduct extensive experiments on three real-world datasets. Empirical results show our proposed algorithms significantly outperformed existing methods in terms of index cost and query latency and scaled well.

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

Additional Key Words and Phrases: Text Alignment, Jaccard Similarity, One Permutation Hashing

ACM Reference Format:
Zhencan Peng, Yuheng Zhang, and Dong Deng. 2024. Near-Duplicate Text Alignment with One Permutation Hashing. *Proc. ACM Manag. Data* 2, N4 (SIGMOD), Article 200 (September 2024), 26 pages. https://doi.org/10.1145/3677136

1 Introduction

Texts are ubiquitous in the real world. A common and computational intensive operation over texts is near-duplicate text alignment. Specifically, given a collection of long texts and a short query text, the near-duplicate text alignment problem finds all the *subsequences* in the texts that are similar to the query. Near-duplicate text alignment finds applications in bioinformatics [37],

*Corresponding author

Authors' Contact Information: Zhencan Peng, Rutgers University, USA, zp128@scarletmail.rutgers.edu; Yuheng Zhang, Rutgers University, USA, yz1391@scarletmail.rutgers.edu; Dong Deng, Rutgers University, USA, dong.deng@rutgers.edu.

ACM 2836-6573/2024/9-ART200
https://doi.org/10.1145/3677136

large language model evaluation [11], log analysis [29], plagiarism detection [42], just to name a few. In this paper, we use the widely adopted Jaccard similarity to measure the similarity of two texts, where each text is an ordered list of tokens. Based on the application, a token can be a word, a n-gram, a byte-pair-encoding token [20], a k-mer [19], etc.

Due to the high computational cost, existing methods mostly adopt the "seeding-extension-filtering" heuristic [42], which involves many hard-to-tune parameters and lacks accuracy guarantees [58]. To address these issues, a few recent studies [17, 39] resort to the min-hash techniques, which are designed for efficient and accurate Jaccard similarity estimation [9]. These studies first generate and index the min-hash sketch of every subsequence in each text and then find all the subsequences whose min-hash sketches are similar to that of the query. These approaches only have a couple of parameters and guarantee to find all the near-duplicate subsequences whose estimated Jaccard similarities to the query are no smaller than a given threshold. However, the number of subsequences (and min-hash sketches) in a text is quadratic to the text length. To reduce the index size, they introduce the concept of "compact windows" to losslessly compress all the min-hash sketches in a text. Nevertheless, their index sizes are still overly large, especially for massive datasets, such as the training corpora of large language models, as we will discuss later.

To further reduce the index size, in this paper, we propose to use One Permutation Hashing (OPH) to generate the min-hash sketch [33] and introduce the concept of "OPH compact windows". Specifically, to generate the min-hash sketch of a subsequence, the classical way [9] is to use k independent random hash functions to generate k min-hashes (each min-hash is the smallest token hash value in the subsequence). OPH uses only one random hash function. It partitions all possible token hash values into k disjoint bins (e.g., using the modular arithmetic) and generates one min-hash from each bin (which is the smallest token hash value from that bin in the subsequence).

The key observation of existing methods [17, 39] is that nearby subsequences in a text would share the same min-hash. Specifically, let T be a text, $\mathrm{T}[c]$ be its c-th token, $\mathrm{T}[i, j]$ be its subsequence from the i-th token to j-th token, and h be a random hash function that maps a token $\mathrm{T}[c]$ to a hash value $h(\mathrm{T}[c])$. As illustrated in Figure 1, suppose the hash values of the tokens between $\mathrm{T}[l]$ and $\mathrm{T}[r]$ (excluding $\mathrm{T}[c]$) are all greater than $h(\mathrm{T}[c])$. Based on the definition, the min-hash of every subsequence $\mathrm{T}[i, j]$ where $l \le i \le c \le j \le r$ must be $h(\mathrm{T}[c])$. Thus existing methods represent the min-hashes of all these subsequences in a single "compact window".

We observe that the number of nearby subsequences sharing the same OPH min-hash is much more than that of nearby subsequences sharing the same (classical) min-hash. As shown in Figure 1, suppose the hash values of the tokens between $\mathrm{T}[l']$ and $\mathrm{T}[r']$ (excluding $\mathrm{T}[c]$) either are greater than $h(\mathrm{T}[c])$ or not from the same bin as $h(\mathrm{T}[c])$. We have $l' \le l$ and $r \le r'$ as $h(\mathrm{T}[i])$ is greater than $h(\mathrm{T}[c])$ for every $l \le i \ne c \le r$. Based on the definition of OPH, the OPH min-hash of every subsequence $\mathrm{T}[i, j]$ where $l' \le i \le c \le j \le r'$ must be $h(\mathrm{T}[c])$. We propose to represent the OPH min-hashes of all these subsequences in a single "OPH compact window" of constant size. Clearly, an OPH compact window represents more subsequences (and their min-hashes) than a classic compact window, as $l' \le l$ and $r \le r'$.

For ease of presentation, hereinafter, we refer to a subsequence in a text and its min-hash interchangeably. Given a text with n tokens, existing methods prove that the $O(n^2k)$ min-hashes in the text can be losslessly compressed in $O(nk)$ compact windows using $O(nk)$ space and $O(nk \log n)$ time, where k is the sketch size. In this paper, we show the $O(n^2k)$ OPH min-hashes in the text can be losslessly compressed in $O(n+k)$ OPH compact windows using $O(n+k)$ space and $O(n \log n+k)$ time. The improvement is significant, especially when the sketch size k is large. Furthermore, an algorithm is developed in a recent work [39] to find all the min-hash sketches in a text similar to that of the query directly from the compact windows of the text. It takes $O(n^2k^2 \log nk)$ to process a text with n tokens. In this paper, we adapt the algorithm for OPH compact windows and

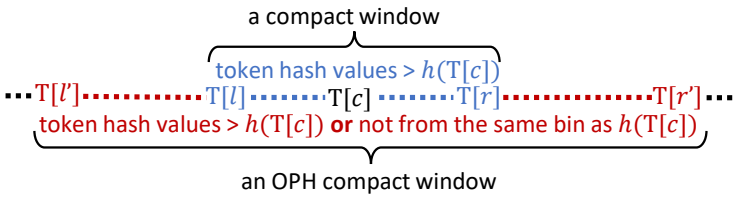


Fig. 1. Compact windows versus OPH compact windows.

propose three optimizations. The optimizations are also applicable to the algorithm for compact windows. We end up with a similar sketch search algorithm for OPH compact windows that takes $O((n+k)\log(n+k))$ time. To avoid redundant results, we discuss how to report only the longest near-duplicate subsequences (i.e., a near-duplicate subsequence $T[i, j]$ is omitted if there exists another near-duplicate subsequence $T[i', j']$ where $[i, j] \subset [i', j']$).

To summarize, this paper makes the following contributions.

- We introduce the concept of "OPH compact windows" and prove that all the $O(n^2k)$ min-hash generated by OPH in a text with n tokens can be losslessly compressed in $O(n+k)$ OPH compact windows using $O(n+k)$ space and $O(n \log n + k)$ time, where k is the sketch size.
- We develop an algorithm to efficiently find all the OPH min-hash sketches in a text similar to that of the query directly from the OPH compact windows of the text, along with three optimizations.
- We conduct extensive experiments on three real-world datasets. Experimental results show that our methods significantly outperformed existing methods and scaled well.

The rest of the paper is organized as follows. We define the problem and introduce preliminary knowledge in Section 2. Section 3 discusses how to losslessly compress all the min-hash generated by OPH in a text into OPH compact windows. Section 4 finds all the sketches similar to that of the query directly from the OPH compact windows. Section 5 presents the experimental results. Section 6 surveys related work, while Section 7 concludes the paper.

2 Preliminaries

2.1 Problem Definition

We first define a few notations. A text T is a list of tokens, where $T[i]$ is the i-th token in the text. Based on the application, a token can be a whitespace-split word [59], a byte-pair-encoding token [20], a k-mer (a.k.a. n-gram) [19], etc. The text length $|T|$ is the total number of tokens in the text T. We define $T[i, j]$ as the subsequence of T that starts from $T[i]$ and ends with $T[j]$ (inclusive), where $1 \le i \le j \le |T|$. Each text or subsequence can be viewed as a set of tokens, where duplicate tokens are removed (i.e., we do not consider multiplicity for simplicity in this paper). Thus we use text and set interchangably when the context is clear. To evaluate the similarity of two texts, we use the popular Jaccard similarity. Specifically, given two texts T and S, their Jaccard similarity is defined as

$$J_{T,S} = \frac{|T \cap S|}{|T \cup S|}$$

where $|T \cap S|$ and $|T \cup S|$ are respectively the number of common tokens and the total number of distinct tokens in them.

Next, we formally define the near-duplicate text alignment (a.k.a. near-duplicate sequence search in [39]) problem as below.

Definition 1 (Near-Duplicate Text Alignment). *Given a collection of texts* D*, a query* Q*, and a Jaccard similarity threshold* $\theta \in [0, 1]$*, the near-duplicate text alignment problem returns all the subsequences* $T[i, j]$*, where* $T \in D$ *and* $1 \le i \le j \le |T|$*, such that* $J_{Q,T[i,j]} \ge \theta$*.*

Example 1. *Consider a set* $\mathrm{D} = \{\mathrm{T}_1, \mathrm{T}_2, \mathrm{T}_3\}$ *of three texts, where* $\mathrm{T}_1 = (7, 1, 2, 8, 5, 9, 7), \mathrm{T}_2 = (2, 9, 7, 8, 4, 6, 3), \mathrm{T}_3 = (6, 1, 1, 9, 5, 8, 2)$*, and a query* $\mathrm{Q} = (8, 2, 9)$*. Each token is an integer. Let the Jaccard similarity threshold be* $\theta = 0.75$*. The near-duplicate text alignment problem returns three subsequences* $\mathrm{T}_1[3, 6] = (2, 8, 5, 9)$*,* $\mathrm{T}_2[1, 4] = (2, 9, 7, 8)$*, and* $\mathrm{T}_3[4, 7] = (9, 5, 8, 2)$*. This is because only the Jaccard similarities of these subsequences to the query are no smaller than the threshold* $\theta = 0.75$*.*

2.2 Min-Hash

k-mins Sketch. Let h be a random universal hash function. The min-hash of a text is its smallest token hash value. A commonly used universal hash function family is $h(x) = (ax + b) \mod p$ where p is a large prime and $a > 0$ and $b \geq 0$ are two random integers smaller than p [53]. We assign each token a unique ID and use the token and its ID interchangeably. The hash function $h(x)$ takes a token ID x as input and outputs the token's hash value. We denote the min-hash of a text T as $h(\mathrm{T}) = \min(h(\mathrm{T}[i]) \mid 1 \leq i \leq |\mathrm{T}|)$. It has been shown that the min-hash collision probability of two texts T and S is equal to their Jaccard similarity [9], i.e.,

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

With k independent random universal hash functions $h_1, h_2, \cdots, h_k$, the Jaccard similarity of two texts T and S can be estimated by

$$\frac{1}{k} \sum_{i=1}^{k} \mathbf{1}\{h_i(\mathrm{T}) = h_i(\mathrm{S})\} \tag{1}$$

where $\mathbf{1}$ is an indicator function. This is an unbiased estimator of the Jaccard similarity with low variance [9]. The k-mins sketch of the text T consists of the k min-hash values $h_1(\mathrm{T}), h_2(\mathrm{T}), \cdots, h_k(\mathrm{T})$.

One Permutation Hashing (OPH). A drawback of the k-mins sketch is that it needs k independent random universal hash functions. Thus it takes $O(nk)$ time to generate the k-mins sketch of a text with n tokens. To remedy this issue, the one permutation hashing (OPH) [33] is proposed. It uses only one hash function. Formally, given a random hash function h, OPH first evenly partitions all possible hash values of h into k disjoint bins $\Omega_1, \Omega_2, \cdots, \Omega_k$. Then, given a text T, OPH generates one min-hash from each bin for the text, where the t-th min-hash of T is defined as

$$h(\mathrm{T}, t) = \begin{cases} \min(h(\mathrm{T}[x]) \in \Omega_t) & \text{if } \exists\, x, h(\mathrm{T}[x]) \in \Omega_t \\ E & \text{otherwise.} \end{cases} \tag{2}$$

where E means no token hash value from the text T belongs to the bin Ω_t and the t-th bin of the text is "empty" (thus the t-th min-hash of the text is denoted as E in this case). Given two texts T and S, it has been shown that, when their t-th bins are not empty at the same time, their min-hash collision probability is equal to their Jaccard similarity, i.e.,

$$\mathbf{Pr}(h(\mathrm{T}, t) = h(\mathrm{S}, t) \mid h(\mathrm{T}, t) \neq E \vee h(\mathrm{S}, t) \neq E) = \mathrm{J}_{\mathrm{T,S}}.$$

Thus their Jaccard similarity can be accurately estimated by

$$\hat{\mathrm{J}}_{\mathrm{T,S}} = \frac{N_{mat}}{k - N_{emp}} \tag{3}$$

Hash Table: token hash value with its token position

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

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$h(\mathrm{S}[i])$	90	64	39	30	66	42	22	63	28	56	91	11	96	99	53	61	88	73	31

OPH Table: OPH min-hash with its bin id

t	1	2	3	4	5	6	7	8	9	10
$h(\mathrm{T}, t)$	91	22	53	64	E	E	57	48	39	90
$h(\mathrm{S}, t)$	11	22	53	64	E	56	E	28	39	30

Partition Ω into 10 bins:

$\boldsymbol{\Omega_{10} = \{x \bmod 10 = 0\}}$,

$\boldsymbol{\Omega_t = \{x \bmod 10 = t\}, 1 \le t \le 9}$

Fig. 2. An example of one permutation hashing min-hash.

where

$$N_{emp} = \sum_{t=1}^{k} \mathbf{1}\{h(\mathrm{T}, t) = E \wedge h(\mathrm{S}, t) = E\},$$
$$N_{mat} = \sum_{t=1}^{k} \mathbf{1}\{h(\mathrm{T}, t) = h(\mathrm{S}, t) \wedge (h(\mathrm{T}, t) \neq E \vee h(\mathrm{S}, t) \neq E)\}. \tag{4}$$

That is to say N_{emp} and N_{mat} are respectively the number of "jointly empty bins" and the number of min-hash collisions of the two texts. Note that two empty bins (whose min-hashes are denoted as E) do not count for a min-hash collision. This estimator is also an unbiased estimator with low variance [33]. Clearly, it uses only one random universal hash function and generates the k-mins sketch of a text with n tokens in $O(n + k)$ time. The sketch size remains the same, which is $O(k)$.

Example 2. *The first two tables in Figure 2 (namely "hash tables") show the token hash values of two texts* T *and* S *under a random universal hash function h. The min-hash of* T *is* $h(\mathrm{T}) = h(\mathrm{T}[3]) = 22$*, while the min-hash of* S *is* $h(\mathrm{S}) = h(\mathrm{S}[12]) = 11$*. To generate* $k = 10$ *min-hashes for the two texts using OPH, we partition all the token hash values into 10 bins* $\Omega_1, \cdots, \Omega_{10}$ *using the modulus operation as illustrated in the figure. For example, the token hash value 22 belongs to* Ω_2 *as* $22 \mod 10 = 2$*. As shown in the third table in Figure 2 (namely "OPH table"), we can generate 10 min-hashes for each of the two texts* T *and* S*. For example,* $h(\mathrm{T}, 3) = 53$ *as only the two token hash values* $h(\mathrm{T}[14]) = 73$ *and* $h(\mathrm{T}[15]) = 53$ *in* T *belongs to* Ω_3 *and* 53 *is the smaller one. Note that* $h(\mathrm{T}, 5) = h(\mathrm{S}, 5) = E$ *as no token hash value in both* T *and* S *belong to* Ω_5*. In this example, we have* $N_{emp} = 1$ *and* $N_{mat} = 4$ *respectively as the* 5*-th bin of* T *and* S *are jointly "empty" and when t is* 2, 3, 4, 9 *their min-hash collides. Thus the estimated Jaccard similarity* $\hat{\mathrm{J}}_{\mathrm{T,S}} = \frac{4}{10-1} = 0.444$.

Remark. With the Jaccard similarity estimator, we aim to find all the subsequences whose estimated Jaccard similarities with the query are no smaller than the given threshold (i.e., replacing the $\mathrm{J}_{\mathrm{Q,T}[i,j]}$ in Definition 1 with $\hat{\mathrm{J}}_{\mathrm{Q,T}[i,j]}$). To this end, we propose to first generate the OPH min-hash sketch of every subsequence in a given text (Section 3) and then find all min-hash sketches similar to that of the query as determined by the estimator (Section 4).

3 OPH Compact Windows

In this section, we discuss how to generate the OPH min-hash sketch of every subsequence in a given text T. That is to say, we need to calculate $h(\mathrm{T}[i, j], t)$ for every $1 \le i \le j \le |\mathrm{T}| = n$ and $1 \le t \le k$. In total, there are $O(kn^2)$ OPH min-hash in a text with n tokens, which is excessively large. However, we observe that, in the same bin, many nearby subsequences would share the same OPH min-hash. For example, consider the two subsequences $\mathrm{T}[10, 13]$ and $\mathrm{T}[11, 13]$ as shown in Figure 2. We have $h(\mathrm{T}[10, 13], 8) = h(\mathrm{T}[11, 13], 8) = h(\mathrm{T}[12]) = 48$ in Ω_8. Based on this observation, we introduce the concept of OPH compact windows and prove that the $O(kn^2)$ OPH min-hash in a

text with n tokens can be losslessly compressed in $O(n+k)$ OPH compact windows using $O(n+k)$ space.

3.1 Empty OPH Compact Windows

We first focus on the empty bins where the OPH min-hash is E. Let $\mathrm{T}[l-1]$ and $\mathrm{T}[r+1]$ be two tokens whose hash values are from the same bin Ω_t while the hash values of the tokens in between are not. That is to say, $h(\mathrm{T}[l-1]) \in \Omega_t$ and $h(\mathrm{T}[r+1]) \in \Omega_t$ while $h(\mathrm{T}[i]) \notin \Omega_t$ for every $l \le i \le r$. We observe that the OPH min-hash of every subsequence between the two tokens must be E in the bin Ω_t, i.e., $h(\mathrm{T}[i,j],t) = E$ for every $l \le i \le j \le r$. This is because there is no token hash value in $\mathrm{T}[i,j]$ belongs to Ω_t. Thus, based on Equation 2, its OPH min-hash in Ω_t must be E. For example, consider the text T in Figure 2. $\mathrm{T}[6]$ and $\mathrm{T}[13]$ are two tokens whose hash values are from Ω_9 while the hash values of those in between are not. All the subsequences between them, including $\mathrm{T}[7,10]$ and $\mathrm{T}[8,12]$, share the same OPH min-hash E in Ω_9. Based on this observation, we formally define the empty OPH compact window as below.

Definition 2 (Empty OPH Compact Window). *Given a random hash function h and a partition of k bins $\Omega_1, \cdots, \Omega_k$, an empty OPH compact window in a text* T *is a tuple* $\langle \mathrm{T}, h, t, l, r\rangle$ *where*

(1) $1 \le l \le r \le |\mathrm{T}|$;
(2) $1 \le t \le k$;
(3) $\forall\, l \le i \le r$, $h(\mathrm{T}[i]) \notin \Omega_t$;
(4) if $l \neq 1$, $h(\mathrm{T}[l-1]) \in \Omega_t$;
(5) if $r \neq |\mathrm{T}|$, $h(\mathrm{T}[r+1]) \in \Omega_t$.

Note that, in the definition, the last two conditions are intentionally designed to make the empty OPH compact window "maximal" such that it represents as many OPH min-hashes as possible. The empty OPH compact window $\langle \mathrm{T}, h, t, l, r\rangle$ represents all the OPH min-hashes $h(\mathrm{T}[i,j],t)$ where $l \le i \le j \le r$. For example, in Figure 2, $\langle \mathrm{T}, h, 9, 7, 12\rangle$ is an empty OPH compact window. The OPH min-hashes $h(\mathrm{T}[8,10],9)$ and $h(\mathrm{T}[7,11],9)$ are represented by it.

We observe that all the OPH min-hashes represented by an empty OPH compact windows are E, as formalized below.

Lemma 1. *Given an empty OPH compact window* $\langle \mathrm{T}, h, t, l, r\rangle$, $h(\mathrm{T}[i,j],t) = E$ *for every* $l \le i \le j \le r$.

Proof. Based on the third condition in Definition 2, $\forall x \in [i,j] \subseteq [l,r]$, $h(\mathrm{T}[x]) \notin \Omega_t$. Thus, based on Equation 2, $h(\mathrm{T}[i,j],t) = E$. □

Moreover, we observe that every OPH min-hash that is equal to E is represented by one and only one empty OPH compact window.

Lemma 2. *Given a text* T*, every OPH min-hash* $h(\mathrm{T}[i,j],t) = E$ *is represented by one and only one empty OPH compact window.*

Proof. Let l be the smallest position where $\mathrm{T}[x] \notin \Omega_t$ for every $x \in [l,i]$ and r be the largest position where $\mathrm{T}[x] \notin \Omega_t$ for every $x \in [j,r]$. We prove $\langle \mathrm{T}, h, t, l, r\rangle$ is an empty OPH compact window. Since l is the smallest position where $\mathrm{T}[x] \notin \Omega_t$ for every $x \in [l,i]$, we have either $l = 1$ or $h(\mathrm{T}[l-1]) \in \Omega_t$. Thus the fourth condition in Definition 2 is satisfied. Similarly, the fifth condition is also satisfied. Furthermore, by Equation 2, as $h(\mathrm{T}[i,j],t) = E$, we have $h(\mathrm{T}[x]) \notin \Omega_t$ for every $x \in [i,j]$. Moreover, $\mathrm{T}[x] \notin \Omega_t$ for every $x \in [l,i]$ and $x \in [j,r]$. Together, we have $h(\mathrm{T}[x]) \notin \Omega_t$ for every $x \in [l,r]$. Thus the third condition in Definition 2 is satisfied. The first two conditions hold trivially. Thus $\langle \mathrm{T}, h, t, l, r\rangle$ is an empty OPH compact window and it represents the OPH min-hash $h(\mathrm{T}[i,j],t)$.

Next, suppose $h(\mathrm{T}[i, j], t)$ is represented by two different empty OPH compact windows $\langle \mathrm{T}, h, t, l, r\rangle$ and $\langle \mathrm{T}, h, t, l', r'\rangle$. That is to say $l \le i \le j \le r$ and $l' \le i \le j \le r'$, while either $l \ne l'$ or $r \ne r'$. Without loss of generality, let us assume $l \ne l'$ and $l < l'$. Thus we have $1 \le l < l'$. By Definition 2 (fourth condition), $h(\mathrm{T}[l' - 1]) \in \Omega_t$. However, we also have $l \le l' - 1 \le r$ as $l' - 1 \le i \le r$. By the third condition, we have $h(\mathrm{T}[l' - 1]) \notin \Omega_t$, which leads to a contradiction. Thus $h(\mathrm{T}[i, j], t)$ cannot be represented by two different empty OPH compact windows. □

Furthermore, we find that there are at most $n + k - 2$ empty OPH compact windows in a text with n tokens, as formalized below.

Lemma 3. *There are at most $n + k - 2$ empty OPH compact windows in a text with n tokens.*

Proof. Consider a text T with n tokens and a random hash function h. We first prove by contradiction that for each $l \in [2, n]$, there is at most one empty OPH compact window $\langle \mathrm{T}, h, t, l, r\rangle$ in the text. Suppose there are two different empty OPH compact windows $\langle \mathrm{T}, h, t_1, l, r_1\rangle$ and $\langle \mathrm{T}, h, t_2, l, r_2\rangle$ for some $l \in [2, n]$ in the text T. Based on the fourth condition in Definition 2, since $l \ne 1$, we have $h(\mathrm{T}[l - 1]) \in \Omega_{t_1}$ and $h(\mathrm{T}[l - 1]) \in \Omega_{t_2}$. Thus $t_1 = t_2$, which yields $r_1 \ne r_2$. Without loss of generality, we assume $r_1 < r_2$. On the one hand, based on the fifth condition in Definition 2, since $r_1 \ne |\mathrm{T}|$, $h(\mathrm{T}[r_1 + 1]) \in \Omega_{t_1}$. On the other hand, based on the third condition in Definition 2, since $l \le r_1 + 1 \le r_2$, $h(\mathrm{T}[r_1 + 1]) \notin \Omega_{t_2}$. Since $t_1 = t_2$, $h(\mathrm{T}[r_1 + 1]) \in \Omega_{t_1}$ contradicts with $h(\mathrm{T}[r_1 + 1]) \notin \Omega_{t_2}$.

Next, we prove by contradiction for each $t \in [1, k]$, there is at most one empty OPH compact window $\langle \mathrm{T}, h, t, l = 1, r\rangle$. Suppose there are two different empty OPH compact windows $\langle \mathrm{T}, h, t, 1, r_1\rangle$ and $\langle \mathrm{T}, h, t, 1, r_2\rangle$ for some $t \in [1, k]$ in the text T. Then $r_1 \ne r_2$. Without loss of generality, we assume $r_1 < r_2$. Based on the fifth condition in Definition 2, since $r_1 \ne |\mathrm{T}|$, $h(\mathrm{T}[r_1 + 1]) \in \Omega_t$. However, based on the third condition in Definition 2, since $l \le r_1 + 1 \le r_2$, $h(\mathrm{T}[r_1 + 1]) \notin \Omega_t$, which contradicts with $h(\mathrm{T}[r_1 + 1]) \in \Omega_t$.

In addition, suppose $h(\mathrm{T}[1]) \in \Omega_t$. The tuple $\langle \mathrm{T}, h, t, l = 1, r\rangle$ cannot be an empty OPH compact window for any $r \in [1, n]$; otherwise, the third condition in Definition 2 indicates $h(\mathrm{T}[1]) \notin \Omega_t$, which contradicts with $h(\mathrm{T}[1]) \in \Omega_t$.

Thus, in total, there are at most $n + k - 2$ empty OPH compact windows $\langle \mathrm{T}, h, t, l, r\rangle$ in a text with n tokens. At most $n - 1$ of them having $l \ne 1$ and at most $k - 1$ of them having $l = 1$. □

3.2 Non-Empty OPH Compact Window

Next, we focus on the non-empty bins where the OPH min-hash is not E. We observe that all the subsequences around a "local minimal" token hash value in a bin would share the same OPH min-hash in that bin. For example, consider the text T in Figure 2. There are three token hash values in it from Ω_9, which are $h(\mathrm{T}[2]) = 59$, $h(\mathrm{T}[6]) = 39$, and $h(\mathrm{T}[13]) = 99$. All the subsequences containing the "local minimal" $\mathrm{T}[13]$ while not containing $\mathrm{T}[6]$ would share the same OPH min-hash in Ω_9. For example, as one can verify, for the two subsequences $\mathrm{T}[10, 14]$ and $\mathrm{T}[11, 15]$, in Ω_9, their OPH min-hash $h(\mathrm{T}[10, 14], 9) = h(\mathrm{T}[12, 15], 9) = h(\mathrm{T}[13]) = 99$. Based on this observation, we formally define the non-empty OPH compact window as below.

Definition 3 (Non-Empty OPH Compact Window). *Given a random hash function h and k bins $\Omega_1, \cdots, \Omega_k$, a non-empty OPH compact window in a text* T *is a tuple $\langle \mathrm{T}, h, t, l, c, r\rangle$ where*

(1) $1 \le l \le c \le r \le |\mathrm{T}|$;
(2) $1 \le t \le k$ and $h(\mathrm{T}[c]) \in \Omega_t$;
(3) $\forall\, l \le i \ne c \le r$, either $h(\mathrm{T}[i]) \notin \Omega_t$ or $h(\mathrm{T}[i]) > h(\mathrm{T}[c])$;
(4) if $l \ne 1$, $h(\mathrm{T}[l - 1]) \in \Omega_t$ and $h(\mathrm{T}[l - 1]) < h(\mathrm{T}[c])$;
(5) if $r \ne |\mathrm{T}|$, $h(\mathrm{T}[r + 1]) \in \Omega_t$ and $h(\mathrm{T}[r + 1]) < h(\mathrm{T}[c])$.

The non-empty OPH compact window $\langle \mathrm{T}, h, t, l, c, r\rangle$ represents all the OPH min-hashes $h(\mathrm{T}[i, j], t)$ where $l \le i \le c \le j \le r$. The last two conditions are intentionally designed to make the non-empty OPH compact window "maximal" such that it represents as many OPH min-hashes as possible. For example, in Figure 2, $\langle \mathrm{T}, h, 9, 1, 6, 15\rangle$ is a non-empty OPH compact window. The OPH min-hashes $h(\mathrm{T}[4, 7], 9)$ and $h(\mathrm{T}[5, 8], 9)$ are represented by it, which are both equal to $h(\mathrm{T}[6]) = 39$.

We observe that all the OPH min-hashes represented by a non-empty OPH compact window are the same, as formalized below.

Lemma 4. *Given a non-empty OPH compact window* $\langle \mathrm{T}, h, t, l, c, r\rangle$, $h(\mathrm{T}[i, j], t) = h(\mathrm{T}[c])$ *for every* $l \le i \le c \le j \le r$.

Proof. Based on the second condition in Definition 3, we have $h(\mathrm{T}[c]) \in \Omega_t$. Thus, based on Equation 2, $h(\mathrm{T}[i, j], t) \neq E$ as $c \in [i, j]$. In addition, based on the third condition in Definition 3, for every $x \in [i, j]$, either $h(\mathrm{T}[x]) \notin \Omega_t$ or $h(\mathrm{T}[x]) > h(\mathrm{T}[c])$. That is to say, among all the token hash values in $\mathrm{T}[i, j]$ that belong to Ω_t, $h(\mathrm{T}[c])$ is the smallest. Thus $h(\mathrm{T}[i, j], t) = h(\mathrm{T}[c])$ by definition. □

Moreover, every OPH min-hash that is not equal to E is represented by one and only one non-empty OPH compact window.

Lemma 5. *Given a text* T*, every OPH min-hash* $h(\mathrm{T}[i, j], t) \neq E$ *is represented by one and only one non-empty OPH compact window.*

Proof. Let c be the position where $h(\mathrm{T}[c]) = h(\mathrm{T}[i, j], t)$. Let l be the smallest position where $h(\mathrm{T}[x]) \notin \Omega_t$ or $h(\mathrm{T}[x]) > h(\mathrm{T}[c])$ for every $x \in [l, i]$ and r be the largest position where $h(\mathrm{T}[x]) \notin \Omega_t$ or $h(\mathrm{T}[x]) > h(\mathrm{T}[c])$ for every $x \in [j, r]$. We prove $\langle \mathrm{T}, h, t, l, c, r\rangle$ is a non-empty OPH compact window. Since l is the smallest position where $h(\mathrm{T}[x]) \notin \Omega_t$ or $h(\mathrm{T}[x]) > h(\mathrm{T}[c])$, we have either $l = 1$ or $h(\mathrm{T}[l-1]) \in \Omega_t$ and $h(\mathrm{T}[l-1]) < h(\mathrm{T}[c])$. Thus, the fourth condition in Definition 3 is satisfied. Similarly, the fifth condition is also satisfied. Furthermore, by Equation 2, as $h(\mathrm{T}[c]) = \min(\{h(\mathrm{T}[x]) \in \Omega_t\})$, we have $h(\mathrm{T}[x]) \notin \Omega_t$ or $h(\mathrm{T}[x]) > h(\mathrm{T}[c])$ for every x where $l \le x \neq c \le r$. Moreover, $h(\mathrm{T}[x]) \notin \Omega_t$ or $h(\mathrm{T}[x]) > h(\mathrm{T}[c])$ for every $x \in [l, i]$ and $x \in [j, r]$. Together, we have $h(\mathrm{T}[x]) \notin \Omega_t$ or $h(\mathrm{T}[x]) > h(\mathrm{T}[c])$ for every $x \in [l, r]$. Thus, the third condition in Definition 3 is satisfied. The first two conditions hold trivially. Thus, $\langle \mathrm{T}, h, t, l, c, r\rangle$ is a non-empty OPH compact window and it represents the OPH min-hash $h(\mathrm{T}[i, j], t)$.

Next, suppose $h(\mathrm{T}[i, j], t)$ is represented by two different non-empty OPH compact windows $\langle \mathrm{T}, h, t, l, c, r\rangle$ and $\langle \mathrm{T}, h, t, l', c', r'\rangle$. That is to say $l \le i \neq c \le j \le r$ and $l' \le i \neq c' \le j \le r'$, while any below is met, $l \neq l'$, $r \neq r'$ or $c \neq c'$. Based on Lemma 4, among all x such that $h(\mathrm{T}[x]) \in \Omega_t$ where $i \le x \le j$, both $h(\mathrm{T}[c])$ and $h(\mathrm{T}[c'])$ are $\min(\{h(\mathrm{T}[x])\})$. Since only one token's hash value is chosen as $\min(\{h(\mathrm{T}[x])\})$, $c = c'$. Therefore, either $l \neq l'$ or $r \neq r'$. Without loss of generality, let us assume $l \neq l'$ and $l < l'$. Thus, we have $1 \le l < l'$. By Definition 3 (fourth condition), $h(\mathrm{T}[l'-1]) \in \Omega_t$ and $h(\mathrm{T}[l'-1]) < h(\mathrm{T}[c])$. However, we also have $l \le l'-1 \le r$ as $l'-1 \le i \le r$. By the third condition, we have $h(\mathrm{T}[l'-1]) \notin \Omega_t$ or $h(\mathrm{T}[l'-1]) > h(\mathrm{T}[c])$, which leads to a contradiction. Thus, $h(\mathrm{T}[i, j], t)$ cannot be represented by two different non-empty OPH compact windows. □

Lemma 6. *There are at most* n *non-empty OPH compact windows in a text with* n *tokens.*

Proof. Consider a text T with n tokens and a random hash function h. We prove that for each $c \in [1, n]$, there is at most one non-empty OPH compact window $\langle \mathrm{T}, h, t, l, c, r\rangle$ in T. We prove it by contradiction. Assume there are two different non-empty OPH compact windows $\langle \mathrm{T}, h, t_1, l_1, c, r_1\rangle$ and $\langle \mathrm{T}, h, t_2, l_2, c, r_2\rangle$. Based on the second condition of Definition 3, we have $h(\mathrm{T}[c]) \in \Omega_{t_1}$ and

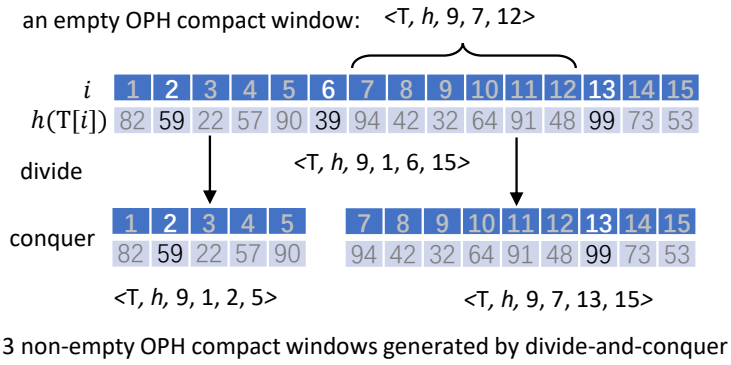


Fig. 3. An example of OPH compact window generation.

$h(\mathrm{T}[c]) \in \Omega_{t_2}$. Thus $t_1 = t_2$, which yields either $l_1 \neq l_2$ or $r_1 \neq r_2$. Without loss of generality, let us assume $l_1 \neq l_2$ and $l_1 < l_2$. Thus, we have $1 \leq l_1 < l_2$. By Definition 3 (fourth condition), $h(\mathrm{T}[l_2 - 1]) \in \Omega_{t_2}$ and $h(\mathrm{T}[l_2 - 1]) < h(\mathrm{T}[c])$. However, we also have $l_1 \leq l_2 - 1 < c \leq r_1$. By the third condition, we have $h(\mathrm{T}[l_2 - 1]) \notin \Omega_{t_1}$ or $h(\mathrm{T}[l_2 - 1]) > h(\mathrm{T}[c])$, which contradicts with $h(\mathrm{T}[l_2 - 1]) \in \Omega_{t_2}$ and $h(\mathrm{T}[l_2 - 1]) < h(\mathrm{T}[c])$ as $t_1 = t_2$. Thus, there are at most n non-empty OPH compact windows $\langle \mathrm{T}, h, t, l, c, r \rangle$ in a text with n tokens, one for each $c \in [1, n]$. □

Remark. Both empty and non-empty OPH compact windows are called OPH compact window. The $O(n + k)$ OPH compact windows in a text with n tokens losslessly represent the $O(kn^2)$ OPH min-hash in it, as formalized below.

Theorem 1. *Given a text with n tokens, there are $O(n + k)$ OPH compact windows in it. Every OPH min-hash in the text is represented by one and only one of its OPH compact windows.*

Proof. Based on Lemmas 3 and 6, there are at most $n + k - 2$ empty OPH compact windows and n non-empty OPH compact windows. Thus in total, there are $O(n + k)$ OPH compact windows. Based on Lemmas 2 and 5, each OPH min-hash in the text is represented by one and only one of its OPH compact windows. □

In comparison, previous works use $O(nk)$ compact windows to represent all the $O(n^2k)$ classical min-hash (without OPH) in a text with n tokens [17]. Our work significantly reduces the space consumption by a factor of k. Next, we discuss how to efficiently generate all the OPH compact windows in a text.

3.3 OPH Compact Window Generation

In this section, we discuss how to efficiently generate all the OPH compact windows in a text. We first define a notation. Given a text T, we induce k texts from it, where the t-th induced text T^t contains all the token hash values from the t-th bin Ω_t in T. That is to say, $\mathrm{T}^t = (\langle i, h(\mathrm{T}[i]) \rangle \mid h(\mathrm{T}[i]) \in \Omega_t)$. For ease of presentation, we also include two dummy tokens $\langle 0, \infty \rangle$ and $\langle |\mathrm{T}| + 1, \infty \rangle$ in each of the k induced texts. For example, consider the text T as shown in Figure 3, we have $\mathrm{T}^9 = (\langle 0, \infty \rangle, \langle 2, 59 \rangle, \langle 6, 39 \rangle, \langle 13, 99 \rangle, \langle 16, \infty \rangle)$. The i and $h(\mathrm{T}[i])$ in a tuple $\langle i, h(\mathrm{T}[i]) \rangle \in \mathrm{T}^t$ refer to the token position and token hash value. When the tuples in the induced text T^t are ordered by their token positions, every two consecutive tuples $\mathrm{T}^t[x] = \langle i, h(\mathrm{T}[i]) \rangle$ and $\mathrm{T}^t[x + 1] = \langle j, h(\mathrm{T}[j]) \rangle$ determines an empty OPH compact window $\langle \mathrm{T}, h, t, i + 1, j - 1 \rangle$. This is because the hash value of every token between $\mathrm{T}[i + 1]$ and $\mathrm{T}[j - 1]$ cannot be from Ω_t (third condition of Definition 2), either $i + 1 \neq 1$ or $h(\mathrm{T}[i]) \in \Omega_t$ (fourth condition), and either $j - 1 \neq |\mathrm{T}|$ or $h(\mathrm{T}[j]) \in \Omega_t$ (fifth condition).

Algorithm 1: OPHCompactWindowGeneration

Input: T: a text; h: a random hash function; k: an integer.
Output: W: all the non-empty OPH compact windows in T; W_E: all the empty OPH compact windows in T.

1 **foreach** $t \in [1, k]$ **do** add $\langle 0, \infty \rangle$ to T^t;
2 **foreach** $i \in [1, |\mathrm{T}|]$ **do**
3 add $\langle i, h(\mathrm{T}[i]) \rangle$ to T^t where t is given by $h(\mathrm{T}[i]) \in \Omega_t$;
4 **foreach** $t \in [1, k]$ **do** add $\langle |\mathrm{T}| + 1, \infty \rangle$ to T^t;
5 **foreach** $t \in [1, k]$ **do**
`// generate empty OPH compact windows`
6 **foreach** $x \in [1, |\mathrm{T}^t| - 1]$ **do**
7 add $\langle \mathrm{T}, h, t, i + 1, j - 1 \rangle$ to W_E where i and j are the token positions in $\mathrm{T}^t[x]$ and $\mathrm{T}^t[x + 1]$;
`// generate non-empty OPH compact windows`
8 DivideConquer($\mathrm{T}^t, 1, |\mathrm{T}|, W$);
9 **return** W *and* W_E;

Algorithm 2: DivideConquer(T^t, l, r, W)

Input: T^t: the t-th induced text; l and r: two integers; W: all the non-empty OPH compact windows in $\mathrm{T}[l, r]$.

1 **if** *exists a tuple in* T^t *whose token position is within* $[l, r]$ **then**
2 $\langle c, h(\mathrm{T}[c]) \rangle \leftarrow$ RangeMinimumQuery($\mathrm{T}^t, [l, r]$);
3 add $\langle \mathrm{T}, h, t, l, c, r \rangle$ to W;
4 DivideConquer($\mathrm{T}^t, l, c - 1, W$);
5 DivideConquer($\mathrm{T}^t, c + 1, r, W$);

To generate all the non-empty OPH compact windows in a given text T, we use a divide-and-conquer algorithm. For each induced text T^t, the algorithm finds the tuple $\langle c, h(\mathrm{T}[c]) \rangle$ with the smallest token hash value in T^t. Then $\langle \mathrm{T}, h, t, 1, c, |\mathrm{T}| \rangle$ must be a non-empty OPH compact window. Then the algorithm recursively finds all the non-empty OPH compact windows from Ω_t in the two ranges $[1, c - 1]$ and $[c + 1, |\mathrm{T}|]$.

The pseudo-code of OPH compact window generation is shown in Algorithm 1. The algorithm takes a text T, a random hash function h, and an integer k as input and outputs W and W_E, the sets of all the empty and non-empty OPH compact windows in T. It first generates k induced texts from T, each with two dummy tuples (Lines 1 to 4). Then, it iterates each T^t to generate the compact windows(Lines 5 to 9). Based on the observation that adjacent tuples in T^t determine a non-empty OPH compact window, it iterates each pair of $\mathrm{T}^t[x]$ and $\mathrm{T}^t[x + 1]$, where i and j are their token positions, and adds $\langle \mathrm{T}, h, t, i + 1, j - 1 \rangle$ to W_E (Line 7). It then invokes the DivideConquer to recursively divide each T^t and generate non-empty OPH compact windows.

The pseudo code of DivideConquer is shown in Algorithm 2. It takes an induced text T^t, a range $[l, r]$, and a result set W as input. If there exists a tuple in T^t whose token position is within the range $[l, r]$, it identifies the tuple $\langle c, h(\mathrm{T}[c]) \rangle$ with the smallest hash value in T^t where $c \in [l, r]$ (Line 2), adds a non-empty OPH compact window $\langle \mathrm{T}, h, t, l, c, r \rangle$ into W (Line 3), and recursively generates the non-empty OPH compact windows in the two ranges $[l, c - 1]$ and $[c + 1, r]$ (Lines 4 to 5).

Example 3. *Consider the induced text* T^9 *and its tuples as shown in Figure 3. The algorithm has already inserted two dummy tuples into* T^9*, resulting in* $\mathrm{T}^9 = (\langle 0, \infty\rangle, \langle 2, 59\rangle, \langle 6, 39\rangle, \langle 13, 99\rangle, \langle 16, \infty\rangle)$*. It then iterates over adjacent tuple pairs to generate all the empty OPH compact windows. For example,* $\langle \mathrm{T}, h, 9, 7, 12\rangle$ *is a generated empty OPH compact window between tuple pairs* $\langle 6, 39\rangle$ *and* $\langle 13, 99\rangle$*. Subsequently, the algorithm finds the minimum hash value within range [1,15] in* T^9 *as* $\mathrm{T}[6]$*. Therefore, a non-empty compact window* $\langle \mathrm{T}, h, 9, 1, 6, 15\rangle$ *is generated. The text* T *is then divided into* $\mathrm{T}[1, 5]$ *and* $\mathrm{T}[7, 15]$*. Then the algorithm recursively processes* $\mathrm{T}[1, 5]$ *and* $\mathrm{T}[7, 15]$*, generating* $\langle \mathrm{T}, h, 9, 1, 2, 5\rangle$ *and* $\langle \mathrm{T}, h, 9, 7, 13, 15\rangle$ *respectively.*

Theorem 2. *The OPH compact window generation algorithm is sound (all tuples generated are OPH compact windows) and complete (all OPH compact windows are generated).*

Complexity Analysis. Given a text with n tokens, it takes $O(n+k)$ to produce the k induced texts. The generation of empty OPH compact windows involves iterating all the induced texts, which takes $O(n + k)$ time. When generating the non-empty OPH compact windows, we can use the segment tree to facilitate the range minimum query operations. The divide-and-conquer procedure is invoked at most $O(n)$ times. Each invocation takes $O(\log n)$ time using the segment tree. Thus the overall time complexity is $O(n \log n + k)$. The OPH compact windows generated take $O(n + k)$ space. In comparison, existing methods take $O(nk \log n)$ time to losslessly compress the classical min-hash in $O(nk)$ space [17, 39].

4 Near-Duplicate Text Alignment

4.1 OPH Interval Scan

In this section, we discuss how to efficiently find all the subsequences in a text whose OPH min-hash sketches are similar to that of the query directly from the OPH compact windows. Consider a subsequence $\mathrm{T}[i, j]$, a query Q, and a threshold θ. Based on Equation 3, the OPH min-hash estimation $\hat{\mathrm{J}}_{\mathrm{T}[i,j],\mathrm{Q}} \geq \theta$ iff. $N_{mat} + \theta N_{emp} \geq k\theta$. Let W and W_E be the sets of non-empty and empty OPH compact windows in T. Based on Lemma 2, $h(\mathrm{T}[i, j], t) = E$ iff. there exists an empty OPH compact window $\langle \mathrm{T}, h, t, l, r\rangle \in \mathrm{W}_E$ where $i \in [l, r]$ and $j \in [l, r]$. Based on Equation 4, we have

$$\begin{aligned} N_{emp} &= \sum_{t=1}^{k} \mathbf{1}\{h(\mathrm{T}[i, j], t) = E \wedge h(\mathrm{Q}, t) = E\} \\ &= \sum_{\langle \mathrm{T},h,t,l,r\rangle \in \mathrm{W}_E} \mathbf{1}\{i \in [l, r] \wedge j \in [l, r] \wedge h(\mathrm{Q}, t) = E\}. \end{aligned}$$

Similarly, based on Lemmas 4 and 5, every non-empty OPH min-hash $h(\mathrm{T}[i, j], t)$ is represented by one and only one non-empty OPH compact window $\langle \mathrm{T}, h, t, l, c, r\rangle \in \mathrm{W}$ where $i \in [l, c]$ and $j \in [c, r]$ and $h(\mathrm{T}[i, j], t) = h(\mathrm{T}[c])$. Based on Equation 4, we have

$$\begin{aligned} N_{mat} &= \sum_{t=1}^{k} \mathbf{1}\{h(\mathrm{T}[i, j], t) = h(\mathrm{Q}, t) \wedge (h(\mathrm{T}[i, j], t) \neq E \vee h(\mathrm{Q}, t) \neq E)\} \\ &= \sum_{\langle \mathrm{T},h,t,l,c,r\rangle \in \mathrm{W}} \mathbf{1}\{i \in [l, c] \wedge j \in [c, r] \wedge h(\mathrm{Q}, t) = h(\mathrm{T}[c])\}. \end{aligned}$$

We say an empty OPH compact window $\langle \mathrm{T}, h, t, l, r\rangle$ collides with the query Q iff. $h(\mathrm{Q}, t) = E$. Similarly, a non-empty OPH compact window $\langle \mathrm{T}, h, t, l, c, r\rangle$ collides with the query Q iff. $h(\mathrm{Q}, t) = h(\mathrm{T}[c])$. Let $C \subseteq \mathrm{W}$ and $C_E \subseteq \mathrm{W}_E$ be the set of collided non-empty and empty OPH compact

windows, respectively. We have

$$N_{emp} = \sum_{\langle \mathrm{T},h,t,l,r\rangle \in C_E} \mathbf{1}\{i \in [l,r] \wedge j \in [l,r]\} \text{ and}$$
$$N_{mat} = \sum_{\langle \mathrm{T},h,t,l,c,r\rangle \in C} \mathbf{1}\{i \in [l,c] \wedge j \in [c,r]\}.$$

We define that the "left interval" and "right interval" of a non-empty OPH compact window $\langle \mathrm{T}, h, t, l, c, r\rangle$ are respectively $[l, c]$ and $[c, r]$, while the left interval and right interval of an empty OPH compact window $\langle \mathrm{T}, h, t, l, r\rangle$ are both $[l, r]$. Then, $\hat{\mathrm{J}}_{\mathrm{T}[i,j],\mathrm{Q}} \geq \theta$ iff. i and j are respectively in the left and right intervals of M collided non-empty OPH compact windows and M' collided empty OPH compact windows where $M + M'\theta \geq k\theta$. Next we discuss how to find all subsequences of this kind.

OPH Interval Scan. We collect the two endpoints of the left interval of every collided OPH compact windows in C and C_E, i.e., l and $c+1$ for every collided non-empty OPH compact window $\langle \mathrm{T}, h, t, l, c, r\rangle \in C$ and l and $r+1$ for every collided empty OPH compact window $\langle \mathrm{T}, h, t, l, r\rangle \in C_E$. We observe that for any two adjacent endpoints $x < x'$ (i.e., there is no endpoint between x and x'), the interval $[x, x'-1]$ either is entirely covered (i.e., $\subseteq$) by the left interval of a collided OPH compact window in C or C_E or is disjoint with all the left intervals in C and C_E. This is because otherwise there must exist an endpoint between x and x', which contradicts with that x and x' are adjacent. Let $C' \subseteq C$ and $C'_E \subseteq C_E$ be all the collided non-empty and empty OPH compact windows whose left intervals covers $[x, x'-1]$. If $|C'| + \theta|C'_E| \geq k\theta$, we repeat the above steps for the right intervals.

Specifically, we collect the two endpoints of the right interval of every collided OPH compact windows in C' and C'_E. For any consecutive endpoints $y < y'$, let $C'' \subseteq C'$ and $C''_E \subseteq C'_E$ be all the collided non-empty and empty OPH compact windows whose right intervals covers $[y, y'-1]$. Then $[x, x'-1]$ and $[y, y'-1]$ are respectively in the left and right intervals of all the collided OPH compact windows in C'' and C''_E. If $|C''| + \theta|C''_E| \geq k\theta$, as discussed earlier, we have $\hat{\mathrm{J}}_{\mathrm{T}[i,j],\mathrm{Q}} \geq \theta$ for every $i \in [x, x'-1]$ and $j \in [y, y'-1]$. To reduce the output cost, the "compact alignment" $([x, x'-1], [y, y'-1])$ is returned, which is a concise representation of all the subsequences $\mathrm{T}[i, j]$ where $i \in [x, x'-1]$ and $j \in [y, y'-1]$.

Algorithm 3 shows the pseudo-code. It takes a text T, a threshold $k\theta$, the set of collided non-empty and empty OPH compact windows C and C_E in T as input and outputs a set $\mathcal{R}$ of compact alignments. The algorithm first checks if $|C| + \theta|C_E| < k\theta$. If so, no result can be produced and the algorithm returns an empty set (Line 1). Next, it collects all the endpoints of the left intervals of the collided OPH compact windows (Lines 2 to 5). Each endpoint consists of a token position u, a weight w, and the collided OPH compact window d the endpoint comes from. It sorts the endpoints by their token positions (Line 6). Then, for each distinct token position u_x in the endpoints (Line 7), it visits all the endpoints whose token positions are u_x (Line 8). The algorithm maintains a counter `cnt` (Line 1) to keep track of the value $|C'| + \theta|C'_E|$. The token position in an endpoint indicates the start or end of a left interval. The counter `cnt` increases/decreases by $w = 1$ when the start/end of a left interval of a collided non-empty OPH compact window $d \in C$ is encounter, at which moment d is added to/removed from C' (Line 10). Similarly, the counter `cnt` increases/decreases by $w = \theta$ when the start/end of a left interval of a collided empty OPH compact window $d \in C_E$ is encounter, at which moment d is added to/removed from C'_E (Line 11). After visiting all the endpoints with token positions u_x, if `cnt` exceeds $k\theta$, the algorithm repeats the above process for the right intervals of collided OPH compact windows in C' and C'_E (Lines 13 to 21) to find compact alignments (Lines 22 to 23).

Algorithm 3: OPHIntervalScan(T, $k\theta, C, C_E$)

Input: T: a text; $k\theta$: a threshold; C: the set of collided non-empty OPH compact windows in T; C_E: the set of collided empty OPH compact windows in T.
Output: $\mathcal{R}$: a set of compact alignments.

```
if |C| + θ|C_E| < kθ then return ∅; cnt = 0;
foreach d = ⟨T, h, t, l, c, r⟩ ∈ C do
    add endpoints (l, 1, d) and (c + 1, −1, d) into ep;
foreach d = ⟨T, h, t, l, r⟩ ∈ C_E do
    add endpoints (l, θ, d) and (r + 1, −θ, d) into ep;
sort endpoints (u, w, d) in ep in the ascending order of u;
foreach distinct token position u_x in ep do
    foreach endpoint (u_x, w, d) in ep do
        cnt ← cnt + w;
        if w = 1/−1 then add/remove d to/from C';
        if w = θ/−θ then add/remove d to/from C'_E;
    if cnt ≥ kθ then
        // repeat for the right intervals
        cnt' = 0;
        foreach ⟨T, h, t, l, c, r⟩ ∈ C' do
            add endpoints (c, 1) and (r + 1, −1) into ep';
        foreach ⟨T, h, t, l, r⟩ ∈ C'_E do
            add endpoints (l, θ) and (r + 1, −θ) into ep';
        sort endpoints (v, w) in ep' in the ascending order of v;
        foreach distinct token position v_y in ep' do
            foreach endpoint (v_y, w) in ep' do
                cnt' ← cnt' + w;
            if cnt' ≥ kθ then
                add ([u_x, u_{x+1} − 1], [v_y, v_{y+1} − 1]) to R;
return R;
```

Example 4. *Consider the set C of two collided non-empty OPH compact windows $d_1 = \langle \mathrm{T}, h, 1, 1, 3, 9\rangle$, $d_2 = \langle \mathrm{T}, h, 1, 4, 8, 13\rangle$, and the set C_E of one collided empty OPH compact window $d_3 = \langle \mathrm{T}, h, 2, 6, 10\rangle$ as shown in Figure 4. Let $k\theta = 1.6$. The distinct token positions of their left intervals are $u_1 = 1, u_2 = 4, u_3 = 6, u_4 = 9$, and $u_5 = 11$. After visiting u_3, the set $C' = \{d_2\}$ and the set $C'_E = \{d_3\}$. As one can verify, the overlap of the left intervals of d_2 and d_3 is exactly $[4, 8] \cap [6, 10] = [u_3, u_4)$. Since $|C'| + \theta|C'_E| = 1.8$, we repeat the process over C' and C'_E and produce a compact alignment $([6, 8], [8, 10])$.*

Theorem 3. *The set $\mathcal{R}$ of compact alignments generated by OPH interval scan satisfies that: (1) for every subsequence $\mathrm{T}[i, j]$ where $\hat{\mathrm{J}}_{\mathrm{Q,T}[i,j]} \geq \theta$, there exists one and only one compact alignment $([l_l, l_r], [r_l, r_r]) \in \mathcal{R}$ where $i \in [l_l, l_r]$ and $j \in [r_l, r_r]$ and (2) for every compact alignment $([l_l, l_r], [r_l, r_r]) \in \mathcal{R}$ and every $i \in [l_l, l_r]$ and $j \in [r_l, r_r]$, we have $\hat{\mathrm{J}}_{\mathrm{Q,T}[i,j]} \geq \theta$.*

Complexity Analysis. Let $m = |C| + |C_E|$ be the total number of collided OPH compact windows. The algorithm takes $O(m \log m)$ to sort and visit the endpoints of the left intervals. There are $O(m)$

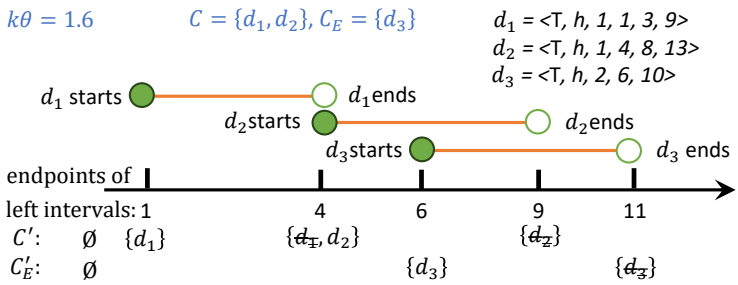


Fig. 4. An example of OPH interval scan.

unique left interval endpoints. For each of them, when necessary, the endpoints of the right intervals in C' and C'_E are sorted and visited, which takes $O(m \log m)$ time as there are $O(m)$ collided OPH compact windows in C' and C'_E. Thus the total time complexity is $O(m^2 \log m)$. Based on Theorem 1, $m = O(n + k)$. Thus the time complexity of OPH interval scan is $O((n + k)^2 \log(n + k))$.

4.2 Optimizations

Order by Token Positions. In the divide-and-conquer algorithm (Algorithm 2) for OPH compact window generation, when there are multiple duplicate tokens with the same smallest hash value in the input, we can either break the tie arbitrarily or order them by their positions in the input. Although the two choices end up with the same number of OPH compact windows, ordering by token positions guarantees that the left intervals of collided OPH compact windows with the same OPH min-hash do not overlap with each other, which benefits the OPH interval scan algorithm as it reduces the chance to check the right intervals. Note that the complexity analysis in Section 4.1 assumes ties are broken arbitrarily.

Specifically, suppose the duplicate token hash values are ordered by their positions in OPH compact windows generation. In Algorithm 3, for each unique left interval endpoint, there are at most k collided OPH compact windows in C' and C'_E. Thus the time complexity of OPH interval scan becomes $O(m \log m + mk \log k) = O((n + k)(\log(n + k) + k \log k))$.

Longest Near-Duplicate Subsequence. In practice, to avoid redundant results, we only need to report the longest near-duplicate subsequences in many applications. That is to say, a near-duplicate subsequence $\mathrm{T}[i, j]$ (i.e., $\hat{\mathrm{J}}_{\mathrm{T}[i,j],\mathrm{Q}} \geq \theta$) is omitted if there exists another near-duplicate subsequence $\mathrm{T}[i', j']$ (i.e., $\hat{\mathrm{J}}_{\mathrm{T}[i',j'],\mathrm{Q}} \geq \theta$) where $[i, j] \subset [i', j']$. In this case, in OPH interval scan (Algorithm 3, Line 18), we can traverse the endpoints of the right intervals in descending order. Once a compact alignment $([u_x, u_{x+1}], [v_y, v_{y+1}])$ is found, we can terminate the traversal. This is because the near-duplicate subsequence $\mathrm{T}[u_x, v_{y+1}]$ dominates all the near-duplicate subsequence generated later during the traversal, if there is any.

Moreover, when visiting the left interval endpoints (Algorithm 3, Lines 7 to 11), we maintain a variable z ($z = 0$ at the beginning). When a compact alignment $([u_x, u_{x+1} - 1], [v_y, v_{y+1} - 1])$ is found when traversing the right interval endpoints, only if $z < v_{y+1} - 1$, we update $z = v_{y+1} - 1$ and report a longest near-duplicate subsequence $\mathrm{T}[u_x, v_{y+1} - 1]$. This is because u_x is monotonically increasing. When $z \geq v_{y+1} - 1$, the near-duplicate subsequence $\mathrm{T}[u_x, v_{y+1} - 1]$ must be dominated by another one generated earlier and thus can be omitted.

With the above two modifications, it can be proved that the near-duplicate subsequences generated are all longest near-duplicate subsequences. Furthermore, all near-duplicate subsequences are dominated by the near-duplicate subsequences generated.

Maintaining Right Intervals in a Segment Tree. To avoid repeatedly sorting and visiting the endpoints of the right intervals of collided OPH compact windows, we propose to manage the right

intervals in a dynamic segment tree when visiting the endpoints of the left intervals. Specifically, whenever an endpoint l of the left interval of a collided OPH compact window $\langle \mathrm{T}, h, t, l, c, r\rangle \in C$ (or $\langle \mathrm{T}, h, t, l, r\rangle \in C_E$) is encountered, we increase the weight of the segment $[c, r]$ (or $[l, r]$) by 1 (or θ) in the dynamic segment tree. In contrast, whenever an endpoint $c+1$ (or $r+1$) of a collided OPH compact window $\langle \mathrm{T}, h, t, l, c, r\rangle \in C$ (or $\langle \mathrm{T}, h, t, l, r\rangle \in C_E$) is encountered, we decrease the weight of the segment $[c, r]$ (or $[l, r]$) by 1 (or θ) in the dynamic segment tree. If $\texttt{cnt} \geq k\theta$, we query the dynamic segment tree and retrieve the rightmost segment whose weight is at least $k\theta$ in the dynamic segment tree, if there is one. Using the lazy propagation trick, each query or update on the dynamic segment tree takes $O(\log m)$ time. Thus the time complexity of the OPH interval scan (only reporting the longest near-duplicate subsequences) is reduced to $O(m \log m) = O((n+k)\log(n+k))$.

Remark. Similar to the OPH interval scan algorithm, an algorithm is presented in [39] to find all the sketches in a text similar to that of the query directly from the compact windows of the text. We denote the algorithm as KMS interval scan. Its time complexity is $O(n^2k^2 \log nk)$. The three optimizations also work for KMS interval scan. Specifically, for the same token hash values, if we break the ties by the token positions, the time complexity of KMS interval scan becomes $O(nk(\log nk + k \log k))$. In addition, with the dynamic segment tree, the time complexity of KMS interval scan can be reduced to $O(nk \log nk)$ in the same way.

4.3 Put Together

To process a collection D of texts, during offline index time, we first generate all the OPH compact windows in every text in D using Algorithm 1 and index them in an inverted index $\mathcal{L}$ and a two-dimensional array $\mathcal{W}$, where the non-empty OPH compact window $\langle \mathrm{T}, h, t, l, c, r\rangle$ is appended to the inverted list $\mathcal{L}[h(\mathrm{T}[c])]$, while the empty OPH compact window $\langle \mathrm{T}, h, t, l, r\rangle$ is appended to the array $\mathcal{W}[t]$. Once a query Q arrives, we first calculate its k OPH min-hash. If the t-th OPH min-hash $h(\mathrm{Q}, t) = E$, we retrieve the array $\mathcal{W}[t]$; otherwise, it does not equal to E, we retrieve the inverted list $\mathcal{L}[h(\mathrm{Q}, t)]$. All the OPH compact windows in the retrieved arrays and inverted lists are collided OPH compact windows. We use a heap to merge and group them by the text T. Each group contains the sets C and C_E of collided non-empty and empty OPH compact windows in a text T. We invoke Algorithm 3 to find all the compact alignments between T and the query Q.

4.4 Discussion

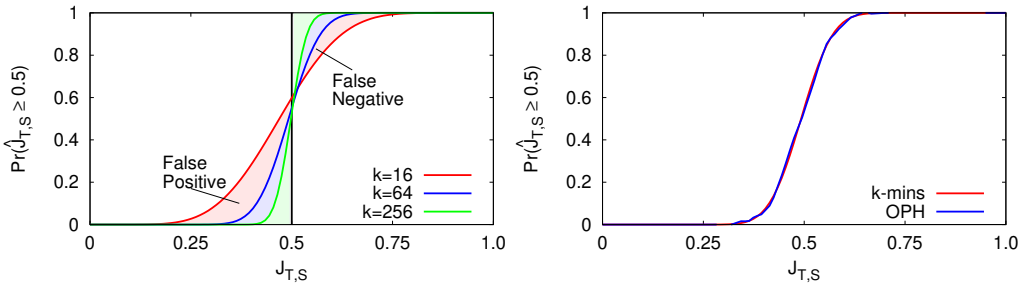
Min-Hash Estimation Accuracy. We first focus on the k-mins estimation. T and S are considered similar iff. $\hat{\mathrm{J}}_{\mathrm{T,S}} \geq \theta$. Based on Equation 1, the probability that T and S are reported as similar is

$$\mathbf{Pr}(\hat{\mathrm{J}}_{\mathrm{T,S}} \geq \theta) = 1 - \sum_{j=0}^{m-1} \binom{k}{j} (\mathrm{J}_{\mathrm{T,S}})^j (1 - \mathrm{J}_{\mathrm{T,S}})^{k-j} \tag{5}$$

where $m = \lceil k\theta \rceil$. Figure 5(a) illustrates three probability distributions, $\mathbf{Pr}(\hat{\mathrm{J}}_{\mathrm{T,S}} >= 0.5)$, under different k for k-mins estimation.

The probability distribution of OPH estimation is very complex due to its sample-without-replacement nature [33]. To get a sense of its estimation accuracy, we simulate the probability distribution of OPH estimation. For this purpose, for each similarity score in $[0.001, 0.002, \cdots, 1.000]$, we randomly generated 1000 pairs of texts whose Jaccard similarities are equal to that score and estimated their Jaccard similarity using OPH estimation[1]. Figure 5(b) shows the result for $k = 64$ and $\theta = 0.5$. As we can see, the simulated probability distribution of OPH estimation is almost the same as the theoretical probability distribution of k-mins estimation. Thus the two estimations bear similar accuracy in practice.

[1]https://github.com/rutgers-db/DuplicateSearch_k-partition/tree/pzcgyy/simulation



(a) k-mins for $k = 16, 64, 256$ (b) k-mins and OPH for $k = 64$

Fig. 5. The probability distribution of k-mins estimation and simulated probability distribution of OPH estimation.

Integration into SQLite. FTS5 is a virtual table module in SQLite for full-text search. It stores the inverted index in the database tables. Implementing our near-duplicate text alignment operation in SQLite is feasible with the FTS5 extension. This is because our techniques also index the OPH compact windows in inverted indexes.

Multiplicity. For multiplicity Jaccard similarity, each occurrence of a duplicate token has a unique hash value. Thus the hash value of a token is no longer fixed, which makes (OPH) compact window generation rather challenging. Feng et al. present an algorithm [17] to handle it for compact window generation (without complexity analysis though). It is possible to extend that algorithm to accommodate OPH compact windows. We leave it as a future work.

5 Experimental Evaluation

Environment. We implemented all the methods (including the baselines) by ourselves using C++ and compiled the programs using GCC 11.4.0 with -O3 optimization. All the experiments were conducted on a machine with an Intel(R) Xeon(R) Gold 6148 CPU@2.40GHz and 1TB memory. The operating system is Ubuntu 18.04.s.

Datasets. We used three datasets PAN [41], OWT [23], and WIKI [24]. PAN[2] is a benchmark for external plagiarism detection [41]. Each text in PAN is a book. OWT contains 8 million web texts highly ranked on Reddit, while WIKI consists of cleaned and tokenized English articles from Wikipedia. We use the GPT2Tokenizer[3] to tokenize the texts in each dataset. Table 1 shows the dataset details. For example, on average, each text in OWT contains 1095 tokens.

5.1 Evaluating OPH Compact Window Generation

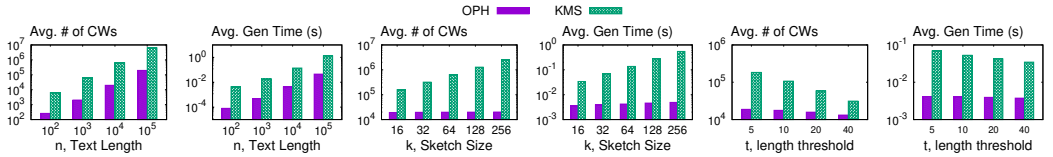
In this section, we compare our OPH compact window generation algorithm (denoted as OPH) with the compact window generation algorithm (denoted as KMS) in the existing work [39].

Exp-1. We first vary the text length n and the sketch size k and report the number of OPH compact windows generated by KMS and the number of classical compact windows generated by OPH, as well as the generation time. Note that the number of compact windows generated and the generation time are proportional to the index size and index time in KMS and OPH. Specifically, for a fixed text length n, texts in a dataset with less than n tokens were omitted, while texts with more than n tokens were truncated to the first n tokens. We report the average number of compact windows

[2]https://pan.webis.de/data.html
[3]https://huggingface.co/docs/transformers/model_doc/gpt2#transformers.GPT2Tokenizer

Table 1. Dataset details.

dataset	# of tokens	# of text	avg. length	size (in MB)
OWT	8,773,924,283	8,013,769	1,095	33,501
PAN	642,380,109	11,093	57,909	2,451
WIKI	2,172,725,598	2,926,536	742	8,300



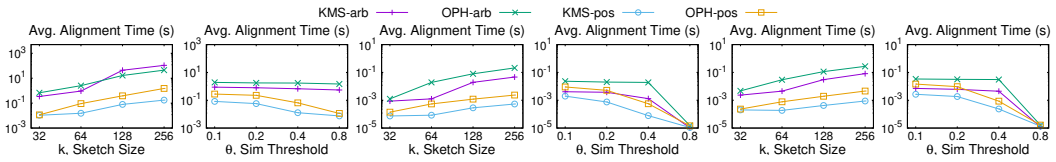
(a) OWT (k=64, t=1) (b) OWT (k=64, t=1) (c) OWT (n=10^4, t=1) (d) OWT (n=10^4, t=1) (e) OWT (k=64, n=10^4) (f) OWT (k=64, n=10^4)

Fig. 6. Evaluating compact window (CW for short) generation.

and generation time per text. The three datasets had very similar results. Due to space limit, we only report the results on OWT.

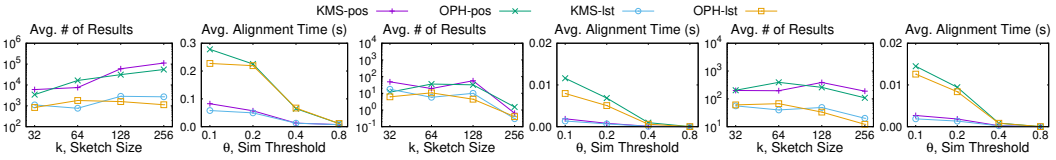
Figures 6(a) to 6(d) show the results. As we can see, although both the number of classical compact windows and the number of OPH compact windows grew linearly with the text length n, the number of OPH compact windows is significantly lower than that of the classical compact windows. Moreover, with the increase of the sketch size k, the number of OPH compact windows remained almost unchanged, while the number of classical compact windows grew proportionally with k. The results are consistent with our complexity analysis, where there are $O(nk)$ classical compact windows and $O(n + k)$ OPH compact windows in a text with n tokens. The compact window generation time emerged similar trends as the number of compact windows. For example, when the sketch size k increased from 16 to 256, it took 3.6 ms and 4.9 ms on average to generate the OPH compact windows of a text with 10,000 tokens, while it took 34 ms and 548 ms on average to generate the classical compact windows. This is because it takes $O(n \log n + k)$ time to generate the OPH compact windows, while it takes $O(nk \log n)$ for classical compact window generation.

Exp-2. In KMS, only near-duplicate subsequences no shorter than a given length threshold t will be reported. KMS utilizes the length threshold to optimize compact window generation. This optimization is orthogonal to our proposed techniques. Thus OPH can also leverage this optimization. In Exp-1, the length threshold t is set to 1 for a fair comparison. In this experiment, we evaluate the impact of the length threshold. Specifically, we vary the length threshold t, while fix $k = 64$ and $n = 10000$, and report the average number of compact windows generated and the average generation time. Figures 6(e) to 6(f) show the results. As we can see, the number of classical compact windows decreased linearly with the length threshold t. This is because the expected number of classical compact windows in a text with n tokens is $2k\frac{n+1}{t+1} + 1$ [39]. In comparison, the number of OPH compact windows decreased sublinearly with t. This is because the advantage of OPH compact windows diminishes when short subsequences are filtered out by the length threshold. However, the number of OPH compact windows remained significantly lower than that of classical compact windows. For example, when $t = 40$, the number of OPH compact windows was only 40% of that of classical compact windows. In the meanwhile, the average classical compact window generation time decreased sublinearly with t. For example, it dropped from 0.07s to 0.035s when t increased from 5 to 40. This is because the overhead of calculating token hash values and building segment tree does not change with the length threshold t. Nevertheless, the average generation time of OPH was still much lower than that of KMS. For example, the average OPH compact windows generation time was only 0.0041s when $t = 5$. Hereinafter, we set $t = 1$ by default.



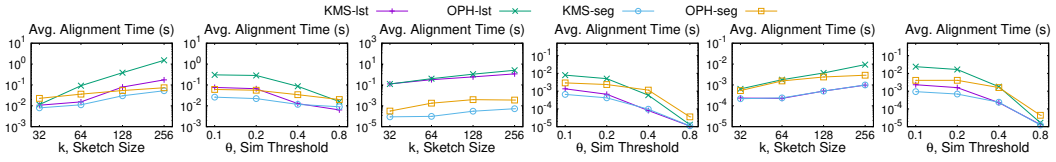
(a) PAN ($\theta = 0.4$) (b) PAN ($k = 64$) (c) OWT ($\theta = 0.4$) (d) OWT ($k = 64$) (e) WIKI ($\theta = 0.4$) (f) WIKI ($k = 64$)

Fig. 7. Evaluating the by position optimization.



(a) PAN ($\theta = 0.1$) (b) PAN ($k = 64$) (c) OWT ($\theta = 0.1$) (d) OWT ($k = 64$) (e) WIKI ($\theta = 0.1$) (f) WIKI ($k = 64$)

Fig. 8. Evaluating the longest near-duplicate subsequence optimization.



(a) PAN ($\theta = 0.4$) (b) PAN ($k = 64$) (c) OWT ($\theta = 0.4$) (d) OWT ($k = 64$) (e) WIKI ($\theta = 0.4$) (f) WIKI ($k = 64$)

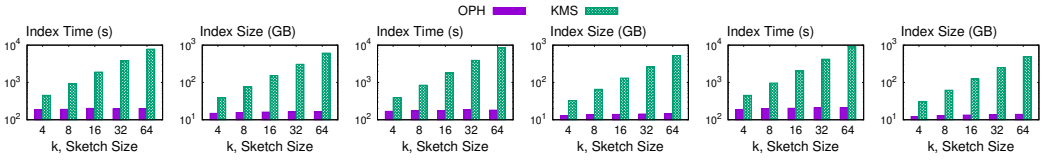
Fig. 9. Evaluating the segment tree optimization.

5.2 Evaluating OPH Interval Scan

In this section, we compare the interval scan algorithm in the existing work [39] (which finds all the k-mins sketches in a text similar to that of the query directly from classical compact windows) with our OPH interval scan algorithm and evaluate the three optimizations we proposed. The interval scan algorithm is coupled with a sanitizer similar to that of our algorithm that skips inputs with collided compact windows fewer than the threshold.

Query Construction: We pair each query with a text and report the average results of the two algorithms (alignment time or number of results) over 100 pairs. The PAN dataset comes with ground truth, where each suspicious passage is coupled with a source document containing the corresponding plagiarized passage. We use the suspicious passage and source document as our query-text pair. We randomly select 100 pairs from the ground truth for evaluation. The average query and text lengths were respectively 1424.6 tokens and 77665.1 tokens. For WIKI and OWT, we randomly select a text (with at least 64 unique tokens) and a 128-token subsequence (not necessarily from the selected text) from the dataset. The selected subsequence and text become a query-text pair. The average text lengths in the 100 randomly selected pairs were 1911.6 tokens and 1534.6 tokens in WIKI and OWT, respectively.

Exp-3: By Position Optimization. We first evaluate the optimization of ordering duplicate token hash values by token positions in compact window generation (instead of breaking ties arbitrarily [39]). We implemented four methods. `KMS-arb` is the interval scan algorithm in [39], while `KMS-pos` is the interval scan algorithm with our "by position" optimization. `OPH-arb` is our OPH interval scan algorithm, whereas `OPH-pos` is our algorithm with the by position optimization. Figure 7 shows the results. As we can see, the optimization significantly improved the alignment time, by almost 100×. For example, when $k = 256$, on the OWT dataset, the average alignment time for `OPH-pos` and `OPH-arb` were respectively $0.0023s$ and $0.21s$. This is consistent with our complexity



(a) PAN (b) PAN (c) WIKI (d) WIKI (e) OWT (f) OWT

Fig. 10. Evaluating index construction.

Table 2. Evaluating query accuracy ($k = 64$ for min-hash based methods).

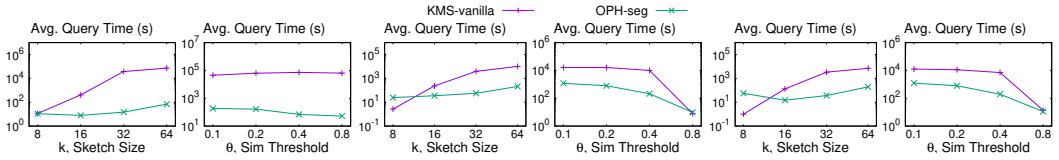
Threshold	KMS				OPH				SeedExtension				Faerie	bruteforce
	Time (s)	P	R	F1	Time (s)	P	R	F1	Time (s)	P	R	F1	Time (s)	Time (s)
0.2	0.789	0.497	0.875	0.633	0.309	0.503	0.877	0.639	13.995	1.000	0.436	0.608	163.846	196.836
0.3	0.676	0.738	0.850	0.790	0.195	0.761	0.820	0.790	12.231	1.000	0.465	0.635	87.939	125.949
0.4	0.668	0.748	0.862	0.801	0.091	0.819	0.858	0.838	10.360	1.000	0.572	0.727	53.596	90.140
0.5	0.663	0.792	0.918	0.850	0.040	0.782	0.925	0.848	8.583	1.000	0.595	0.746	32.169	60.879

analysis, where the OPH interval scan algorithm take $O(m^2 \log m)$ time and the optimization boost it to $O(m \log m + mk \log k)$ where $m \geq k$ is the number of collided OPH compact windows. The by position optimization is enabled by default in all the experiments hereinafter.

We observe that our OPH interval scan algorithm was slower than the interval scan algorithm. This is because the empty OPH min-hash leads to a lot of collided OPH compact window (i.e., a larger m), while there is no "empty min-hash" in the k-mins sketch in the interval scan algorithm. Nevertheless, our algorithm with the by position optimization (i.e., `OPH-pos`) still outperformed the vanilla interval scan algorithm (i.e., `KMS-arb`). The alignment time of all the algorithms grew with the increase of the sketch size k and the decrease of the threshold θ. This is because the number of collided compact windows increases with the sketch size, while fewer pairs can pass the sanitizer check when the threshold is high.

Exp-4: Longest Near-Duplicate Optimization. We evaluate the optimization that reports only the longest near-duplicate subsequences. We first vary the sketch size k and report the average number of results (i.e., compact alignments or longest near-duplicates) and then vary the similarity threshold θ and report the average alignment time. Four methods are evaluated. `KMS-pos` and `OPH-pos` as described earlier. `KMS-lst` and `OPH-lst` are respectively the interval scan algorithm and our algorithm with both the by position optimization and the longest near-duplicate optimization. Figure 8 shows the results. As we can see, the number of results produced by `OPH-lst` (or `KMS-lst`) was consistently smaller than that of `OPH-pos` (or `KMS-pos`). The optimization can effectively reduce redundant results. The number of results produced by `OPH-pos` (or `OPH-lst`) was on a par with that of `KMS-pos` (or `KMS-lst`). This is because they both use unbiased Jaccard similarity estimators with comparable variances. The optimization slightly reduces the alignment time. This is attributed to the early termination in the optimization.

Exp-5: Segment Tree Optimization. We compare four methods in this section. `KMS-lst` and `OPH-lst` as described earlier. `KMS-seg` and `OPH-seg` are respectively the interval scan algorithm and our algorithm with all the three optimizations. We vary the sketch size k and the similarity threshold θ and report the average alignment time. Figure 9 shows the results. As we can see, the segment tree optimization almost always helped reduce the average alignment time. For example, in the PAN dataset with $k = 256$ and $\theta = 0.4$, `KMS-seg` reduced the alignment time of `KMS-lst` from 0.079s to 0.030s. Furthermore, `OPH-seg` improved `OPH-lst`'s time from 1.51s to 0.07s, achieving up to a 200× speed up. This is because the optimization uses a segment tree to manage the right intervals of collided OPH compact windows, which avoids repeatedly sorting and visiting the same right intervals.



(a) PAN ($\theta = 0.4$) (b) PAN ($k = 64$) (c) OWT ($\theta = 0.4$) (d) OWT ($k = 64$) (e) WIKI ($\theta = 0.4$) (f) WIKI ($k = 64$)

Fig. 11. Evaluating query latency.

Table 3. Evaluating accuracy by varying sketch size ($\theta = 0.4$).

Method	Metric	k				
		16	32	64	128	256
KMS	Time (s)	0.119	0.341	0.668	43.66	107.5
	P	0.418	0.522	0.748	0.838	0.923
	R	0.850	0.907	0.862	0.880	0.924
	F1	0.561	0.662	0.801	0.859	0.924
OPH	Time (s)	0.004	0.011	0.092	0.386	1.515
	P	0.566	0.686	0.819	0.842	0.920
	R	0.714	0.818	0.858	0.893	0.876
	F1	0.632	0.746	0.838	0.867	0.898

5.3 Comparing with Existing Methods

Baselines. In this section, we compare our near-duplicate text alignment algorithm with four baselines. (1) The state-of-the-art method KMS [39]. (2) A "seeding-extension-filtering" baseline SeedExtension based on an existing method that won a plagiarism detection competition [42]. SeedExtension uses the common tokens between the data text and the query as "seed matches". Adjacent seed matches that are less than *maxgap* tokens away are merged to form "local largest" clusters. For each cluster, if the corresponding subsequence is similar to query, it reports the corresponding subsequence. For the remaining clusters, it reduces the threshold *maxgap* by one and re-clusters the seed matches (excluding those in the reported subsequences). The process is repeated until $maxgap = 4$. We set $maxgap = 4\theta|Q|$ initially in our experiment. (3) We adapted Faerie [31] for near-duplicate text alignment. Faerie is designed to find all the subsequences in a document that approximately match entities (short texts such as names) in a given dictionary. It supports many similarity measures, including Jaccard simlarity. We treat the query as an entity and the text as a document for near-duplicate text alignment. (4) A brute-force method that enumerates every subsequence in a text and compares them with the query.

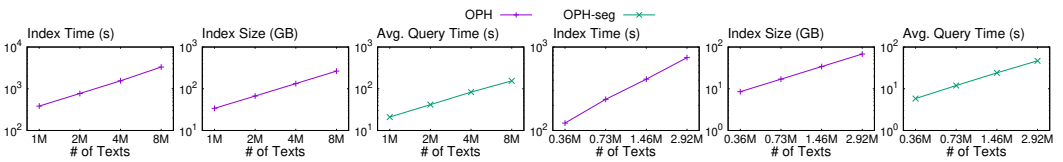
Exp-6: Index Construction. We construct inverted indexes using the classical compact windows generated by KMS and the OPH compact windows using OPH. We vary the sketch size k and report the index time and index size. Note that the other three baselines (SeedExtension, Faerie, and bruteforce) have no index phase. To avoid from getting the out-of-memory error, we randomly sampled 500,000 and 700,000 texts respectively from OWT and WIKI. The total numbers of tokens in the sampled texts in OWT and WIKI are approximately 550 millions and 520 millions, resulting in an average text length of 1099 and 742 tokens. For PAN, we used all the texts. Figure 10 shows the results. Both the index size and the index time grew linearly with the sketch size k for KMS, while they were almost flat for OPH. For example, for the PAN dataset, when the sketch size k increased 16 times from 4 to 64, the index time of KMS grew from 452s to 7812s, while the index size grew from 38GB to 612GB. In comparison, the index time of OPH only increased from 184s to 199s, while the index size changed from 14.8GB to 16.39GB. This is because the index time and index size are respectively determined by the total number of compact windows and the compact window generation time. In conclusion, the novel concept of OPH compact windows introduced in this paper can significantly reduce the index time and index size for near-duplicate text alignment.

Exp-7: Query Latency. We compare the query latency of our most advanced algorithm OPH-seg with the interval scan algorithm [39]. For a fair comparison, we enabled the longest near-duplicate optimization for the interval scan algorithm so that the two methods produce the same thing. We denote the interval scan algorithm with the longest near-duplicate optimization as KMS-vanilla. Note that OPH-seg uses the OPH index, while KMS-vanilla uses the KMS index. The query set is constructed the same way as described in Section 5.2. However, each query is aligned against all the (sampled) texts in the dataset rather than only one text. The other three baselines are too inefficient to run the experiment. We report their query latency in the next experiment using small scale datasets. Figure 11 shows the results. As we can see, our approach OPH-seg significantly outperformed KMS-vanilla. For example, on the PAN dataset with $k = 64$ and $\theta = 0.4$, KMS-vanilla takes over 71,000s to process a query, in contrast to only 69s for OPH-seg. This is attributed to the three optimizations we proposed for OPH interval scan. Nevertheless, KMS-vanilla occasionally outperformed OPH-seg. This is because the empty OPH min-hash creates a lot of min-hash collisions, which enforces OPH-seg to check the right intervals more often than KMS-vanilla. However, given the huge advantage in terms of index cost brought by OPH, the small overhead of query latency is acceptable.

Exp-8: Query Accuracy. Given a data text T and a query Q. Let $\mathcal{S} = \{\mathrm{T}[i, j] \mid \mathrm{J}_{\mathrm{T}[i,j],\mathrm{Q}} \geq \theta\}$ and $\oplus\mathcal{S} = \{x \mid \mathrm{T}[i, j] \in \mathcal{S} \text{ and } x \in [i, j]\}$ (i.e., all distinct token positions in $\mathcal{S}$). We use the brute-force method to find $\mathcal{S}$, which is the ground truth. Let $\mathcal{R}$ be a set of subsequences in T. We define the precision and recall of $\mathcal{R}$ as $|\oplus\mathcal{S} \cap \oplus\mathcal{R}|/|\oplus\mathcal{R}|$ and $|\oplus\mathcal{S} \cap \oplus\mathcal{R}|/|\oplus\mathcal{S}|$ respectively. The F1 score is the harmonic mean of the precision and recall.

We used the PAN dataset. We report the average query latency, precision, recall, and F1 score per pair of query and text. The query-text pairs are constructed as described in Section 5.2. Note that Faerie and the bruteforce method always achieved perfect accuracy, recall, and F1 score. Thus we omit these numbers. Table 2 shows the results. In terms of query latency, the min-hash based methods KMS and OPH outperformed SeedExtension by almost two orders of magnitude, while Faerie and the brute-force method were an order of magnitude slower than SeedExtension. This is because SeedExtension needs to repeatedly cluster seed matches and verify the clusters. Because of the verification step, SeedExtension always achieved perfect precision (i.e., 100%). However, the recall of SeedExtension was significantly lower than those of KMS and OPH. This is because SeedExtension will miss a similar subsequence if the gap of adjacent common tokens between the subsequence and the query is greater than *maxgap*. With the increase of the threshold θ, the recall of SeedExtension raised as true positives became rare. The accuracy of OPH was on a par with that of KMS. Overall, the min-hash based approaches not only achieved lower query latency but also gained higher F1 scores. For example, when $k = 64$ and $\theta = 0.3$, the F1 scores of both KMS and OPH reached 0.79, while it was only 0.635 for SeedExtension. We also evaluated the accuracy and query latency of OPH and KMS by varying the sketch size k. As shown in Table 3, with the increase of k, the precision and recall of both methods steadily improved. This is attributed to the low variance in similarity estimation with a large sketch size. Besides, the query latency of KMS is much higher than that of OPH.

Configuration of k. In practice, configuring the value of k in OPH involves a trade-off between query latency and accuracy. We first focus on the query latency. As k increases, the index size and index time remain almost unchanged, while the query latency increases especially when k approaches the length of the query. For example, as shown in Figure 11(c), it took 35s when $k = 16$ for OPH-seg to process a 128-token query. However, when $k = 64$, the query latency raised to 210s. This is because the inverted list length of an empty OPH min-hash is much longer than that of non-empty OPH min-hash. As k approaches the query length, there are likely more empty OPH min-hashes and it



(a) OWT (θ=0.4, k=64) (b) OWT (θ=0.4, k=64) (c) OWT (θ=0.4, k=64) (d) WIKI (θ=0.4, k=64) (e) WIKI (θ=0.4, k=64) (f) WIKI (θ=0.4, k=64)

Fig. 12. Evaluating scalability.

takes longer to access the inverted lists. Conversely, a higher k improves accuracy. As we can see in Figure 5(a), both false positive rate and false negative rate decrease with k (Figure 5(b) shows the accuracy of k-mins estimation and OPH estimation are almost the same in simulation). It is also illustrated in Table 3 that with k increased from 16 to 256, the precision increased significantly from 0.566 to 0.92, while the recall modestly increased from 0.714 to 0.876. In practice, k ranges from tens and hundreds offers a balanced approach, avoiding high query latency while achieving satisfactory accuracy. In summary, with the increase of k, the index size, index time, and query latency would grow, while the false positive/false negative rates would decrease. Typically, the value of k ranges from tens to hundreds.

Anecdotal Results. Here is a query in the WIKI dataset: "*The book was not completed at the time of Jordan's death on September 16, 2007. His widow Harriet McDougal and his publisher, Tor Books president Tom Doherty made the decision to have the book completed posthumously, with McDougal saying, 'I am sad to see the series end. But I would be far more distressed to leave it unfinished, incomplete and dangling forever.'_NEWLINE_On December 11, 2007, four months after Jordan's death, it was announced that McDougal had chosen Brandon Sanderson to*". *With* $\theta = 0.35$, `OPH-seg` *identifies the following near-duplicate subsequence in a text: "The original book was incomplete at the time of Jordan's death on September 16, 2007, from cardiac amyloidosis; his widow Harriet McDougal and publisher Tom Doherty chose to publish the book posthumously. Tor Books announced that Brandon Sanderson had been chosen to finish writing the book._NEWLINE_The unfinished book was split into three volumes because it was believed a single volume". The result shows* `OPH-seg` *is able to find subsequences in texts similar to queries. The Jaccard similarity of the query and the identified subsequence is 0.42.*

5.4 Scalability

Exp-9: Scalability. We evaluate the scalability of our techniques by varying the number of texts in the two datasets OWT and WIKI. The queries are randomly selected as described in Section 5.2. Figure 12 shows the results. As we can see, the index time, index size and query latency all grew almost linearly with the number of texts in the dataset. For example, for the OWT dataset, increasing the number of texts from 1M to 2M doubled the index time and the index size from 34GB to 67GB and from 385s to 771s, respectively, while increased the query latency from 21s to 41.7s.

6 Related Work

Near-Duplicate Text Detection. Both near-duplicate text alignment [5, 8, 10, 25, 26, 30, 35, 44] and near-duplicate text detection (a.k.a., similarity search) [6, 32, 57, 60, 61] have been extensively studied due to their importance in text processing. However, the two problems are rather different. While near-duplicate text detection focuses on if two entire texts are similar, near-duplicate text alignment concerns the subsequences in the texts. There are both exact algorithms [32, 57] and approximate algorithms [3, 28] for near-duplicate text detection. Locality sensitive hashing (LSH) [3, 4, 15, 21, 22, 27, 28, 34, 49–51] is one of the most well-known techniques for near-duplicate text detection. However, it is non-trivial to apply LSH for near-duplicate text alignment.

Near-Duplicate Text Alignment. Because of the high computation cost of near-duplicate text alignment, existing methods (including many in bioinformatics [2, 12, 16, 38, 48, 52, 54]) usually rely on sophisticated rule-based heuristics and follow the seeding-extension-filtering strategy [5, 8, 10, 25, 26, 30, 35, 44]. They start by identifying seed matches in the texts, which are short subsequences such as fingerprints [40], k-mers [19], fixed-length windows [56], q-grams [43], or sentences [42]. Next, the seed matches are extended on both ends to form candidate alignments. In addition, candidate alignments are merged if they occur adjacently with a small gap between them. Finally, candidate alignments that do not meet certain criteria (e.g., extremely short or overlapping alignments) are filtered. However, the seeding-extension-filtering heuristic involves numerous hard-to-tune hyper-parameters and lacks of any guarantees [1]. For instance, if the seeds are too coarse (i.e., seed length is too long), it may miss many results, whereas if the seeds are too fine-grained, it may generate an overwhelming number of seed matches that can affect the subsequent extension and filtering steps. Methods based on bloom-filter and suffix array cannot backtrack sources or handle noises [36, 55].

Min-Hash. Brewer et al. propose min-hash as a technique for coordinated sampling [7]. Flajolet and Martin develop probabilistic counting algorithms based on min-hash for database applications [18]. Broder et al. employ min-hash for near-duplicate webpage detection [10], while Cohen uses it to estimate set relations [13]. It takes $O(nk)$ to generate a sketch with k min-hash for a text with n tokens. To reduce the sketch generation time, various improvements have been proposed, such as the bottom-k sketch [53], one permutation hashing (OPH) [33], and fast similarity sketch [14]. Specifically, bottom-k uses the k smallest token hash values in the text as the sketch, which takes $O(n \log k)$ time [53]. Wang et al. develop a technique to losslessly compress the $O(n^2)$ bottom-k sketches in a text with n tokens into compact windows using $O(nk^2)$ space [58]. OPH uses a single random hash function to generate multiple min-hash and takes $O(n+k)$ [33]. However, the number of min-hash is no longer fixed in OPH. To address this issue, a few works propose to densify the OPH min-hash sketch [45–47]. Fast similarity sketch generates a sketch with k min-hash using $O(k \log k + n)$ time, which achieves the best of both world, i.e., a short sketch generation time and a fixed number of min-hash per sketch. However, we find that it does not bring any benefit in compact window generation in terms of index size.

7 Conclusion

In this paper, we study the near-duplicate text alignment problem. Our solution first generates the min-hash sketch of every subsequence in each text and then finds all the subsequences whose min-hash sketches are similar to that of the query. A few recent studies propose to group the nearby subsequences sharing the same min-hash into compact windows and prove that all the $O(n^2k)$ min-hash can be losslessly compressed into $O(nk)$ compact windows using $O(nk)$ space, where n is the number of tokens in the text and k is the sketch size. We propose to use One Permutation Hashing (OPH) to generate the min-hash sketches and introduce the concept of "OPH compact windows". We show the $O(n^2k)$ min-hash generated by OPH can be losslessly compressed in $O(n+k)$ OPH compact windows using $O(n+k)$ space. Furthermore, we develop an algorithm that efficiently finds all the sketches in a text similar to that of the query directly from the OPH compact windows, along with three optimizations.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. This material is based upon work supported by the National Science Foundation under Grants No. 2152908 and 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] Alexandr Andoni and Piotr Indyk. 2006. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. In *FOCS*. 459–468.

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

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

[6] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling up all pairs similarity search. In *WWW*. 131–140.

[7] 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.

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

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

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

[11] 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

[12] Maria Chatzou, Cedrik Magis, Jia-Ming Chang, Carsten Kemena, Giovanni Bussotti, Ionas Erb, and Cedric Notredame. 2016. Multiple sequence alignment modeling: methods and applications. *Briefings in bioinformatics* 17, 6 (2016), 1009–1023.

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

[14] 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.

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

[16] Da-Fei Feng and Russell F Doolittle. 1987. Progressive sequence alignment as a prerequisitetto correct phylogenetic trees. *Journal of molecular evolution* 25, 4 (1987), 351–360.

[17] 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

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

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

[24] Mandy Guo, Zihang Dai, Denny Vrandečić, and Rami Al-Rfou. 2020. Wiki-40b: Multilingual language model dataset. In *Proceedings of the 12th Language Resources and Evaluation Conference*. 2440–2452.

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

[26] 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

[27] 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.

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

[29] Bernard J Jansen. 2006. Search log analysis: What it is, what's been done, how to do it. *Library & information science research* 28, 3 (2006), 407–432.

[30] 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.

[31] Guoliang Li, Dong Deng, and Jianhua Feng. 2011. Faerie: efficient filtering algorithms for approximate dictionary-based entity extraction. In *SIGMOD Conference*. 529–540.

[32] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. 2011. PASS-JOIN: A Partition-based Method for Similarity Joins. *PVLDB* 5, 3 (2011), 253–264.

[33] Ping Li, Art B. Owen, and Cun-Hui Zhang. 2012. One Permutation Hashing. In *NIPS*. 3122–3130.

[34] 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.

[35] Udi Manber. 1994. Finding Similar Files in a Large File System. In *USENIX Winter 1994 Technical Conference*. USENIX Association, 1–10.

[36] Marc Marone and Benjamin Van Durme. 2023. Data Portraits: Recording Foundation Model Training Data. https://doi.org/10.48550/ARXIV.2303.03919

[37] Chris Mayor, Michael Brudno, Jody R Schwartz, Alexander Poliakov, Edward M Rubin, Kelly A Frazer, Lior S Pachter, and Inna Dubchak. 2000. VISTA: visualizing global DNA sequence alignments of arbitrary length. *Bioinformatics* 16, 11 (2000), 1046–1047.

[38] Saul B Needleman and Christian D Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology* 48, 3 (1970), 443–453.

[39] 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

[40] 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.

[41] 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.

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

[46] 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.

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

[48] Temple F Smith, Michael S Waterman, et al. 1981. Identification of common molecular subsequences. *Journal of molecular biology* 147, 1 (1981), 195–197.

[49] 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.

[50] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2009. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*. 563–576.

[51] 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.

[52] Julie D Thompson, Desmond G Higgins, and Toby J Gibson. 1994. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic acids research* 22, 22 (1994), 4673–4680.

[53] 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

[54] Richard Van Noorden, Brendan Maher, and Regina Nuzzo. 2014. The top 100 papers. *Nature News* 514, 7524 (2014), 550.

[55] Thuy-Trang Vu, Xuanli He, Gholamreza Haffari, and Ehsan Shareghi. 2023. Koala: An Index for Quantifying Overlaps with Pre-training Corpora. *arXiv preprint arXiv:2303.14770* (2023).

[56] 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.

[57] Xubo Wang, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. 2017. Leveraging Set Relations in Exact Set Similarity Join. *PVLDB* 10, 9 (2017), 925–936.
[58] 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
[59] 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*.
[60] Chuan Xiao, Wei Wang, and Xuemin Lin. 2008. Ed-Join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB* 1, 1 (2008), 933–944.
[61] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. 2011. Efficient similarity joins for near-duplicate detection. *ACM Trans. Database Syst.* 36, 3 (2011), 15.

Received January 2024; revised April 2024; accepted May 2024