## 1 Response to Meta-Reviews

**M1: About Experiments.**

**M1.1 [R1O1, R1D1]:** *Overall query speedup, which is the most important metric, shows modest gains over existing methods. For example, Figure 7(e) (f) (k) (l) show that the speed up is limited to 1-3x, where 3x speedup is achieved with extremely large text (with length $10^6$). It is unclear if such long texts are common in practice, for the 3x speedup to be realized, likely weakening the benefit of the proposed algorithm in practice.*

**Response.** It can be seen that in the multi-set Jaccard setting, our query time consistently outperforms `AllAlign`, achieving up to a 3× speedup; the speedup increases approximately linearly with the text length $n$. This speedup is important for long-input alignment, which is realistic and widely used in practice, such as log analysis, LLM memorization, data contamination, DNA/genome alignment. Even when the original inputs consist of short strings, it is common to concatenate multiple sequences into a single long text for large-scale retrieval and alignment tasks [25, 35].

Moreover, for practitioners using `AllAlign`, a major challenge is the lack of complexity analysis: when processing a query, `AllAlign` may finish in the next minute or may run indefinitely. This unpredictability is undesirable in practice. In contrast, `MonoActive` has theoretical guarantees, providing a clear expectation of query time.

**M1.2 [R2W1, R2D4]:** *The paper completely disregards the effectiveness of the examined solutions. Is MonoActive always providing the correct responses? That is, recall, precision and F-Measure are always 100%? The paper presenting AllAlign ([15]) does not assume such an exact solution, but evaluates all these effectiveness measures.*

**Response.** Note that Definition 1 aims at reporting all the subsequences of the corpus whose *estimated* (under min-hash) jaccard similarity exceeds the threshold $\theta$. Under this definition, `MonoActive` indeed guarantees 100% precision and recall. In other words, `MonoActive` reports exclusively all the subsequences whose *estimated* Jaccard similarity with the query $\geq \theta$.

When using the actual Jaccard similarity to select the ground truth, the effectiveness is controlled by the number $k$ of hash functions in the min-hash. For any $\epsilon > 0$, choosing $k = O(1/\epsilon^2)$ ensures that the expected error of the estimate Jaccard similarity is at most $\epsilon$. We show in technical report [59] the precision, recall and F1-score of `MonoActive` when $k = 256$ compared to the ground-truth selected with actual Jaccard similarity.

When using the annotations of a dataset PAN to generate ground-truth, we compare the effectiveness of `MonoActive` against the baseline `SeedExtension` in Table 4. Because `AllAlign` generates the same set of results as `MonoActive`, we do not show `AllAlign` in Table 4.

**M1.3 [R2W2, R3W1, R3D5]:** *The experimental analysis is not very convincing in the sense that it includes a single baseline approach and just two datasets. It would be nice if the authors extended their experimental analysis with the dataset used by* `AllAlign`*, with the baseline methods AllAlign considers and with the effectiveness evaluation measures.*

**Response.** We thank the reviewer for the suggestion to strengthen the experimental analysis. Specifically, we have added an additional News dataset, which is also used in `AllAlign`. We have included the baseline method considered by `AllAlign`, namely Seeding-Extension-Filtering (`SeedExtension`). We have also added effectiveness evaluation measures, including recall, precision, and F1-score, please see our responses to M1.2 for detail.

**M1.5 [R2D2]:** *The motivation of this work (and of AllAlign) is the need for computing weighted Jaccard similarities. However, there is no experiments that justifies the benefits of weighted Jaccard over common Jaccard similarity. The experiment in Figures 6(a)-6(f) could approximate this experiment, but exclusively consider the time efficiency of the two similarity measures.*

**Response.** Thank you for the comment. We agree that the original experiments primarily focused on time efficiency and did not directly justify the effectiveness advantage of weighted Jaccard similarity over the common Jaccard similarity. To address this concern, we have added a new experiment in Section 6.4 that explicitly compares the effectiveness of `MonoActive` under weighted Jaccard (e.g., multi-set Jaccard) with `MonoActive` under common Jaccard. On this dataset, the best effectiveness is achieved by multi-set Jaccard; results for other weighting functions are reported in the technical report [59]. Note that different datasets may favor different weighting functions.

Weighted Jaccard similarity has been widely adopted in prior work (e.g., [31]) because it better captures token importance than the unweighted Jaccard similarity. This observation motivates both AllAlign and our work, and is generally accepted in the literature.

The primary contribution of this paper is not to re-establish the superiority of weighted Jaccard over unweighted Jaccard, but to propose a new algorithmic framework that efficiently supports weighted Jaccard similarity. The newly added experimental results (Table 4) show that appropriate weighting schemes can yield a better precision–recall trade-off and significantly higher F1 scores. Moreover, we observe that `MonoActive` also improves effectiveness even in the unweighted setting compared with prior approaches. We have clarified this motivation and positioning in the revised manuscript.

**M1.6 [R2D7]:** *It would be nice to add a table with the technical characteristics of the datasets used in the experimental evaluation. Please also report the number of queries per dataset or per experiment from each dataset.*

**Response.** We have added a table summarizing the dataset sizes, the average text lengths and the number of queries. For each dataset, the number of queries is fixed and remains the same across all experiments.

**M1.7 [R2D8]:** *Please explain how the baseline method,* `AllAlign`*, was implemented and configured. Did you implemented yourselves or you used an existing implementation by its authors?*

**Response.** We implemented `AllAlign` and `MonoActive` within the same framework, ensuring a fully fair comparison. For the `SeedExtension` baseline, we used the open-source implementation based in python (https://github.com/CubasMike/plagiarism_detection_pan2015).

**M1.8 [R2D9]:** *Page 9 states that: "Texts with fewer than n tokens were concatenated until they reached or exceeded n tokens, after which they were truncated to the first n tokens as a single text." Why is this necessary? Is this the only approach?*

**Response.** The purpose of this is to explicitly control the text length $n$. Concatenating and truncating provides a straightforward and intuitive way to generate inputs of exact length $n$ while largely preserving the original token distribution. This is not the only possible approach. Alternatives include padding shorter texts or allowing variable-length inputs. However, these alternatives either do not allow precise control of $n$ or introduce artificial tokens. We have clarified this motivation in the revised version.

**M1.9 [R2D10]:** *The discussion of the experimental comparison between AllAlign and MonoActive is a bit superficial, focusing on the observed performance patterns. It would be nice to provide more insights into the relative performance of the two methods.*

**Response.** We have revised the experimental analysis to provide deeper insights into the relative performance of `AllAlign` and `MonoActive`. First, we consolidate the absolute performance metrics (e.g., partition size and time) together with the corresponding reduction or speedup curves into the same figures, which allows a direct and clearer comparison between the two methods. Second, when the $x$-axis is plotted on a logarithmic scale, we also use a logarithmic scale on the $y$-axis to better reveal the scaling behavior. With this log–log visualization, we observe that the query-time speedup of `MonoActive` over `AllAlign` improves approximately linearly as the text length $n$ increases, providing a more informative insight of their relative performance beyond raw performance trends.

**M1.10 [R2W2, R3W1, R3D5]:** *The experimental baseline selection is insufficient. Apart from AllAlign, the paper fails to compare with more recent state-of-the-art approximate matching methods, which limits the comprehensiveness of the evaluation. The authors are suggested to discuss these studies and their efforts to find such studies (if any).*

**Response.** Please see our responses to M1.3.

**M2: About Theoretical Analysis.**

**M2.1 [R1O2, R1D2]:** *The main theoretical guarantee of the algorithm assumes a simplified raw count term frequency. Can this result be generalized to other classes of weight functions?*

**Response.** The theoretical guarantee is not restricted to the raw count term frequency. We use raw count weights primarily for clarity of exposition. The analysis in fact applies to a broader class of weight functions that satisfy the AoW assumption introduced in Section 5. We have made this clarification explicit in the revised version.

**M2.2 [R3D1]:** *Section 5 introduces ICWS and assumes AoW, but lacks rigorous propositions/proofs to validate MonoActive's correctness in weighted scenarios. Recommend supplementing formal proofs in the revised manuscript.*

**Response.** In the revision, we made the correctness argument for the weighted setting explicit. Specifically, we showed that under the AoW assumption, the weighted setting induces the same hash value set structure as in the unweighted case. We removed the ambiguous informal discussion of correctness and instead added a dedicated correctness theorem in Section 5. The proof establishes that the weighted case is structurally identical to the unweighted one, and that the original correctness argument carries over as long as the hash value set is well defined.

**M2.3 [R3W3, R3D2]:** *Duplicate text alignment algorithms are typically deployed on large-scale data. The paper focuses on expected partition size in theoretical analysis but overlooks worst-case performance boundaries. Suggest discussing or experimentally validating these boundaries for comprehensive evaluation.*

**Response.** We clarify that the parameter $f_\Gamma$ in our complexity analysis explicitly captures the worst-case characteristics of the input. Therefore the input-side worst-case behavior is already bounded through $f_\Gamma$.

The source of randomness in the complexity analysis comes from the hash functions. This is precisely why our guarantees are stated in expectation. The complexity may degrade only when the hash functions fail, which occurs with negligible probability.

We discussed this more explicitly in the revised manuscript.

**M3: About Motivation.**

**M3.1 [R2D1]:** *In Section 1, it is not clear why a text with n tokens yields $O(n^2)$ subsequences. Please add references and justify this complexity.*

**Response.** We have clarified in Section 1 why a text of length $n$ yields $O(n^2)$ subsequences by explicitly noting that each subsequence is uniquely determined by its start and end positions, which results in $\Theta(n^2)$ possibilities.

**M3.2 [R2D11]:** *Is Section 3 novel or based on AllAlign, given that the compact window is already proposed in [15]?*

**Response.** We clarify that Section 3 is not claimed as a novel contribution, but is intended purely as background. To avoid any confusion, we have renamed its title to Background in the revised version. This section reviews and formalizes the compact window concept from AllAlign [15] to provide the necessary context for our new algorithms and analysis.

## 2 Response to Individual Reviews

**R1O1:** *Experiments seem to show limited overall query speedup.*

**Response.** Please see our responses to M1.1.

**R1O2:** *Assumptions used in theoretical guarantees.*

**Response** Please see our responses to M2.1.

**R1D1:** *Overall query speedup, which is the most important metric in my view, shows modest gains over existing methods. For example, Figure 7(e) (f) (k) (l) show that the speed up is limited to 1-3x, where 3x speedup is achieved with extremely large text (with length $10^6$). It is unclear if such long texts are common in practice, for the 3x speedup to be realized, likely weakening the benefit of the proposed algorithm in practice.*

**Response.** Please see our responses to M1.1.

**R1D2:** *The main theoretical guarantee of the algorithm assumes a simplified raw count term frequency. Generalizing this result to other classes of weight functions would further strengthen the paper.*

**Response.** Please see our responses to M2.1.

**R2W1:** The paper completely disregards the effectiveness of the examined solutions. Is MonActive always providing the correct responses? That is, recall, precision and F-Measure are always 100%? The paper presenting AllAlign ([15]) does not assume such an exact solution, but evaluates all these effectiveness measures.

**Response.** Please see our responses to M1.2.

**R2W2:** *The experimental analysis is not very convincing in the sense that it includes a single baseline approach and just two datasets. It would be nice if the authors extended their experimental analysis with the dataset used by AllAlign, with the baseline methods AllAlign considers and with the effectiveness evaluation measures.*

**Response.** Please see our responses to M1.3.

**R2W3:** *Improvements are needed in the presentation, as explained below in more detail.*

**R2D1:** *In Section 1, it is not clear why a text with n tokens yields $O(n^2)$ subsequences. Please add references and justify this complexity. In case of lack of space, you could move this explanation to the online extended version.*

**Resposne.** Please see our responses to M3.1.

**R2D2:** *The motivation of this work (and of AllAlign) is the need for computing weighted Jaccard similarities. However, there is no experiments that justifies the benefits of weighted Jaccard over common Jaccard similarity. The experiment in Figures 6(a)-6(f) could approximate this experiment, but exclusively consider the time efficiency of the two similarity measures.*

**Response.** Please see our responses to M1.5.

**R2D3:** *Figure 1 is mentioned page 2 but is found in page 4. You should move it closer to its first mention.*

**Response.** Thank you for pointing this out. We have moved Figure 1 closer to its first mention in the text to improve clarity and readability.

**R2D4:** *In Section 2.3, please explain whether this problem has an exact solution, meaning that all correct pairs are always identifies, or an approximate one, meaning that false positives and false negatives are possible. In the former case, solutions are only compared against time efficiency and scalability, while in the latter, the comparison includes effectiveness measures, too.*

**Response.** Please see our responses to M1.2.

**R2D5:** *Please refer to line numbers in the discussion of Algorithms 1-5.*

**Response.** Thank you for the suggestion. We have updated the corresponding discussions of Algorithms 1-5 to explicitly refer to their line numbers.

**R2D6:** *In Line 7, Algorithm 3 contains K, which is not initialized before. Please add an initialization command (like Line 1), adding a comment about the role of K.*

**Response.** Thank you for pointing this out. We have added an initialization of $K$ at the beginning of Algorithm 3 and included a brief comment explaining its role.

**R2D7:** *It would be nice to add a table with the technical characteristics of the datasets used in the experimental evaluation. Please also report the number of queries per dataset or per experiment from each dataset.*

**Response.** Please see our responses to M1.6.

**R2D8:** *Please explain how the baseline method, AllAlign, was implemented and configured. Did you implemented yourselves or you used an existing implementation by its authors?*

**Response.** Please see our responses to M1.7.

**R2D9:** *Page 9 states that: "Texts with fewer than n tokens were concatenated until they reached or exceeded n tokens, after which they were truncated to the first n tokens as a single text." Why is this necessary? Is this the only approach.*

**Response.** Please see our responses to M1.8.

**R2D10:** *The discussion of the experimental comparison between AllAlign and MonoActive is a bit superficial, focusing on the observed performance patterns. It would be nice to provide more insights into the relative performance of the two methods.*

**Response.** Please see our responses to M1.9.

**R2D11:** *Is Section 3 novel or based on AllAlign, given that the compact window is already proposed in [15]?*

**Response.** Please see our responses to M3.2.

**R2D12:** *Minor comments: * Page 2: "while represent" -> "while representing"? * Page 8: "In a nutshell, it designs a" -> "In a nutshell, one design a" * Page 9: "A few example" -> "A few examples"*

**Response.** We have corrected all listed minor wording and grammatical errors at the indicated locations in the revised manuscript.

**R3W1:** *The experimental baseline selection is insufficient. Apart from AllAlign, the paper fails to compare with more recent state-of-the-art approximate matching methods, which limits the comprehensiveness of the evaluation.*

**Response.** Please see our responses to M1.3.

**R3W2:** *The sections on MonoActive's skyline update and ActiveKeys generation lack intuitive explanations or clear visual illustrations, reducing readability and accessibility for readers.*

**Response.** We would like to clarify that the manuscript already contains illustrative examples for the skyline update and ActiveKeys generation. Specifically, Figure 4 and Example 9 together provide both a visual and step-by-step explanation of how the skyline and ActiveKeys update. We hope that these examples help address the reviewer's concern regarding intuition and clarity.

**R3W3:** *While duplicate text alignment algorithms are typically applied to large-scale data, the theoretical analysis in this paper focuses primarily on partition size under expected conditions. The worst-case performance boundaries are equally critical and should be discussed or experimentally validated to provide a complete performance assessment.*

**Response.** Please see our responses to M2.3.

**R3D1:** *Section 5 introduces ICWS and assumes AoW, but lacks rigorous propositions/proofs to validate MonoActive's correctness in weighted scenarios. Recommend supplementing formal proofs in the revised manuscript.*

**Response.** Please see our responses to M2.2.

**R3D2:** *Duplicate text alignment algorithms are typically deployed on large-scale data. The paper focuses on expected partition size in theoretical analysis but overlooks worst-case performance boundaries. Suggest discussing or experimentally validating these boundaries for comprehensive evaluation.*

**Response.** Please see our responses to M2.3.

**R3D3:** *Multiple symbols* $(n, f, k, \theta)$ *and concepts (active keys, compact windows, skyline) are inconsistently defined/used across chapters. Recommend compiling a unified symbol table in Preliminaries or a dedicated section to enhance readability.*

**Response.**We have added a notation table in the preliminaries and carefully revised the manuscript to ensure consistent definitions and usage across sections. In particular, in the experimental section, we replace the symbol $f$ previously used to denote the maximum token frequency with the notation $f_{\mathrm{T}}$, which is defined in the preliminaries to avoid ambiguity and maintain consistency.

**R3D4:** *Key examples (e.g., skyline in Section 4) require more diagrammatic support and step-by-step illustrations to improve clarity.*

**Resposne.** Please see our responses to R3W2.

**R3D5:** *The baseline selection should include additional state-of-the-art duplicate text alignment algorithms to strengthen comparative evaluation.*

**Response.** Please see our responses to M1.3.

# Near-Duplicate Text Alignment under Weighted Jaccard Similarity

Yuheng Zhang[1] Miao Qiao[2] Zhencan Peng[1] Dong Deng[1*]
[1]Rutgers University [2]University of Auckland
{yuheng.zhang,zhencan.peng,dong.deng}@cs.rutgers.edu,miao.qiao@auckland.edu

## Abstract

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

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

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

## 1 Introduction

This paper studies the near-duplicate text alignment problem [15, 37, 38, 41, 42, 56]. Given a collection of data texts, the task takes a short query text and returns all subsequences (i.e., substrings) of the data texts that are similar to the query. Near-duplicate text alignment has become increasingly important in the era of large language models (LLMs), with key applications in test set leakage (also known as data contamination) detection [32], training data deduplication [29], and memorization analysis [9, 54]. Beyond LLMs, this problem also plays a crucial role in domains such as bioinformatics [2], log analysis [13], and plagiarism detection [39, 43].

Due to the high computational cost of near-duplicate text alignment, previous methods often adopt the seeding–extension–filtering heuristic [3, 5, 7, 23, 24, 28, 34, 39, 46, 47, 55]. However, these methods lack accuracy guarantees and often involve many hard-to-tune hyperparameters [17]. To address these limitations, recent studies have proposed to use min-hash techniques [6, 30] for near-duplicate text alignment. These approaches guarantee to retrieve all subsequences whose estimated Jaccard similarities with the query exceed a user-defined threshold [15, 37, 38, 56]. However, a key limitation is that they only support the unweighted Jaccard similarity, which treats all tokens equally—regardless of their frequency or importance (note depending on the tokenizer, a token can be a word [57], a $q$-gram [18], a byte-pair-encoding [19], etc.).

To highlight the issue of unweighted Jaccard similarity, consider Q = *AAAAAATTTTTCCCCCC*, T = *AAAAAATTTTGCCCCCC*, and S = *AATTGCC*. Intuitively, Q is much more similar to T than to S. Yet, under unweighted Jaccard similarity (with each token as a 2-gram), both Q and T, as well as Q and S yield the same similarity score: 4/7. In contrast, weighted Jaccard similarity correctly reflects the difference: when each token in a text is weighted by its frequency in the text, Q and T have a similarity of 0.8, while Q and S only score 0.2. Moreover, unweighted Jaccard similarity fails to distinguish between stop words and content words. For example, consider Q = "*I read about Einstein in a book*", T = "*I studied Einstein through a book*", S = "*I roamed about in a castle*". Semantically, Q is clearly closer to T than to S. However, the unweighted Jaccard similarity is the same for both: 4/9 (with each token as a word). If a meaningful token weighting scheme such as TF-IDF [52] is applied, then stop words, e.g., *I, in, a, about, through*, could have near-zero weight and the other words may have near-one weight, and thereby drawing clear distinctions. Under this scheme, the weighted Jaccard similarity between Q and T is around 0.5, while that between Q and S drops to nearly zero.

**Existing Methods.** Existing methods propose to group, as the indexing process, the subsequences in each data text based on their min-hash values. This way, when a query arrives, they use the index to identify all the min-hash sketches that are similar to the query's min-hash sketch and then return the corresponding subsequences as the results efficiently [15, 37, 38]. A key observation made by these methods is that the nearby subsequences of a text tend to share the same min-hash (this is because appending a token to a subsequence most likely would not change the min-hash of the subsequence). Given a text with $n$ tokens, they group the $O(n^2)$ subsequences in the text (since each subsequence is defined by a valid pair of start and end positions, the number of such valid position pairs is $\sum_{i=1}^{n}(n - i + 1) = O(n^2)$ [22]) into $O(n)$ groups while representing each group with a tuple of $O(1)$ space.

*Dong Deng is the corresponding author.

**Limitations of Existing Methods.** In many real-world scenarios, tokens are associated with weights, such as TF-IDF (term frequency-inverse document frequency) weights [36]. Estimating weighted Jaccard similarity requires consistent weighted sampling (CWS) [26], as standard min-hash[6] applies only to the unweighted case. Existing methods do not support CWS or weighted Jaccard similarity. The only exception is `AllAlign` [15] which supports multi-set Jaccard, a special case of weighted Jaccard where each token's weight is its frequency in the text. `AllAlign` is a greedy algorithm which groups the subsequences by their "multi-set min-hash". However, it lacks a formal complexity analysis, including an upper bound on the number of groups generated and its time complexity. Due to its recursive nature, analyzing its complexity is particularly difficult.

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

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

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

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

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

**Table 1: Summary of symbols and notations.**

## 2 Preliminaries

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

### 2.1 Multi-Set Jaccard Similarity

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

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

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

### 2.2 Min-Hash for Multi-Set Jaccard Similarity

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

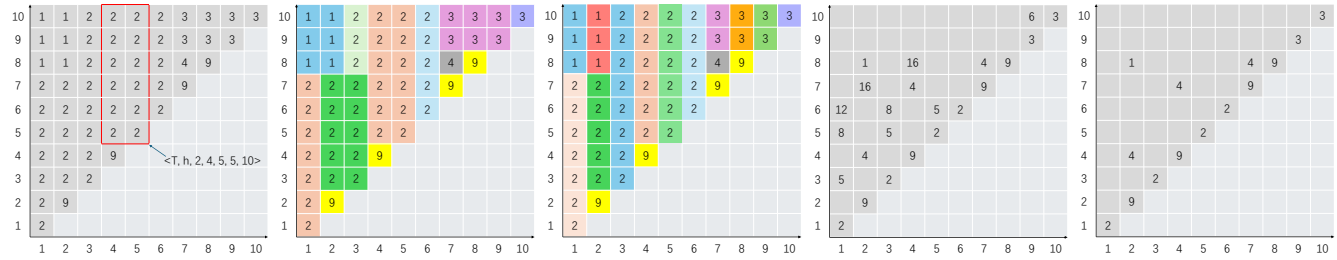
$$h(\mathrm{T}) = \min\{h(t, x) \mid t \in \mathrm{T}, 1 \le x \le f(t, \mathrm{T})\}. \tag{1}$$

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

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

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

This is because there are $\Sigma_{t \in \mathrm{T} \cup \mathrm{S}} \max(f(t, \mathrm{T}), f(t, \mathrm{S}))$ unique hash values and $\Sigma_{t \in \mathrm{T} \cup \mathrm{S}} \min(f(t, \mathrm{T}), f(t, \mathrm{S}))$ common hash values in the




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

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

two texts T and S. T and S share the same min-hash if and only if the smallest hash value of all unique hash values is among their common hash values. The probability of the latter is exactly $J_{T,S}$.

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

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

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

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

### 2.3 Near-Duplicate Text Alignment

Near-duplicate text alignment is formally defined as below.

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

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

## 3 Background

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

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

### 3.1 Compact Window and Partition

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

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

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

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

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

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

- *Disjointness: For any two compact windows* $W_x$ *and* $W_y$ *in* $\mathcal{P}(T,h)$*,*
$$([a_x, b_x] \times [c_x, d_x]) \cap ([a_y, b_y] \times [c_y, d_y]) = \phi.$$
- *Coverage: The union of all compact windows in* $\mathcal{P}(T,h)$ *covers all the subsequences in* T*, i.e.,*
$$\{(i,j) \mid 1 \leq i \leq j \leq |T|\} \subseteq \cup_{x=1}^{l} ([a_x, b_x] \times [c_x, d_x]).$$

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

### 3.2 Indexing and Query Processing

Algorithm 1 shows the pseudo-code of indexing in our framework. It takes a collection D of data texts and an integer $k$ as input and produces $k$ inverted indexes of compact windows. For this purpose,

it first randomly selects $k$ independent hash functions $h_1, h_2, \cdots, h_k$ (Line 1). Then, for each text T in D, and each hash function $h_i$, it generates a partition $\mathcal{P}(\mathrm{T}, h_i)$ (Line 4). The partition generation algorithm will be described in the next section. For each compact window $\langle \mathrm{T}, h_i, v, a, b, c, d \rangle$ in the partition, it is appended to the inverted list $I_i[v]$ (Line 6). Finally, $k$ inverted indexes $I_1, \cdots, I_k$ are returned (Line 7).

**Algorithm 1**: Indexing(D, $k$)

**Input**: D: a collection of data texts; $k$: an integer.
**Output**: $I_1, I_2, \cdots, I_k$: $k$ inverted indexes.
1 randomly select $k$ hash functions $h_1, h_2, \cdots, h_k$ from $\mathcal{H}$;
2 **foreach** T ∈ D **do**
3  **foreach** $i \in [1, k]$ **do**
4  $\mathcal{P} \leftarrow$ PartitionGeneration(T, $h_i$);
5  **foreach** $\langle \mathrm{T}, h_i, v, a, b, c, d \rangle \in \mathcal{P}$ **do**
6  append $\langle \mathrm{T}, h_i, v, a, b, c, d \rangle$ to $I_i[v]$;
7 **return** $I_1, I_2, \cdots, I_k$ ;

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

**Input**: Q: a query text; $\theta$: a similarity threshold; $k$: an integer; $h_1, \cdots, h_k$: hash functions; $I_1, \cdots, I_k$: inverted indexes.
**Output**: All subsequences T$[x, y]$ in D where $\hat{\mathrm{J}}_{\mathrm{Q,T}[x,y]} \geq \theta$.
1 **foreach** $i \in [1, k]$ **do** $v_i \leftarrow h_i(\mathrm{Q})$;
2 **return** PlaneSweep($k, \theta, I_1[v_1], \cdots, I_k[v_k]$);

Algorithm 2 presents the pseudo-code of query processing in our framework. The input consists of a query text Q, a multi-set Jaccard similarity threshold $\theta$, the integer $k$, the same $k$ random hash functions $h_1, h_2, \ldots, h_k$ used during indexing, and the $k$ inverted indexes $I_1, \cdots, I_k$. It first calculates the $k$ min-hash $v_1, \cdots, v_k$ of Q (Line 1) and then retrieves the $k$ corresponding inverted lists $I_1[v_1], \cdots, I_k[v_k]$ from the $k$ inverted indexes. The near-duplicate subsequences can be identified by a simple plane sweep algorithm over the compact windows in the $k$ inverted lists (Line 2). Specifically, the plane sweep algorithm produces all the cells $(x, y)$ where there are at least $\lceil k\theta \rceil$ compact windows $\langle \mathrm{T}, h, v, a, b, c, d \rangle$ in the $k$ inverted lists satisfying $(x, y) \in [a, b] \times [c, d]$. This is because the estimated multi-set Jaccard similarity $\hat{\mathrm{J}}_{\mathrm{Q,T}[x,y]} \geq \theta$. The simple plane sweep algorithm is detailed in [37], while an optimized version is presented in [38]. Due to space constraints, we omit the details in this paper.

## 4 Partition Generation

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

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

### 4.1 Monotonic Partitioning

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

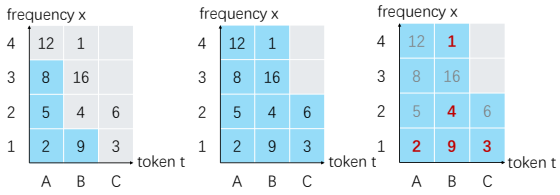
Definition 5 (hash value set). *Given a random universal hash function* $h$*, the hash value set of a subsequence* T$[i, j]$ *is*

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

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

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

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



(a) H(T[3, 6]). (b) H(T). (c)active hash in H(T).

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

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

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

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

Proof. Based on Definition 5, H(T$[i, j]$) ⊆ H(T). Based on Equation 3, $h$(T$[i, j]$)=min H(T$[i, j]$). Thus $h$(T$[i, j]$) ∈ H(T). □

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

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

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

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

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

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

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

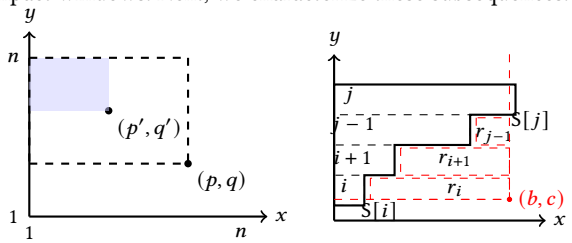
Proof. Based on Definitions 5, 6, and 7, we have $\mathsf{H}(\mathrm{T}[i,j]) = \{h(p,q,\mathrm{T}) \mid (p,q) \in \mathsf{K}(\mathrm{T}[i,j])\}$. Based on Equation 1, we have $h(\mathrm{T}[i,j]) = \min\{h(p,q,\mathrm{T}) \mid (p,q) \in \mathsf{K}(\mathrm{T}[i,j])\}$. □

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

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

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

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




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

**Figure 3: Monotonic partitioning.**

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

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

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

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

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

Lemma 4. *For a key set* L *and its skyline* $\mathsf{S}(\mathsf{L})$*,* $\mathrm{rec}(\mathsf{L}) = \mathrm{rec}(\mathsf{S}(\mathsf{L}))$.

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

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

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

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

## 4.2 The Monotonic Partitioning Algorithm

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

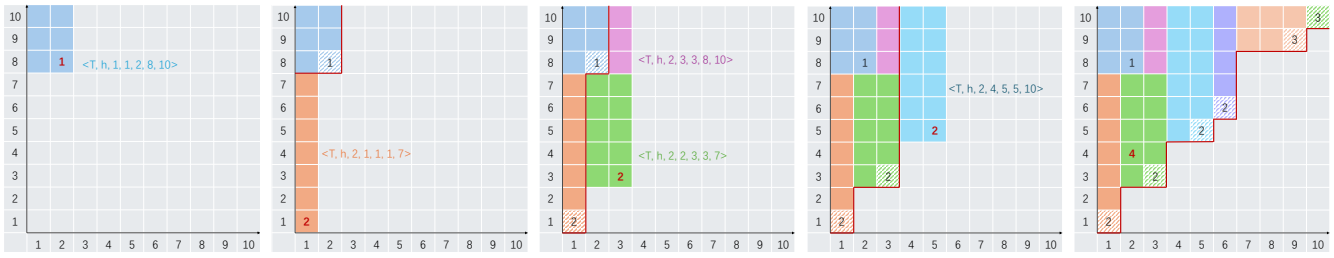
Definition 11 (Coordinate Order). *A skyline* S *is in coordinate order if its keys are ordered lexicographically by their coordinates. We denote* $\mathsf{S}[i]$ *as the* $i$*-th key in* S *under this order.*

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

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

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

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




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

**Figure 4: An example of monotonic partitioning.**

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

**Algorithm 3**: GenerateKeys(T, $h$)

**Input**: T : a text of length $n$; $h$: a universal hash function.
**Output**: K: the set of keys in T.

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

**Algorithm 4**: MonotonicPartitioning(T, $h$)

**Input**: T: a text of length $n$; $h$: a universal hash function.
**Output**: A partition $\mathcal{P}(T, h)$.

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

**Algorithm 5**: GenerateActiveKeys(T, $h$)

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

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

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

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

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

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

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

When $(b, c)$ is not dominated by any key in S, we first find all the keys in S that are dominated by $(b, c)$. In the technical report [59], we prove that with two binary searches on S (Lines 6-7), $S[i + 1]$ to $S[j - 1]$ are exactly the keys in S dominated by $(b, c)$. Lines 8-13 generate up to $j - i$ compact windows (red dashed rectangles in Figure 3(b)). As formally proved in the technical report [59], these compact windows are disjoint and cover all the subsequences $T[i, j]$ where $(i, j) \in \text{rec}(b, c) \setminus \text{rec}(L)$, whose min-hash must be $v$ (by Lemma 3 and Lemma 4). We update S by removing these dominated keys (Line 14) and inserting the new key $(b, c)$ (Line 15).

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

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

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

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

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

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

Due to space limitations, the proof of Theorem 1 is provided in the technical report [59], while the proof of Theorem 2 appears in Section 4.3. We emphasize that the "expectation" in Theorem 2 arises solely from the randomness of the hash functions, rather than from any distributional assumption on the input text T. In particular, the bound holds for arbitrary inputs and explicitly captures the worst-case input characteristics through the parameter $f_{\mathrm{T}}$, the maximum token frequency in T. We also provide a matching lower bound in Theorem 3, establishing the optimality of our analysis with respect to $f_{\mathrm{T}}$, with the proof included in the extended version for interested readers.

## 4.3 Active Keys and Complexity Analysis

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

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

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

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

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

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

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

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

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

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

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

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

Lemma 10. *The total number of compact windows generated by Algorithm 4 is no more than* $2|\mathsf{X}(\mathrm{T})|$*.*

Proof. Consider Lines 9-13. The algorithm produces at most $j-i$ compact windows. Note that inserting the key $(b,c)$ results in the removal of $j-i-1$ other keys from the skyline, and these removed keys will not be reinserted. Thus, among the $j-i$ compact windows generated, one corresponds to the insertion of $(b,c)$ and one to the removal of each key. Thus one compact window is produced upon the insertion of a key and one compact window is produced upon the removal of a key. Moreover, by Lemma 8, only active keys can be added to the skyline. Hence, the total number of compact windows generated by Algorithm 4 is at most $2|\mathsf{X}(\mathrm{T})|$. □

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

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

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

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

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

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

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

Due to space limitations, the proof of the correctness of this optimization is provided in the technical report [59]. Note that there are $O(nf_{\mathrm{T}})$ keys in K(T). Thus, the time and space complexities of vanilla Algorithm 4 (without the active key optimization) are respectively $O(nf_{\mathrm{T}}\log n)$ and $O(nf_{\mathrm{T}})$.

## 5 Generalization to Weighted Jaccard Similarity

Consider a text T where each distinct token $t$ is associated with a weight. The weight is determined by a weight function $w(t,\mathrm{T})$ where $w(t,\mathrm{T}) > 0$ for $t \in \mathrm{T}$ and $w(t,\mathrm{T}) = 0$ for $t \notin \mathrm{T}$. The weighted Jaccard similarity of two texts T and S is defined as $\mathrm{J}^{w}_{\mathrm{T,S}} = \frac{\sum_{t\in\mathrm{T}\cup\mathrm{S}}\min(w(t,\mathrm{T}),w(t,\mathrm{S}))}{\sum_{t\in\mathrm{T}\cup\mathrm{S}}\max(w(t,\mathrm{T}),w(t,\mathrm{S}))}$. To estimate weighted jaccard similarity, one can use **improved consistent weighted sampling** [27]. In a nutshell, one designs a hash family $\mathcal{H}$. Given a text T, for each distinct token $t$ in T, a hash function $h \in \mathcal{H}$ takes the token $t$ and its weight $w(t,\mathrm{T})$ as input and outputs a hash value, denoted as $h(t,w(t,\mathrm{T}))$. With hash function $h$, we define the *weighted min-hash* of a text T as the smallest hash value among all distinct tokens in T, denoted as $h(\mathrm{T},w)$. Formally,

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

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

**Algorithm 6**: Improved Consistent Weighted Sampling [27]

```
1  Class ICWS:
2    def init(self, Σ: the vocabulary of tokens):
3      foreach token t ∈ Σ do
4        self.r_t ~ Gamma(2, 1)
5        self.c_t ~ Gamma(2, 1)
6        self.β_t ~ Uniform(0, 1)
7    def h(t: a token, weight: a positive real value):
8      y = exp(r_t(⌊ln weight / r_t + β_t⌋ − β_t));
9      a = 1/y · c_t / exp(r_t);
10     return HashValue(t, y, a);
11 Class HashValue:
12   def init(self, t, y, a): self.t = t, self.y = y, self.a = a;
13   def operator=(v_1, v_2): return v_1.t = v_2.t and v_1.y = v_2.y;
14   def operator<(v_1, v_2): return v_1.a < v_2.a;
```

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

**Implementation Details.** Algorithm 6 implements the improved consistent weighted sampling scheme [27]. Drawing a random hash function $h \in \mathcal{H}$ is the same as instantiating a new object $o$ of ICWS class (Lines 1-10). The constructor (Lines 2-6) samples three independent random variables for every token $t$ in the vocabulary $\Sigma$ from the ICWS specified distributions. The hash computation (Lines 7-10) $o.h(t,w(t,\mathrm{T}))$ of $o$ returns the hash value based on token $t$ and the weight of $t$ in T. Note that an object is initialized with a random seed; the set of possible objects form the hash family.

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

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

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

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

**Term Frequency–Inverse Document Frequency (TF-IDF)** [52]. TF-IDF is a widely used weighting scheme that captures both the importance of a token within a text and its rarity across a corpus D.

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

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

The TF-IDF weight function is defined as $w(t,\mathrm{T})=\mathrm{tf}\cdot\mathrm{idf}$ where tf is the TF weight and idf is the IDF weight. A few examples TF and IDF weights we support (but not limited to) are listed in Table 2.

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

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

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

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

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

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

By Lemma 12 and Equation 4, we have

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

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

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

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

Proof. Given a text with $n$ tokens and a weight function $w$. Consider a token $t\in\mathrm{T}$. Let $x=f(t,\mathrm{T})$. Clearly, $h(t,1)$ must be an active hash value. For any $i\in[2,x]$, based on the proof of Lemma 12, $h(t,i)$ is active if and only if $h(t,i)<h(t,i-1)$. Let $h(t,i)=(t,y_i,a_i)$ and $h(t,i-1)=(t,y_{i-1},a_{i-1})$. $h(t,i)<h(t,i-1)$ if and only if $y_i\in(w(t,i-1),w(t,i)]$. By uniformity, $y_i$ distributed uniformly over $[0,w(t,i)]$. Thus the probability that $h(t,i)$ is active is $\frac{w(t,i)-w(t,i-1)}{w(t,i)}$. There are $x-i+1=O(x)$ keys corresponding to the hash value $h(t,i)$ (imagine a sliding window of size $i$ over a string of length $x$). Thus in expectation, the number of active keys in T is $\mathbb{E}[\mathsf{X}(\mathrm{T})]=O(\sum_{t\in\mathrm{T}}f(t,\mathrm{T})\sum_{i=1}^{f(t,\mathrm{T})}\frac{w(t,i)-w(t,i-1)}{w(t,i)})$. Note that $w(t,0)=0$. Clearly, for $x\ge1$, when $w(t,x)=x\cdot\mathrm{IDF}$, we have $\mathbb{E}[|\mathsf{X}(\mathrm{T})|]=O(n+n\log f_{\mathrm{T}})$. When $w(t,x)=1\cdot\mathrm{IDF}$, $\mathbb{E}[|\mathsf{X}(\mathrm{T})|]=O(n)$. For $w(t,x)=\log(x+1)\cdot\mathrm{IDF}$, $\mathbb{E}[|\mathsf{X}(\mathrm{T})|]=O(n+n\log\log f_{\mathrm{T}})$ based on taylor expansion. For $w(t,x)=x^2\cdot\mathrm{IDF}$, $\mathbb{E}[|\mathsf{X}(\mathrm{T})|]=O(n+n\log f_{\mathrm{T}})$. □

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

## 6 Experiment

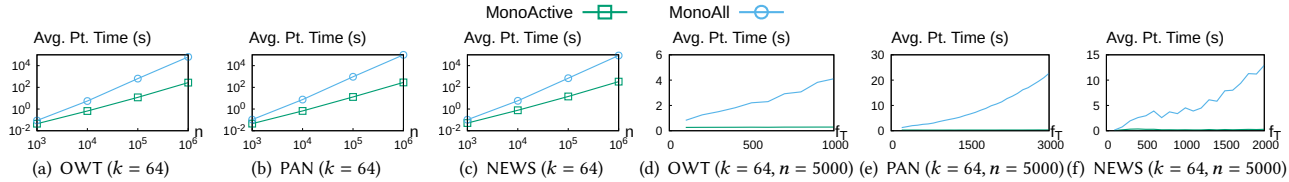
**Environment.** We implemented our proposed method `MonoAll`, `MonoActive` and the `AllAlign` baseline in C++. For the `SeedExtension` baseline, we used the open-source implementation based in python [1]. All C++ programs were compiled with GCC 11.4.0 and optimized using the `-O3` flag. All experiments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

**Datasets.** We conducted experiments on three datasets, PAN [40], OWT [20], and NEWS. All datasets were tokenized using the GPT-2 byte-pair encoding (BPE) tokenizer [44], with a vocabulary size of 50,257. PAN [2] is a benchmark for external plagiarism detection [40]. Each text in PAN is a book. The dataset contains 11,093 texts and 642,380,109 tokens after tokenization. OWT consists of 8 million web texts highly ranked on Reddit. Each document corresponds to a web page. NEWS is a large-scale news corpus collected from publicly available online news sources. Each document corresponds to a news article. Table 3 summarizes the dataset sizes, the average text lengths and the number of queries.

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

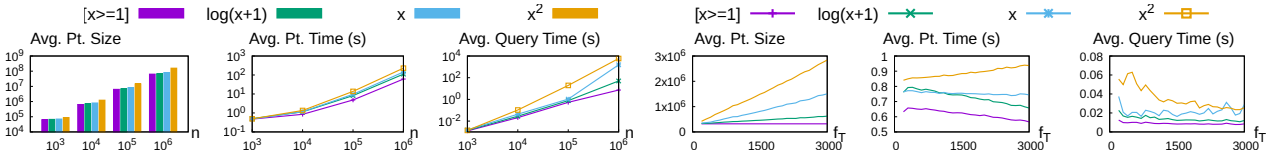
**Table 3: Datasets statistics and the number of queries.**

[1] https://github.com/CubasMike/plagiarism_detection_pan2015
[2] https://pan.webis.de/data.html




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

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




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

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

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

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

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

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

**Compared Methods.** We consider 4 methods in our experiment.

- `MonoAll`: Our vanilla monotonic partitioning, Algorithm 4.
- `MonoActive`: Our monotonic partitioning with active hash optimization (Section 4.3). It calls Algorithm 5 to generate keys.
- `AllAlign`: A greedy partitioning algorithm for multi-set Jaccard similarity [15]. `AllAlign` generates compact windows in recursion. In each iteration, it takes a rectangle of the shape $[a, b] \times [a, c]$ as input and partitions all the subsequences $T[i, j]$ in this rectangle into a few compact windows and one or more smaller rectangles of that shape. The smaller rectangles are recursively partitioned until no rectangles left. At the beginning, the input rectangle is $[1, n] \times [1, n]$. However, `AllAlign` is greedy and its time and space complexities are unknown.
- `SeedExtension`: A widely used approach in plagiarism detection. It enumerates sentence pairs as candidate seeds based on a similarity threshold, and applies multiple extension and filtering stages to generate final alignments. We use the open-source implementation with the recommended parameter settings.

## 6.1 Evaluating Active Hash Optimization

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

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

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

## 6.2 Evaluating Weighted Jaccard Similarity

This section evaluates near-duplicate text alignment under weighted Jaccard similarity. Our algorithm `MonoActive` is the only method that supports this problem. We evaluated four token weight functions $w(t, x)$: (1) binary TF weight $[x \geq 1]$; (2) logarithmic TF weight $\log(x + 1)$; (3) raw count TF weight $x$; and (4) squared TF weight $x^2$; all combined with unary IDF weight.

Figures 6(a) reports the partition sizes (i.e.,, the number of compact windows generated) as the text length $n$ varies from $10^3$ to $10^6$. As observed, the squared weight consistently yields the largest partitions, followed by the raw count weight, whereas the binary weight produces the smallest partitions. For all weight functions, the partition size increases approximately linearly with $n$. For instance, when $n$ increases from 1,000 to 1 million on PAN, the partition sizes under the four weight functions increase from 64K, 67K, 71K, and 89K to 64M, 68M, 82M, and 158M, respectively. This is because the partition sizes of the binary, logarithmic, raw count, and squared weights are respectively $O(n)$, $O(n + n \log \log f_T)$, $O(n + n \log f_T)$, and $O(n + n \log f_T)$, according to Lemma 13. Although squared and raw count weights have the same complexity, the constant factor of squared weight is larger than that of raw count weight.

Figures 6(b) presents the average partitioning time (i.e.,, the compact window generation time) under the four weight functions as the text length $n$ increases from $10^3$ to $10^6$. Among them, the binary weight consistently achieved the fastest partitioning time, followed by the logarithmic weight, while the squared weight resulted in the slowest runtime. All weight functions exhibit quasilinear growth

| MonoActive | $\theta = 0.05$ | $\theta = 0.1$ | $\theta = 0.15$ | $\theta = 0.2$ | $\theta = 0.3$ | $\theta = 0.4$ | $\theta = 0.5$ |
|---|---|---|---|---|---|---|---|
| Binary TF (set Jaccard) | 0.4335 / 0.9932 / 0.6035 | 0.4798 / 0.9625 / 0.6404 | 0.5206 / 0.8794 / **0.6540** | 0.5518 / 0.7016 / 0.6178 | 0.7300 / 0.4406 / 0.5495 | 0.7240 / 0.3326 / 0.4558 | 0.6697 / 0.1890 / 0.2948 |
| Binary TF+standard IDF | 0.4660 / 0.9248 / **0.6197** | 0.5579 / 0.7755 / 0.6489 | 0.6072 / 0.5748 / 0.5906 | 0.6293 / 0.4637 / 0.5340 | 0.6859 / 0.3560 / 0.4687 | 0.6359 / 0.2240 / 0.3313 | 0.7720 / 0.0486 / 0.0915 |
| Raw TF (multi-set Jaccard) | 0.4109 / 0.9988 / 0.5823 | 0.4248 / 0.9979 / 0.5960 | 0.4457 / 0.9961 / 0.6158 | 0.4822 / 0.9770 / **0.6457** | 0.6622 / 0.8339 / **0.7382** | 0.7848 / 0.6807 / **0.7291** | 0.7912 / 0.3898 / **0.5223** |
| Raw TF+standard IDF | 0.4793 / 0.8696 / 0.6180 | 0.5779 / 0.7842 / **0.6654** | 0.6655 / 0.5986 / 0.6303 | 0.6947 / 0.4890 / 0.5740 | 0.7161 / 0.3797 / 0.4963 | 0.6827 / 0.2473 / 0.3631 | 0.7787 / 0.0376 / 0.0718 |

**Table 4: Effectiveness (Precision/Recall/F1) of MonoActive ($k = 64$) under different similarity thresholds $\theta$ and TF-IDF weighting schemes. The effectiveness of SeedExtension is 0.3939 / 0.7070 / 0.5059.**

with respect to $n$. For example, when $n$ increased from $10^3$ to $10^6$ on PAN, the partitioning times under the four weights increased from 0.47, 0.50, 0.48, and 0.49 seconds to 60.96, 110.99, 143.43, and 229.02 seconds, respectively. This aligns with our complexity analysis, which shows that the runtime of MonoActive scales with $|X(T)| \log n$ for all weight functions.

Figures 6(d) shows the average partition sizes generated under the four weight functions, with $n = 5{,}000$ and $k = 64$ fixed while varying the maximum token frequency $f_T$. As $f_T$ increased, the partition size under the binary weight remained constant. The partition size under logarithmic increased modestly, whereas those under raw count and squared weights grew sublinearly with $f_T$ On PAN dataset, when $f_T$ increased from the range $[901, 1000]$ to $[2901, 3000]$, the partition sizes under the binary, logarithmic, raw count, and squared weights increased from 320K, 412K, 607K, and 1M to 320K, 625K, 1.5M, and 2.8M, respectively. These results are consistent with our theoretical analysis. The numbers of compact windows generated by binary, logarithmic, raw count, and squared weights scale with $n$, $n \log\log f_T$, $n \log f_T$, and $n \log f_T$, respectively.

Figures 6(e) reports the average partitioning time of the four weight functions under varying $f_T$. The partitioning time either remained nearly constant or decreased slightly across all four weight functions. This is because a significant part of MonoActive's runtime is spent on sorting the active hash values, the number of which is proportional to the number of distinct tokens in the text. As $f_T$ increases while $n$ remains fixed, the number of distinct tokens decreases, thereby reducing the sorting cost.

Due to space constraints, we report the results on the PAN dataset in the main paper, leaving the complete results on the full dataset, as well as a detailed discussion of query time to the technical report.

## 6.3 Evaluating Effectiveness

This section evaluates MonoActive's effectiveness (precision, recall, F1) under different TF-IDF schemes on PAN dataset compared to SeedExtension. For unary IDF, we omit IDF weighting because it is constant 1. In this study, the only non-unary IDF variant considered is the standard IDF in Table 2, computed over the entire PAN dataset. The PAN dataset consists of manually annotated plagiarism cases, where each case specifies a suspicious text segment and its corresponding plagiarized segment. To adapt this dataset to our near-duplicate alignment setting, for each source–suspicious pair, we treat the annotated suspicious segment as the query and the full source document as the source text.

By using the annotated plagiarism spans as the ground truth, we report the effectiveness of baseline SeedExtension and MonoActive under different TF-IDF strategies in Table 4. Notably, the effectiveness results of AllAlign are identical to those of Binary TF (set Jaccard) and Raw TF (Multi-set Jaccard) with unary IDF. This because MonoActive and AllAlign all rely on the min-hash scheme. If the random seeds are the same, they estimate the Jaccard similarity of subsequences identically and differ only in partitions.

As shown in Table 4, overall, the best F1 score is usually achieved by Raw TF. Specifically, smaller thresholds lead to consistently high recall but relatively low precision, while increasing $\theta$ improves precision at the cost of recall. This trend is expected, as stricter similarity thresholds reduce the number of candidate subsequences.

Among the weighting strategies, Binary TF and Raw TF exhibit extremely high recall across all thresholds, reaching nearly perfect recall at small $\theta$. However, this comes with limited precision, especially at larger thresholds. Incorporating IDF weighting substantially improves precision across all thresholds, leading to more balanced precision–recall trade-offs and higher F1 scores in most cases. For instance, at $\theta = 0.1$, Raw TF+IDF achieves an 11.5% relative improvement in F1 score compared to Raw TF.

Notably, the highest F1 score 0.7382 is achieved by Raw TF under multi-set Jaccard at $\theta = 0.3$, highlighting the benefits of multi-set Jaccard similarity when proper thresholds are used.

**Compare with SeedExtension**. SeedExtension has an F1 score of 0.5059, lower than that of most of MonoActive's configurations. This is expected as SeedExtension is primarily designed for full-document–to–full-document comparison while our task focuses on matching short fragment queries against long source documents. The lack of context of query makes SeedExtension difficult to establish reliable seed matches, leading to limited performace.

## 6.4 Comparing with State-of-the-Art

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

Figures 7(a)-7(c) show, when varying the text length $n$ from $10^3$ to $10^6$, the average number of compact windows generated by MonoActive and AllAlign (i.e., partition sizes, which are proportional to their index sizes) on three datasets OWT, PAN and NEWS. In the same figures, the reduction ratio, defined as $1 - \frac{\text{MonoActive}}{\text{AllAlign}}$, is overlaid as a purple line using a secondary $y$-axis.. As shown, MonoActive consistently generated less compact windows than AllAlign, up to 30.82%. More importantly, the gap between the number of compact windows for MonoActive and AllAlign steadily widens as $n$ increases. For example, when $n$ increased from 1000 to 1 million, the gap grew from 10.86% to 30.82% on PAN and from 7.69% to 26.08% on OWT. This is because AllAlign's recursive approach divides the rectangles into multiple rectangular regions, imposing early boundaries that may split what is a single compact window in MonoActive into multiple windows. The number of compact windows generated by MonoActive grew linearly with $n$, which is consistent with our complexity analysis.

Figures 7(d)-7(f) show when varying the text length $n$ from $10^3$ to $10^6$, the average compact window generation time (i.e., partition time, which is proportional to the index time) of AllAlign and MonoActive, respectively. In the same figures, the purple line shows the speedup of MonoActive over AllAlign. MonoActive was significantly faster than AllAlign. For example, on the PAN dataset, when $n = 10^6$, AllAlign took 7,581s on average to generate the compact windows, whereas MonoActive only spent 284s, achieving a speedup of 26.7×. This is because AllAlign produces more compact windows than MonoActive and AllAlign is recursive whereas MonoActive is non-recursive. Moreover, the speedup of MonoActive over AllAlign increased as $n$ grew, e.g., on the PAN

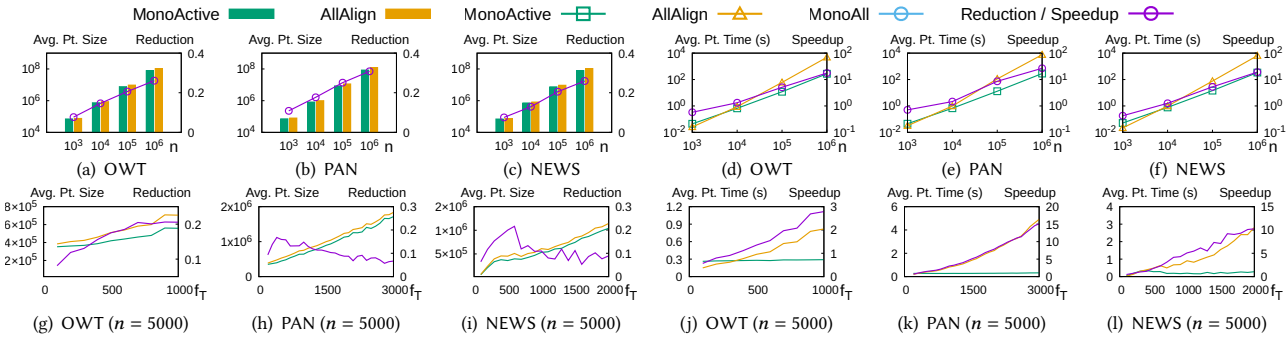



**Figure 7: Comparing with the state-of-the-art for multi-set Jaccard similarity. $k = 64$ in all experiments.**

dataset, when $n$ increased from $10^4$ to $10^6$, the speedup improved from 1.46 to 26.7. A similar trend is observed on OWT and NEWS.

We also evaluate the impact of the maximum token frequency $f_T$. Figures 7(g)–7(l) present the number of compact windows generated and the compact window generation time of the two methods, under varying $f_T$ with fixed text length $n = 5000$. By the same purple line, the relative improvements are marked in the same figures. As $f_T$ increases, the generation time of `MonoActive` remains largely stable, whereas that of `AllAlign` increases significantly. For example, on the PAN dataset, when $f_T$ increases from 600 to 3000, the speedup improved from 1.41 to 16.26. In addition, the gap in the number of compact windows produced by the two methods first grows and then decreases with larger $f_T$. This trend arises because higher token frequencies lead to deeper recursion and more early boundaries in `AllAlign`, which fragment compact windows and increase computational cost. However, as $f_T$ approaches $n$, the text instances approach the worst-case scenario and both methods produce a large number of compact windows. The gap of the query latency increases with the increase of $f_T$. This is because the collided windows are dominated by tokens with the highest frequencies. In `AllAlign`, the high-frequency tokens trigger deeper recursion, resulting in more finely fragmented compact windows. These fragments increase the number of collided compact windows and the number of iterations in the inner loop. Due to space constraints, we defer the discussion of query time to the technical report [59].

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

## 7 Related Work

**Near-Duplicate Text Alignment.** Most of existing methods for near-duplicate text alignment rely on rule-based heuristics and the "seeding-extension-filtering" framework [3, 5, 7, 23, 24, 28, 34, 47]. They first find "seed matches" between the data texts and the query. Various kinds of seeds have been proposed, such as fingerprints [39], super-shingles [8], 0 mod p [34], fixed-length windows [55], q-grams [46], and sentences [45]. Then they extend the seed matches as far as possible to form candidates. Finally, candidates failing predefined criteria (e.g., length or overlap thresholds) are filtered. However, these heuristics are highly sensitive to the hard-to-tune hyper-parameters [17] such as the granularity of the seeds and various kinds of thresholds [1, 45]. They also lack accuracy guarantees.

A few recent works propose to use the min-hash techniques [6] for near-duplicate text alignment [15, 37, 38, 56]. They introduce the concept of "compact windows" to group nearby subsequences sharing the same min-hash. It has been shown that when duplicate tokens have the same hash value, the $O(kn^2)$ min-hashes in a text with $n$ tokens can be compressed in compact windows using $O(kn)$ space and $O(kn \log n)$ time, where $k$ is the sketch size [15]. Along this line, an algorithm is developed to group the $O(n^2)$ bottom-$k$ sketches in a text with $n$ tokens into compact windows using $O(nk^2)$ space and $O(n \log n + nk)$ time [56]. To further reduce the space cost, another algorithm is designed to group the $O(kn^2)$ one-permutation hashing [30] min-hash sketches into compact windows using $O(n+k)$ space and $O(n \log n + k)$ time [38]. These algorithms are used to evaluate the memorization behaviour in large language models (LLMs), revealing that up to 10% of texts generated by GPT-2 [44] had near-duplicates in its training data. It also shows that the min-hash sketches of all subsequences no shorter than $t$ tokens in a text with $n$ tokens can be grouped into $O(\frac{n}{t})$ compact windows on average [37]. However, none of these works can deal with weighted min-hash (i.e., consistent weighted sampling [27]).

**Min-Hash Sketch.** Min-hash was originally introduced in statistics for coordinated sampling [4] and later adapted for database applications by Flajolet and Martin [16]. Broder [7] employed the min-hash sketch to detect near-duplicate web pages. Variants of the min-hash sketch have been proposed to improve the sketching time of the classic min-hash sketch, including the bottom-$k$ sketch [53], one-permutation hashing (OPH) [30], and fast similarity sketch [12]. The number of min-hashes generated by one-permutation hashing is not fixed. A few works aim to address this issue [49–51].

**Weighted Jaccard Similarity Estimation.** Many techniques have been proposed to estimate the weighted Jaccard similarity [14, 27, 33, 58]. Specifically, [21] extends the classic min-hash to estimate the multi-set Jaccard similarity. A method dealing with integer weights is mentioned in [10]. It was extended by [11] to support a more general weight function. Consistent weighted sampling (CWS) is first proposed in [33] to estimate weighted Jaccard similarity. Ioffe proposes improved consistent weighted sampling, which simplified CWS and guarantees worst-case constant time for each non-zero weight [27]. Shrivastava proposes to use rejected sampling to estimate the weigthed Jaccard Similarity [48].

## 8 Conclusion

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

# References


[1] E. Agirre, C. Banea, D. M. Cer, M. T. Diab, A. Gonzalez-Agirre, R. Mihalcea, G. Rigau, and J. Wiebe. Semeval-2016 task 1: Semantic textual similarity, monolingual and cross-lingual evaluation. In *SEMEVAL*, pages 497–511. The Association for Computer Linguistics, 2016.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[3] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

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

[5] S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In *SIGMOD*, pages 398–409. ACM Press, 1995.

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

[7] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Comput. Networks*, 29(8-13):1157–1166, 1997.

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

[9] N. Carlini, D. Ippolito, M. Jagielski, K. Lee, F. Tramèr, and C. Zhang. Quantifying memorization across neural language models. *CoRR*, abs/2202.07646, 2022.

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

[11] O. Chum, J. Philbin, A. Zisserman, et al. Near duplicate image detection: Min-hash and tf-idf weighting. In *Bmvc*, volume 810, pages 812–815, 2008.

[12] S. Dahlgaard, M. B. T. Knudsen, and M. Thorup. Fast similarity sketching. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 663–671. IEEE, 2017.

[13] H. Ding, J. Zhai, D. Deng, and S. Ma. The case for learned provenance graph storage systems. In J. A. Calandrino and C. Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 3277–3294. USENIX Association, 2023.

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

[15] W. Feng and D. Deng. Allign: Aligning all-pair near-duplicate passages in long texts. In *SIGMOD*, pages 541–553. ACM, 2021.

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

[17] T. Foltýnek, N. Meuschke, and B. Gipp. Academic plagiarism detection: A systematic literature review. *ACM Comput. Surv.*, 52(6):112:1–112:42, 2020.

[18] A. Franz and T. Brants. All our n-gram are belong to you. *Google Machine Translation Team*, 20, 2006.

[19] P. Gage. A new algorithm for data compression. *C Users J.*, 12(2):23–38, feb 1994.

[20] A. Gokaslan and V. Cohen. Openwebtext corpus.

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

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

[23] O. A. Hamid, B. Behzadi, S. Christoph, and M. R. Henzinger. Detecting the origin of text segments efficiently. In *WWW*, pages 61–70. ACM, 2009.

[24] T. C. Hoad and J. Zobel. Methods for identifying versioned and plagiarized documents. *J. Assoc. Inf. Sci. Technol.*, 54(3):203–215, 2003.

[25] W.-K. Hon, T.-H. Lin, K. Sadakane, and W.-K. Sung. A space-efficient framework for top-k string retrieval. *SIAM Journal on Computing*, 42(6):2383–2420, 2013.

[26] S. Ioffe. Improved consistent sampling, weighted minhash and L1 sketching. In *ICDM*, pages 246–255. IEEE Computer Society, 2010.

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

[28] J. W. Kim, K. S. Candan, and J. Tatemura. Efficient overlap and content reuse detection in blogs and online news articles. In *WWW*, pages 81–90. ACM, 2009.

[29] K. Lee, D. Ippolito, A. Nystrom, C. Zhang, D. Eck, C. Callison-Burch, and N. Carlini. Deduplicating training data makes language models better. In *ACL*, pages 8424–8445, 2022.

[30] P. Li, A. B. Owen, and C. Zhang. One permutation hashing. In *NIPS*, pages 3122–3130, 2012.

[31] D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *Journal of the American Society for Information Science and Technology*, 58(7):1019–1031, 2007.

[32] I. Magar and R. Schwartz. Data contamination: From memorization to exploitation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 157–165. Association for Computational Linguistics, 2022.

[33] M. Manasse, F. McSherry, and K. Talwar. Consistent weighted sampling. *Unpublished technical report) http://research. microsoft. com/en-us/people/manasse*, 2, 2010.

[34] U. Manber. Finding similar files in a large file system. In *USENIX Winter 1994 Technical Conference, San Francisco, California, USA, January 17-21, 1994, Conference Proceedings*, pages 1–10. USENIX Association, 1994.

[35] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[36] G. Pasi and G. Bordogna. Introduction to the minitrack on intelligent information access and retrieval. In T. X. Bui, editor, *56th Hawaii International Conference on System Sciences, HICSS 2023, Maui, Hawaii, USA, January 3-6, 2023*, pages 4169–4170. ScholarSpace, 2023.

[37] Z. Peng, Z. Wang, and D. Deng. Near-duplicate sequence search at scale for large language model memorization evaluation. *Proc. ACM Manag. Data*, 1(2):179:1–179:18, 2023.

[38] Z. Peng, Y. Zhang, and D. Deng. Near-duplicate text alignment with one permutation hashing. *Proc. ACM Manag. Data*, 2(4):200:1–200:26, 2024.

[39] M. Potthast, A. Barrón-Cedeño, A. Eiselt, B. Stein, and P. Rosso. Overview of the 2nd international competition on plagiarism detection. In *CLEF 2010 LABs and Workshops, Notebook Papers*, volume 1176 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2010.

[40] M. Potthast, A. Eiselt, A. Barrón-Cedeño, B. Stein, and P. Rosso. Overview of the 3rd international competition on plagiarism detection. In *CLEF 2011 Labs and Workshop, Notebook Papers*, volume 1177 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

[41] M. Potthast, T. Gollub, M. Hagen, J. Kiesel, M. Michel, A. Oberländer, M. Tippmann, A. Barrón-Cedeño, P. Gupta, P. Rosso, and B. Stein. Overview of the 4th international competition on plagiarism detection. In *CLEF 2012 Evaluation Labs and Workshop*, volume 1178 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.

[42] M. Potthast, M. Hagen, A. Beyer, M. Busse, M. Tippmann, P. Rosso, and B. Stein. Overview of the 6th international competition on plagiarism detection. In *Working Notes for CLEF 2014 Conference*, volume 1180 of *CEUR Workshop Proceedings*, pages 845–876. CEUR-WS.org, 2014.

[43] M. Potthast, M. Hagen, T. Gollub, M. Tippmann, J. Kiesel, P. Rosso, E. Stamatatos, and B. Stein. Overview of the 5th international competition on plagiarism detection. In *Working Notes for CLEF 2013 Conference*, volume 1179 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[44] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[45] M. A. Sanchez-Perez, A. F. Gelbukh, and G. Sidorov. Adaptive algorithm for plagiarism detection: The best-performing approach at PAN 2014 text alignment competition. In *6th International Conference of the CLEF Association*, volume 9283 of *Lecture Notes in Computer Science*, pages 402–413. Springer, 2015.

[46] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *SIGMOD*, pages 76–85. ACM, 2003.

[47] J. Seo and W. B. Croft. Local text reuse detection. In *SIGIR*, pages 571–578. ACM, 2008.

[48] A. Shrivastava. Simple and efficient weighted minwise hashing. *Advances in Neural Information Processing Systems*, 29, 2016.

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

[50] A. Shrivastava and P. Li. Densifying one permutation hashing via rotation for fast near neighbor search. In *International Conference on Machine Learning*, pages 557–565. PMLR, 2014.

[51] A. Shrivastava and P. Li. Improved densification of one permutation hashing. *arXiv preprint arXiv:1406.4784*, 2014.

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

[53] M. Thorup. Bottom-k and priority sampling, set similarity and subset sums with minimal independence. In *STOC*, pages 371–380. ACM, 2013.

[54] T. Vu, X. He, G. Haffari, and E. Shareghi. Koala: An index for quantifying overlaps with pre-training corpora. In Y. Feng and E. Lefever, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023 - System Demonstrations, Singapore, December 6-10, 2023*, pages 90–98. Association for Computational Linguistics, 2023.

[55] P. Wang, C. Xiao, J. Qin, W. Wang, X. Zhang, and Y. Ishikawa. Local similarity search for unstructured text. In *SIGMOD*, pages 1991–2005. ACM, 2016.

[56] Z. Wang, C. Zuo, and D. Deng. Txtalign: Efficient near-duplicate text alignment search via bottom-k sketches for plagiarism detection. In *SIGMOD*, pages 1146–1159. ACM, 2022.

[57] J. J. Webster and C. Kit. Tokenization as the initial phase in nlp. In *COLING 1992 volume 4: The 14th international conference on computational linguistics*, 1992.

[58] W. Wu, B. Li, L. Chen, C. Zhang, and S. Y. Philip. Improved consistent weighted sampling revisited. *IEEE Transactions on Knowledge and Data Engineering*, 31(12):2332–2345, 2018.

[59] Y. Zhang, M. Qiao, Z. Peng, and D. Deng. Near-duplicate text alignment under weighted jaccard similarity. https://github.com/rutgers-db/WeightAlign/raw/main/Near_Duplicate_Text_Alignment_under_Weighted_Jaccard_Similarity.pdf, 2025. Technical Report.

## A Correctness of Monotonic Partitioning

This section serves as a proof of Theorem 1. Let $n = |\mathsf{T}|$.

**Augmented Skyline.** Since every key $(p, q) \in \mathsf{K}(\mathsf{T})$ has $(p, q) \in [1, n] \times [1, n]$, the two guard keys $(0, 0)$ and $(n + 1, n + 1)$ neither dominate nor are dominated by any key in $\mathsf{K}(\mathsf{T})$. For any key set $\mathsf{L} \subseteq \mathsf{K}(\mathsf{T})$, set $\{(0, 0), (n + 1, n + 1)\} \cup \mathsf{S}(\mathsf{L})$ is a skyline – no key in the set dominates another. We refer to this set as the augmented skyline of $\mathsf{L}$. All the previous discussions about a skyline naturally apply to the corresponding augmented skyline.

By Definition 8, $\mathrm{rec}(\{(0, 0), (n + 1, n + 1)\} \cup \mathsf{S}(\mathsf{L})) = \mathrm{rec}(\mathsf{S}(\mathsf{L})) \cup \mathrm{rec}(0, 0) \cup \mathrm{rec}(n + 1, n + 1) = \mathrm{rec}(\mathsf{S}(\mathsf{L}))$. This is because $\mathrm{rec}(0, 0) = [1, 0] \times [0, n] = \phi$ and $\mathrm{rec}(n + 1, n + 1) = [1, n + 1] \times [n + 1, n] = \phi$.

Lemma 7 tests the existence of any key in $\mathsf{L}$ that dominates $(b, c)$. If not, we need to update the augmented skyline to include $(b, c)$ and, at the same time, generate compact windows according to Lemma 14 below. Please see Figure 3(b) as an illustration.

Lemma 14. *Given a key set $\mathsf{L} \subseteq \mathsf{K}(\mathsf{T})$ and a key $(b, c) \notin \mathsf{L}$. Let $\mathsf{S}$ be the augmented skyline of $\mathsf{L}$ in coordinate order. Let*

- *$i$ be the largest index such that $\mathsf{S}[i].y < c$ and*
- *$j$ be the smallest index such that $\mathsf{S}[j].x > b$.*

*The two guard keys in $\mathsf{S}$ ensure that $i$ and $j$ exist. We claim* C1 *&*C2*.*

C1 *If $i + 1 \le j - 1$, then the keys in $\mathsf{S}$ dominated by $(b, c)$ are exclusively $\mathsf{S}[i + 1], \mathsf{S}[i + 2], \cdots, \mathsf{S}[j - 1]$; otherwise, there is no key in $\mathsf{S}$ dominated by $(b, c)$.*

C2 *If $(b, c)$ is not dominated by any key in $\mathsf{S}$, then*

$$r_k \doteq \begin{cases} [\mathsf{S}[k].x + 1,\ b] \times [c,\ \mathsf{S}[k + 1].y - 1], & k = i, \\ [\mathsf{S}[k].x + 1,\ b] \times [\mathsf{S}[k].y,\ \mathsf{S}[k + 1].y - 1], & k \in [i + 1, j - 1]. \end{cases}$$

*are $j - i$ mutually disjoint rectangles that jointly cover the growing part of the skyline when including $(b, c)$, i.e.,*

$$r_i \cup r_{i+1} \cup \cdots \cup r_{j-1} = \mathrm{rec}(b, c) \setminus \mathrm{rec}(\mathsf{S}).$$

Proof. We prove C1 in two complementary cases.

**Case $i + 1 > j - 1$.** Assume there is a key $\mathsf{S}[k]$ that is dominated by $(b, c)$. Then $\mathsf{S}[k].x \le b < \mathsf{S}[j].x$ and $\mathsf{S}[i].y < c \le \mathsf{S}[k].y$. By Lemma 6, $i < k$ and $k < j$. Hence $i \le k-1 \le j-2$, which contradicts $i + 1 > j - 1$. Thus, no key in S is dominated by $(b, c)$.

**Case $i + 1 \le j - 1$.** For any $k \notin [i + 1, j - 1]$, either $k \le i$ or $k \ge j$. By the definition of $i$, $\mathsf{S}[k].y < c$ for $k \le i$, and by the definition of $j$, $\mathsf{S}[k].x > b$ for $k \ge j$. For any $k \le i$ or $k \ge j$, $(b, c)$ does not dominate $\mathsf{S}[k]$. For any $k \in [i + 1, j - 1]$, by the definitions of $i$ and $j$, we have $\mathsf{S}[k].x \le b$ and $\mathsf{S}[k].y \ge c$. Since $(b, c) \notin \mathsf{L}$, we must have $(b, c) \ne \mathsf{S}[k]$. Thus $[b, c] \subset [\mathsf{S}[k].x, \mathsf{S}[k].y]$. Thus, $(b, c)$ dominates $\mathsf{S}[k]$ for $\forall k \in [i + 1, j - 1]$. This proves C1.

We prove C2 below by considering when $(b, c)$ is not dominated by any key in S. We first show $\mathsf{S}[i + 1]$ exists in the augmented skyline by showing $i < j$. If otherwise, i.e.,, $i \ge j$, by Lemma 6, $\mathsf{S}[i].x \ge \mathsf{S}[j].x > b$, in addition, $\mathsf{S}[i].y < c$ by definition, thus $\mathsf{S}[i]$ dominates $(b, c)$, contradiction. $\mathsf{S}[j]$ exists and thus $\mathsf{S}[i + 1]$ exists.

We then show $r_i, r_{i+1}, \cdots, r_{j-1}$ are mutually disjoint: the $y$-ranges of these rectangles, i.e., $[c, \mathsf{S}[i+1].y-1]$, $[\mathsf{S}[i+1].y, \mathsf{S}[i+2].y]$, $\cdots$, $[\mathsf{S}[j - 2].y, \mathsf{S}[j - 1].y]$, $[\mathsf{S}[j - 1].y, \mathsf{S}[j].y]$ are mutually disjoint.

Next, consider any $(p, q) \in \mathrm{rec}(b, c) \setminus \mathrm{rec}(\mathsf{S})$, we show there exists $k \in [i, j - 1]$ such that $(p, q) \in r_k$. Since $(p, q) \in \mathrm{rec}(b, c)$ and thus $q \in [c, n]$, $q$ will fall in exactly one of the 3 cases below.
Case 1 : $q \in [c, \mathsf{S}[i + 1].y - 1]$,
Case 2 : $q \in [\mathsf{S}[k].y, \mathsf{S}[k + 1].y - 1]$ for some $k \in [i + 1, j - 1]$,
Case 3 : $q \in [\mathsf{S}[j].y, n]$.

In Case 1, the definition of $i$ ensures that $\mathsf{S}[i].y < c$. Besides, for any $\mathsf{S}[u]$ in the augmented skyline, $\mathsf{S}[u].y - 1 \le n$, and thus $[c, \mathsf{S}[i + 1].y - 1] \subseteq [\mathsf{S}[i].y, n]$. $(p, q)$ is not dominated by any key in S including $\mathsf{S}[i]$ and $q \in [c, \mathsf{S}[i + 1].y - 1]$, $p \notin [1, \mathsf{S}[i].x]$. As $(p, q) \in \mathrm{rec}(b, c)$, $p \in [1, b]$, and thus $p \in [\mathsf{S}[i].x + 1, b]$. Thus, $(p, q) \in r_i$.

In Case 2, consider $k \in [i + 1, j - 1]$ such that $q \in [\mathsf{S}[k].y, \mathsf{S}[k + 1].y - 1]$. As $(p, q) \notin \mathrm{rec}(\mathsf{S}[k]) = [1, \mathsf{S}[k].x] \times [\mathsf{S}[k].y, n]$ and $p \in [\mathsf{S}[k].y, \mathsf{S}[k + 1].y - 1] \subseteq [\mathsf{S}[k].y, n]$, $p \notin [1, \mathsf{S}[k].x]$. Together with $p \in [1, b]$, we have $p \in [\mathsf{S}[k].x + 1, b]$. Thus $(p, q) \in r_k$.

Lastly, we show Case 3 is impossible. If $\mathsf{S}[j] = (n + 1, n + 1)$, then $[\mathsf{S}[j].y, n] = \phi$; otherwise, since $(p, q)$ is not dominated by $\mathsf{S}[j]$ and $q \in [\mathsf{S}[j].y, n]$, we have $p \notin [1, \mathsf{S}[j].x]$. As $p \in [1, b]$, $\mathsf{S}[j].x > b \ge p > \mathsf{S}[j].x$, contradiction. Thus, Case 3 is impossible.

In conclusion, $\mathsf{S}[i+1]$ exists and thus the ranges of the three cases are well defined. Case 3 is impossible, thus $q$ can only fall in Case 1 or 2, and in either case, there is $k \in [i, j - 1]$ s.t. $(p, q) \in r_k$. □

Now we prove Theorem 1. Firstly, it is trivial to verify GenerateKeys produces $\mathsf{K}(\mathsf{T})$. Algorithm 4 examines all keys in $\mathsf{K}(\mathsf{T})$ in the increasing order of their hash values. Let $(b, c)$ with hash value $v$ be the current visiting key and $\mathsf{L}$ be the set of visited keys. We prove by induction that at the end of each for-loop of Algorithm 4, S is the augmented skyline of $\mathsf{L} \cup \{(b, c)\}$ and the set of compact windows $\mathcal{P}(b, c)$ generated during the loop constitutes a partition of $\mathrm{rec}(b, c) \setminus \mathrm{rec}(\mathsf{S})$. Thus the union of all generated compact windows forms a partition of $\mathrm{rec}(\mathsf{S})$. Note that a guard key $(p, q)$ has $\mathrm{rec}(p, q) = \phi$, thus $\mathrm{rec}(\mathsf{S}) = \mathrm{rec}(\mathsf{S}(\mathsf{L}))$.

We assume S is the augmented skyline of $\mathsf{L}$ at the beginning of each for loop. This is true at the beginning of the first loop, when $\mathsf{L} = \phi$ and $\mathsf{S} = \{(0, 0), (n + 1, n + 1)\}$. Clearly, S is the augmented skyline of $\mathsf{L}$. Next, we consider two cases for each loop.

In the case there exists a key in S dominates $(b, c)$, by Lemma 7, Line 5 evaluates to true and the loop ends[3]. S remains unchanged. The augmented skyline of $\mathsf{L} \cup \{(b, c)\}$ is S as $(b, c)$ is dominated. $\mathcal{P}(b, c) = \phi$ is a partition of $\mathrm{rec}(b, c) \setminus \mathrm{rec}(\mathsf{S}) = \phi$.

In the case $(b, c)$ is not dominated by any key in $\mathsf{S}(\mathsf{L})$, by Lemma 7, Line 5 evaluates to false. Furthermore, by Lemma 14(C1), all keys dominated by $(b, c)$ in S (if there is any) are precisely $\mathsf{S}[i+1], \cdots, \mathsf{S}[j-1]$. By the definition of skyline, $\mathsf{S} \cup \{(b, c)\} \setminus \{\mathsf{S}[i + 1], \cdots, \mathsf{S}[j - 1]\}$ must be the augmented skyline of $\mathsf{L} \cup \{(b, c)\}$. Next, we show that $\mathcal{P}(b, c)$ generated during the loop is a partition of $\mathrm{rec}(b, c) \setminus \mathrm{rec}(\mathsf{L})$.

Consider Lemma 14 (C2) and determine $i$,$j$ and $r_k$ (denoted as $[a_k, b_k] \times [c_k, d_k]$), $k \in [i, j - 1]$ accordingly. Note that $r_k \ne \phi$ only if $a_k \le b_k$ and $c_k \le d_k$. The tuples added to $\mathcal{P}(\mathsf{T}, h)$ in Lines 9-13 are

$$\mathcal{P}(b, c) = \{W_k \doteq \langle T, h, v, a_k, b_k, c_k, d_k \rangle | k \in [i, j - 1], r_k \ne \emptyset\}.$$

First, any $W_k \in \mathcal{P}(b, c)$ is a compact window because by Lemma 3 and Lemma 4, all subsequences in $\mathrm{rec}(b, c) \setminus \mathrm{rec}(\mathsf{S}(\mathsf{L}))$ have min-hash $v$. Secondly, all the compact windows $W_k$ in $\mathcal{P}(b, c)$ constitute a partition of $\mathrm{rec}(b, c) \setminus \mathrm{rec}(\mathsf{S})$ because by Lemma 14 (C2), the rectangles $\{r_k | k \in [i, j - 1]\}$, form a partition of $\mathrm{rec}(b, c) \setminus \mathrm{rec}(\mathsf{S})$.

[3]When using Lemma 7 on $\mathsf{S}(\mathsf{L})$, $j$ does not exist only if Line 5 on augmented skyline S has $\mathsf{S}[j']$ taking the guard key $(0, 0)$ – Line 5 also tests false; otherwise, the same key is returned, i.e., $\mathsf{S}(\mathsf{L})[j] = \mathsf{S}[j']$. Thus, Line 5 correctly facilitates Lemma 7.

Since along the loop, the increments $\text{rec}(b,c) \setminus \text{rec}(\mathsf{S})$ are disjoint, the union of all generated $\mathcal{P}(b,c)$ forms a partition of $\text{rec}(\mathsf{S}(\mathsf{L}))$.

The resulting $\mathcal{P}(\mathrm{T}, h)$ satisfies the coverage condition in Definition 3: for every subsequence $\mathrm{T}[x,y]$ of T, $(x,y)$ is in $\text{rec}(\mathsf{S}(\mathsf{L}))$ and is thus covered by exactly one compact window. Denote by $v$ the hash value of subsequence $\mathrm{T}[x,y]$. By Lemma 3, $\mathrm{T}[x,y]$ must contain a key, say $(b,c)$, with hash value $v$. Thus, $(x,y) \in \text{rec}(b,c) \subseteq \text{rec}(\mathsf{S}(\mathsf{L}))$. This completes the proof of Theorem 1.

Lemma 15. *Algorithm 4 with* GenerateActiveKeys *produces a partition of* T.

Proof. It is easy to verify GenerateActiveKeys produces all the active keys $\mathsf{X}(\mathrm{T})$. Based on Lemma 8, for any non-active key $(b,c)$, it is skipped in our algorithm, which means $\mathcal{P}(b,c) = \phi$. Based on the proof of Theorem 1, Algorithm 4 with GenerateActiveKeys produces a partitioning of T. □

## B Lower Bound Analysis

This section analyzes the lower bound of the partition generation problem. A hard case is when all the tokens are the same, i.e., there is only one token $t$ in the text T. Next we show every partition $\mathcal{P}(\mathrm{T}, h)$ has $\Omega(n \log n)$ compact windows in expectation in the worst case. Formally, we define two subsequences are *mergeable* as below.

Definition 15 (Mergeable). *Given text* T *and hash function h, two subsequences* $\mathrm{T}[i,j]$ *and* $\mathrm{T}[p,q]$ *are mergeable if they can be represented by the same compact window* $\langle \mathrm{T}, h, v, a, b, c, d \rangle$*, i.e.,* $i, p \in [a,b]$*,* $b \le c$*,* $j, q \in [c,d]$*, and* $h(\mathrm{T}[x,y]) = v, \forall x \in [a,b], y \in [c,d]$.

Clearly, for a set of subsequences from the same text that are mutually not mergeable, it needs at least one compact window in the partition for each subsequence in the set. Thus we analyze the lower bound of the partition generation problem by constructing a set of subsequences that are mutually not mergeable.

Lemma 16. *For a text* T *of length n of one token t, i.e., all tokens in* T *are duplicate, any partition* $\mathcal{P}(\mathrm{T}, h)$ *of* T *contains* $\Omega(n + n \log n)$ *compact windows in expectation. The expectation is introduced by the randomness of the hash function as opposed to any other assumption.*

Proof. Let $h(t,x_1), h(t,x_2), \cdots, h(t,x_m)$ be all the active hash of the token $t$ where $1 \le x_1 < x_2 < \cdots < x_m \le n = |\mathrm{T}|$. That means, $h(t,x_j) > h(t,x_i)$ for every $1 \le j < i \le m$. Construct a set of subsequences $S = S_1 \cup S_2 \cup \cdots \cup S_m$ where $S_i = \{\mathrm{T}[p,q] \mid q-p+1 = x_i\}$. That is to say, $S_i$ is the set of all subsequences in T containing exactly $x_i$ tokens. Consider two sets $S_i$ and $S_j$ where $1 \le i \ne j \le m$. First, they must be disjoint as the lengths of their subsequences are different. Furthermore, based on the definition of active hash, the min-hash of subsequences in $S_i$ and $S_j$ are respectively $h(t,x_i)$ and $h(t,x_j)$. Since $x_i \ne x_j$, the subsequences from $S_i$ and $S_j$ are not mergeable based on Definition 15.

Assume that there are two distinct subsequences $\mathrm{T}[p_1,q_1]$ and $\mathrm{T}[p_2,q_2]$ from the same set $S_i$ that are mergeable. As they all of the same length and are distinct, either $p_1 < p_2$ or $p_1 > p_2$. Without loss of generality, assume $p_1 < p_2$. As the two subsequences are mergeable, there exists a compact window $\langle \mathrm{T}, h, v, a, b, c, d \rangle$ representing both subsequences. Thus $p_1 < p_2 \le b \le c \le q_1$. The subsequence $\mathrm{T}[b,c]$ then contains less than $x_i$ tokens because $c-b+1 \le q_1-p_2+1 = (q_1-p_1+1)+p_1-p_2 = x_i+p_1-p_2 < x_i$. Thus $h(\mathrm{T}[b,c]) > x_i$ since $h(t,x_i)$ is active hash, contradicting the definition of a compact window, i.e., $h(\mathrm{T}[b,c]) = h(\mathrm{T}[p_1,q_1]) = h(t,x_i)$.

Based on the discussion above, the subsequences in $S$ are mutually not mergeable. Next, we calculate the expected size of $S$. There are $n - x_i + 1$ subsequences in $S_i$. Thus $|S| = \sum_{i=1}^{m}(n - x_i + 1)$. Let us consider an arbitrary index $j \in [1,n]$. We observe that $j = x_i$ for some $i \in [1,m]$ if and only if $h(t,j)$ is the smallest among $h(t,1), h(t,2), \cdots, h(t,j)$. The probability is $\frac{1}{j}$. Thus we have

$$\begin{aligned}\mathbb{E}[|S|] &= \mathbb{E}[\sum_{i=1}^{m}(n - x_i + 1)] = \sum_{i=1}^{n}(n - i + 1)\frac{1}{i} \\ &= (n+1)\sum_{i=1}^{n}\frac{1}{i} - n \ge (n+1)\ln n - n = \Omega(n \log n).\end{aligned} \tag{6}$$

Note that when $n = 1$, $\mathbb{E}[|S|] = 1$, besides, $n \le n \log n$ when $n \ge 2$. So $\mathbb{E}[|S|] = \Omega(n \log n + n)$. The subsequences in $S$ are mutually not mergeable. One compact window is needed to represent each of the subsequence in $S$. Thus any partition $\mathcal{P}(\mathrm{T}, h)$ contains $\Omega(n \log n + n)$ compact windows in expectation. □

Lemma 16 shows a hard case in terms of partition size. The reason we keep factor $n$ in the lower bound is for the special case when $n = 1$. This will be useful when we conduct frequency-aware analysis below.

**When $f_{\mathrm{T}} \ll n$.** Next we consider the cases when we know the length and the maximum token frequency $f_{\mathrm{T}}$ of a text.

Lemma 17. *Consider integer* $n > 0$ *and* $f_{\mathrm{T}} \in [1,n]$*, there is a text* T *of length n and maximum frequency* $f_{\mathrm{T}}$ *such that any partition* $\mathcal{P}(\mathrm{T}, h)$ *of* T *contains* $\Omega(n + n \log f_{\mathrm{T}})$ *compact windows in expectation.*

Proof. Without loss of generality, assume that $n$ is a multiple of $f_{\mathrm{T}}$. Let $m$ be $\frac{n}{f_{\mathrm{T}}}$. Let $\mathrm{T} = t_1 t_1 \cdots t_1 t_2 t_2 \cdots t_2 \cdots t_2 t_m \cdots t_m$ where each $t_i$ (where $1 \le i \le m$) appears $f_{\mathrm{T}}$ times next to each other in T.

We construct a set of subsequences $S = S^1 \cup S^2 \cup \cdots \cup S^m$ where $S^i$ is constructed in the same way as in the proof in Lemma 16 for $t_i$. The subsequences within each $S^i$ are mutually not mergeable as shown earlier. Next we show any two subsequences $\mathrm{T}[p_1,q_1]$ and $\mathrm{T}[p_2,q_2]$ from $S^i$ and $S^j$ (where $i \ne j$) are not mergeable. This is obvious as $h(\mathrm{T}[p_1,q_1]) = h(t_i,x)$ for some $x \in [1,f_i]$ and $h(\mathrm{T}[p_2,q_2]) = h(t_j,y)$ for some $y \in [1,f_y]$. However, $h(t_i,x) \ne h(t_j,y)$ assuming the universal hash function $h$ has no collision.

Based on the discussion above, it needs $\Omega(f_{\mathrm{T}}(1+\log f_{\mathrm{T}}))$ compact windows to represent all the subsequences $\mathrm{T}[p,q]$ where $\mathrm{T}[p] = [q] = t_i$ for each $1 \le i \le m$. Thus in total, it needs $\Omega(\sum_{i\in[m]} f_{\mathrm{T}}(1+\log f_{\mathrm{T}})) = \Omega(n + n \log f_{\mathrm{T}})$ compact windows to represent all the subsequences in T. Thus every partition $\mathcal{P}(\mathrm{T}, h)$ has $\Theta(n + n \log f_{\mathrm{T}})$ compact windows. □

Now we prove Theorem 3. Theorem 1 shows that the expected number of compact windows of Algorithm 4 is $O(n + n \log f_{\mathrm{T}})$ while Lemma 17 shows a case of T, given $n$ and $f_{\mathrm{T}}$, such that no partition can have the number of compact windows smaller than $\Omega(n + n \log f_{\mathrm{T}})$ in expectation. Thus, given $n$ and $f_{\mathrm{T}}$, Algorithm 4 is optimal in the worst case.

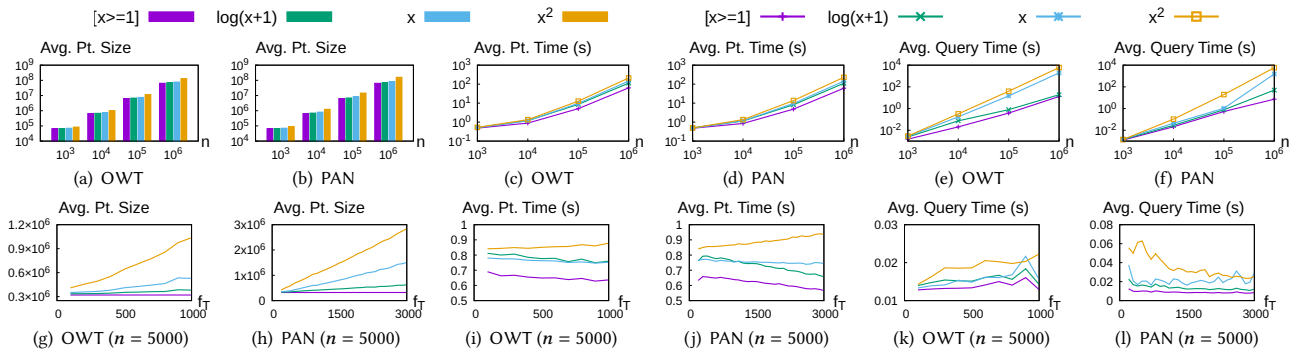



**Figure 8: Evaluating weighted Jaccard similarity under various weight functions. $k = 64$ in all experiments.**

## C Full Experiment of Evaluating Weighted Jaccard Similarity

This section evaluates near-duplicate text alignment under weighted Jaccard similarity. Our algorithm `MonoActive` is the only method that supports this problem. We evaluated four token weight functions $w(t, x)$: (1) binary TF weight $[x \geq 1]$; (2) logarithmic TF weight $\log(x + 1)$; (3) raw count TF weight $x$; and (4) squared TF weight $x^2$; all combined with unary IDF weight.

Figures 8(a)-8(b) report the partition sizes (i.e.,, the number of compact windows generated) as the text length $n$ varies from $10^3$ to $10^6$. As observed, the squared weight consistently yields the largest partitions, followed by the raw count weight, whereas the binary weight produces the smallest partitions. For all weight functions, the partition size increases approximately linearly with $n$. For instance, when $n$ increases from 1,000 to 1 million on PAN, the partition sizes under the four weight functions increase from 64K, 67K, 71K, and 89K to 64M, 68M, 82M, and 158M, respectively. This is because the partition sizes of the binary, logarithmic, raw count, and squared weights are respectively $O(n)$, $O(n + n \log \log f_{\mathrm{T}})$, $O(n + n \log f_{\mathrm{T}})$, and $O(n + n \log f_{\mathrm{T}})$, according to Lemma 13. Although squared and raw count weights have the same complexity, the constant factor of squared weight is larger than that of raw count weight.

Figures 8(c)-8(d) present the average partitioning time (i.e.,, the compact window generation time) under the four weight functions as the text length $n$ increases from $10^3$ to $10^6$. Among them, the binary weight consistently achieved the fastest partitioning time, followed by the logarithmic weight, while the squared weight resulted in the slowest runtime. All weight functions exhibit quasi-linear growth with respect to $n$. For example, when $n$ increased from $10^3$ to $10^6$ on PAN, the partitioning times under the four weights increased from 0.47, 0.50, 0.48, and 0.49 seconds to 60.96, 110.99, 143.43, and 229.02 seconds, respectively. This aligns with our complexity analysis, which shows that the runtime of `MonoActive` scales with $|\mathrm{X(T)}| \log n$ for all weight functions.

Figures 8(e)-8(f) show the average query latency when varying the text length $n$ from $10^3$ to $10^6$. The queries are sampled from the same dataset of the data texts but are disjoint from them. Each query is a text of length 100, and the same set of queries is used across all values of $n$. Similarly, the squared weight had the largest query latency, followed by raw count weight, while the binary weight had the smallest latency. This is because query processing cost grows with the number of compact windows sharing the same hash values between the query and the data texts (a.k.a., collided compact windows). Weight functions that generate more compact windows are more likely to incur a larger number of such collisions, thereby increasing query latency.

Figures 8(g)-8(h) show the average partition sizes generated under the four weight functions, with $n = 5{,}000$ and $k = 64$ fixed while varying the maximum token frequency $f_{\mathrm{T}}$. As $f_{\mathrm{T}}$ increased, the partition size under the binary weight remained constant. The partition size under logarithmic increased modestly, whereas those under raw count and squared weights grew sublinearly with $f_{\mathrm{T}}$ On PAN dataset, when $f_{\mathrm{T}}$ increased from the range $[901, 1000]$ to $[2901, 3000]$, the partition sizes under the binary, logarithmic, raw count, and squared weights increased from 320K, 412K, 607K, and 1M to 320K, 625K, 1.5M, and 2.8M, respectively. These results are consistent with our theoretical analysis. The numbers of compact windows generated by binary, logarithmic, raw count, and squared weights scale with $n$, $n \log \log f_{\mathrm{T}}$, $n \log f_{\mathrm{T}}$, and $n \log f_{\mathrm{T}}$, respectively.

Figures 8(i)-8(j) report the average partitioning time of the four weight functions under varying $f_{\mathrm{T}}$. The partitioning time either remained nearly constant or decreased slightly across all four weight functions. This is because a significant part of `MonoActive`'s runtime is spent on sorting the active hash values, the number of which is proportional to the number of distinct tokens in the text. As $f_{\mathrm{T}}$ increases while $n$ remains fixed, the number of distinct tokens decreases, thereby reducing the sorting cost.

Finally, Figures 8(k)-8(l) show the average query latency of the four weight functions under varying $f_{\mathrm{T}}$. The two datasets showed different behaviors. This is because the query time depends heavily on the query text (which determines the number of collided compact windows).

## D Full Experiment of Evaluating Effectiveness

This section evaluates `MonoActive`'s effectiveness (precision, recall, F1) under different TF-IDF schemes on PAN dataset compared to `SeedExtension`. For unary IDF, we omit IDF weighting because it is constant 1. In this study, the other two IDF variants considered is the standard IDF and smooth IDF in Table 2, computed over the entire PAN dataset. The PAN dataset consists of manually annotated plagiarism cases, where each case specifies a suspicious text segment and its corresponding plagiarized segment. To adapt this

| Method | Precision / Recall / F1 | | | | | | |
|---|---|---|---|---|---|---|---|
| | $\theta = 0.05$ | $\theta = 0.1$ | $\theta = 0.15$ | $\theta = 0.2$ | $\theta = 0.3$ | $\theta = 0.4$ | $\theta = 0.5$ |
| Binary TF (set Jaccard) | 0.4335 / 0.9932 / 0.6035 | 0.4798 / 0.9625 / 0.6404 | 0.5206 / 0.8794 / 0.6540 | 0.5518 / 0.7016 / 0.6178 | 0.7300 / 0.4406 / 0.5495 | 0.7240 / 0.3326 / 0.4558 | 0.6697 / 0.1890 / 0.2948 |
| Binary TF+standard IDF | 0.4660 / 0.9248 / 0.6197 | 0.5579 / 0.7755 / 0.6489 | 0.6072 / 0.5748 / 0.5906 | 0.6293 / 0.4637 / 0.5340 | 0.6859 / 0.3560 / 0.4687 | 0.6359 / 0.2240 / 0.3313 | 0.7720 / 0.0486 / 0.0915 |
| Binary TF+smooth IDF | 0.4662 / 0.9991 / 0.6357 | 0.4701 / 0.8350 / 0.6015 | 0.5540 / 0.7192 / 0.6258 | 0.6026 / 0.5265 / 0.5620 | 0.6913 / 0.4019 / 0.5083 | 0.6432 / 0.2304 / 0.3393 | 0.6019 / 0.0527 / 0.0970 |
| Raw TF (multi-set Jaccard) | 0.4109 / **0.9988** / 0.5823 | 0.4248 / 0.9979 / 0.5960 | 0.4457 / 0.9961 / 0.6158 | 0.4822 / 0.9770 / 0.6457 | 0.6622 / 0.8339 / **0.7382** | 0.7848 / 0.6807 / 0.7291 | 0.7912 / 0.3898 / 0.5223 |
| Raw TF+standard IDF | 0.4793 / 0.8696 / 0.6180 | 0.5779 / 0.7842 / 0.6654 | 0.6655 / 0.5986 / 0.6303 | 0.6947 / 0.4890 / 0.5740 | 0.7161 / 0.3797 / 0.4963 | 0.6827 / 0.2473 / 0.3631 | 0.7787 / 0.0376 / 0.0718 |
| Raw TF+smooth IDF | 0.4176 / 0.9986 / 0.5890 | 0.4357 / 0.9847 / 0.6041 | 0.4562 / 0.9636 / 0.6192 | 0.4945 / 0.9263 / 0.6448 | 0.6937 / 0.7430 / 0.7175 | 0.7256 / 0.4285 / 0.5388 | 0.6624 / 0.2212 / 0.3316 |
| Log TF | 0.4226 / 0.9994 / 0.5940 | 0.4405 / 0.9501 / 0.6020 | 0.4615 / 0.7989 / 0.5851 | 0.4850 / 0.6621 / 0.5599 | 0.6894 / 0.4196 / 0.5217 | 0.7232 / 0.3093 / 0.4333 | 0.7238 / 0.0824 / 0.1479 |
| Log TF+standard IDF | 0.4787 / 0.9139 / 0.6283 | 0.5494 / 0.7112 / 0.6199 | 0.5992 / 0.5567 / 0.5771 | 0.6574 / 0.4867 / 0.5593 | 0.6877 / 0.3578 / 0.4707 | 0.6570 / 0.2038 / 0.3111 | 0.7699 / 0.0334 / 0.0640 |
| Log TF+smooth IDF | 0.4405 / 0.9989 / 0.6114 | 0.4613 / 0.8750 / 0.6041 | 0.5327 / 0.8002 / 0.6396 | 0.6287 / 0.6981 / 0.6616 | 0.7244 / 0.4764 / 0.5748 | 0.6245 / 0.2117 / 0.3163 | 0.7032 / 0.0629 / 0.1155 |
| Square TF | 0.4458 / 0.9742 / 0.6117 | 0.4441 / 0.9200 / 0.5990 | 0.4352 / 0.8617 / 0.5783 | 0.4639 / 0.8054 / 0.5887 | 0.4813 / 0.7139 / 0.5750 | 0.4491 / 0.5671 / 0.5012 | 0.4324 / 0.3987 / 0.4149 |
| Square TF+standard IDF | 0.5269 / 0.8129 / 0.6394 | 0.6202 / 0.7086 / 0.6615 | 0.7660 / 0.6213 / 0.6861 | 0.7763 / 0.5005 / 0.6086 | 0.8033 / 0.4337 / 0.5633 | **0.8508** / 0.2649 / 0.4040 | 0.6349 / 0.0663 / 0.1200 |
| Square TF+smooth IDF | 0.4437 / 0.9630 / 0.6075 | 0.4429 / 0.9238 / 0.5987 | 0.4443 / 0.8719 / 0.5886 | 0.4999 / 0.8217 / 0.6217 | 0.4873 / 0.7171 / 0.5802 | 0.4617 / 0.5718 / 0.5109 | 0.3957 / 0.3031 / 0.3433 |

**Table 5: Effectiveness under different similarity thresholds $\theta$ and TF-IDF weighting schemes. The effectiveness of `SeedExtension` is 0.3939 / 0.7070 / 0.5059.**

dataset to our near-duplicate alignment setting, for each source–suspicious pair, we treat the annotated suspicious segment as the query and the full source document as the source text.

By using the annotated plagiarism spans as the ground truth, we report the effectiveness of baseline `SeedExtension` and `MonoActive` under different TF-IDF strategies in Table 4. Notably, the effectiveness results of `AllAlign` are identical to those of Binary TF (set Jaccard) and Raw TF (Multi-set Jaccard) with unary IDF. This because `MonoActive` and `AllAlign` all rely on the min-hash scheme. If the random seeds are the same, they estimate the Jaccard similarity of subsequences identically and differ only in partitions.

As shown in Table 5, overall, the best F1 score is usually achieved by Raw TF. Specifically, smaller thresholds lead to consistently high recall but relatively low precision, while increasing $\theta$ improves precision at the cost of recall. This trend is expected, as stricter similarity thresholds reduce the number of candidate subsequences.

We first compare different TF variants and observe similar trends across thresholds. Binary TF, Raw TF and Log TF achieve relatively stable F1 scores over a broad range of thresholds, whereas Square TF favors higher recall at larger $\theta$ by amplifying frequent tokens. Binary TF and Raw TF exhibit extremely high recall across all thresholds, reaching nearly perfect recall at small $\theta$. However, this comes with limited precision, especially at larger thresholds. Overall, Raw TF consistently outperforms Binary TF. Notably, the highest F1 score 0.7382 is achieved by Raw TF under multi-set Jaccard at $\theta = 0.3$, highlighting the advantage of multi-set Jaccard similarity over the common set Jaccard similarity.

Incorporating IDF weighting substantially improves precision across all thresholds, leading to more balanced precision–recall trade-offs and higher F1 scores in most cases. For instance, at $\theta = 0.1$, Raw TF+IDF achieves an 11.5% relative improvement in F1 score compared to Raw TF.

We further observe that replacing standard IDF with Smooth IDF leads to more gradual and stable behavior across thresholds. Smooth IDF reduces extreme weight values, resulting in smoother trade-offs between precision and recall. Moreover, Smooth IDF becomes more advantageous at larger $\theta$. This is because standard IDF aggressively amplifies rare tokens, and missing such rare tokens causes the recall to decrease rapidly under stricter thresholds.

**Compare with `SeedExtension`.** `SeedExtension` has an F1 score of 0.5059, lower than that of most of `MonoActive`'s configurations. This is expected as `SeedExtension` is primarily designed for full-document–to–full-document comparison while our task focuses on matching short fragment queries against long source documents. The lack of context of query makes `SeedExtension` difficult to establish reliable seed matches, leading to limited performace.

# E Full Experiment of Comparing with State-of-the-Art

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

Figures 9(a)-9(c) show, when varying the text length $n$ from $10^3$ to $10^6$, the average number of compact windows generated by `MonoActive` and `AllAlign` (i.e., partition sizes, which are proportional to their index sizes) on three datasets OWT, PAN and NEWS. In the same figures, the reduction ratio, defined as $1 - \frac{\texttt{MonoActive}}{\texttt{AllAlign}}$, is overlaid as a purple line using a secondary $y$-axis.. As shown, `MonoActive` consistently generated less compact windows than `AllAlign`, up to 30.82%. More importantly, the gap between the number of compact windows for `MonoActive` and `AllAlign` steadily widens as $n$ increases. For example, when $n$ increased from 1000 to 1 million, the gap grew from 10.86% to 30.82% on PAN and from 7.69% to 26.08% on OWT. This is because `AllAlign`'s recursive approach divides the rectangles into multiple rectangular regions, imposing early boundaries that may split what is a single compact window in `MonoActive` into multiple windows. The number of compact windows generated by `MonoActive` grew linearly with $n$, which is consistent with our complexity analysis.

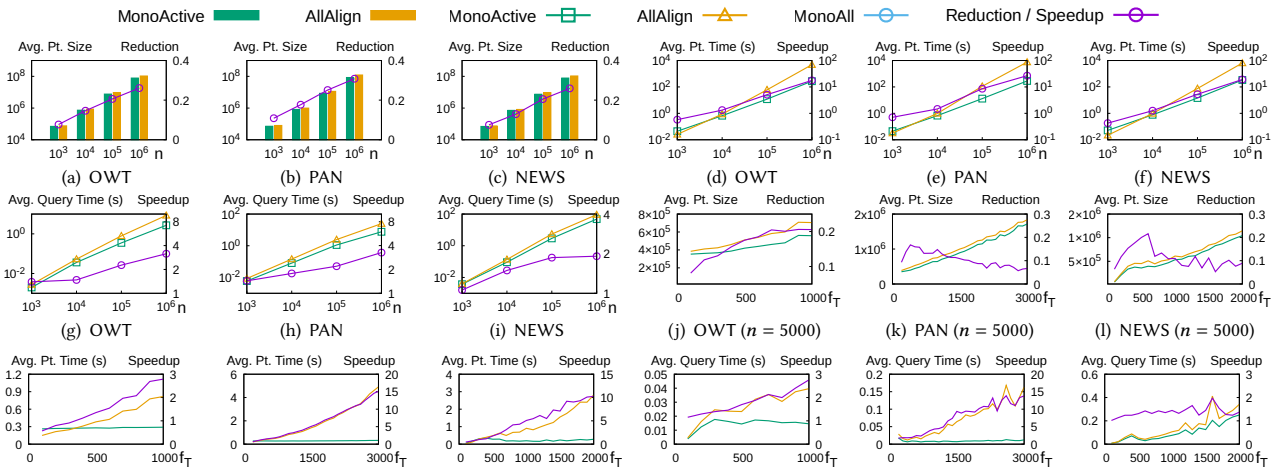
Figures 9(d)-9(f) show when varying the text length $n$ from $10^3$ to $10^6$, the average compact window generation time (i.e., partition time, which is proportional to the index time) of `AllAlign` and `MonoActive`, respectively. In the same figures, the purple line shows the speedup of `MonoActive` over `AllAlign`. `MonoActive` was significantly faster than `AllAlign`. For example, on the PAN dataset, when $n = 10^6$, `AllAlign` took 7,581s on average to generate the compact windows, whereas `MonoActive` only spent 284s, achieving a speedup of 26.7×. This is because `AllAlign` produces more compact windows than `MonoActive` and `AllAlign` is recursive whereas `MonoActive` is non-recursive. Moreover, the speedup of `MonoActive` over `AllAlign` increased as $n$ grew, e.g., on the PAN

dataset, when $n$ increased from $10^4$ to $10^6$, the speedup improved from 1.46 to 26.7. A similar trend is observed on OWT and NEWS.

Figures 9(g)-9(i) show the average query latency of the two methods when varying the text length $n$ from $10^3$ to $10^6$. In the same figures, the purple line shows the speedup of `MonoActive` over `AllAlign`. The queries are generated in the same way as described earlier. The query latency of `MonoActive` was up to 3.15× faster than that of `AllAlign`, though the number of compact windows was only up to 30.82% less. For example, on the PAN dataset, when $n$ increased from $10^3$ to $10^6$, the speedup improved from 1.4 to 3.15, exhibiting linear trend as $n$ grows. A similar trend is observed on OWT and NEWS. This is because, although the number of collided compact windows grows linearly with the number of generated compact windows in expectation, the cost of query processing scales quadratically with the number of collided windows in the worst case. Therefore, reducing the number of compact windows can lead to a significant speedup in query time.

We also evaluate the impact of the maximum token frequency $f_{\mathrm{T}}$. Figures 9(j)–9(r) present the number of compact windows generated and the compact window generation time , and the query latency of the two methods, under varying $f_{\mathrm{T}}$ with fixed text length $n = 5000$. By the same purple line, the relative improvements are marked in the same figures. As $f_{\mathrm{T}}$ increases, the generation time of `MonoActive` remains largely stable, whereas that of `AllAlign` increases significantly. For example, on the PAN dataset, when $f_{\mathrm{T}}$ increases from 600 to 3000, the speedup improved from 1.41 to 16.26. In addition, the gap in the number of compact windows produced by the two methods first grows and then decreases with larger $f_{\mathrm{T}}$. This trend arises because higher token frequencies lead to deeper recursion and more early boundaries in `AllAlign`, which fragment compact windows and increase computational cost. However, as $f_{\mathrm{T}}$ approaches $n$, the text instances approach the worst-case scenario and both methods produce a large number of compact windows. The gap of the query latency increases with the increase of $f_{\mathrm{T}}$. This is because the collided windows are dominated by tokens with the highest frequencies. In `AllAlign`, the high-frequency tokens trigger deeper recursion, resulting in more finely fragmented compact windows. These fragments increase the number of collided compact windows and the number of iterations in the inner loop. As a result, the query latency grows superlinearly with the number of collided compact windows.

*Scalability.* As shown in Figures 9(a)-9(i), the performance gain of `MonoActive` over `AllAlign` increased as the text length $n$ grew in terms of partition size, partition generation time, and query latency. Thus `MonoActive` scales better than `AllAlign`. This is attributed to the complexity guarantees of our algorithm.




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

(g) OWT (h) PAN (i) NEWS (j) OWT ($n = 5000$) (k) PAN ($n = 5000$) (l) NEWS ($n = 5000$)

(m) OWT ($n = 5000$) (n) PAN ($n = 5000$) (o) NEWS ($n = 5000$) (p) OWT ($n = 5000$) (q) PAN ($n = 5000$) (r) NEWS ($n = 5000$)

**Figure 9: Comparing with the state-of-the-art for multi-set Jaccard similarity. $k = 64$ in all experiments.**