Local Similarity Search for Unstructured Text

Pei Wang
Nagoya University, Japan
wang@db.ss.is.nagoya-u.ac.jp

Chuan Xiao*
Nagoya University, Japan
chuanx@nagoya-u.jp

Jianbin Qin
UNSW, Australia
jqin@cse.unsw.edu.au

Wei Wang
UNSW, Australia
weiw@cse.unsw.edu.au

Xiaoyang Zhang
UNSW, Australia
xyzhang@cse.unsw.edu.au

Yoshiharu Ishikawa
Nagoya University, Japan
y-ishikawa@nagoya-u.jp

ABSTRACT

With the growing popularity of electronic documents, replication can occur for many reasons. People may copy text segments from various sources and make modifications. In this paper, we study the problem of local similarity search to find partially replicated text. Unlike existing studies on similarity search which find entirely duplicated documents, our target is to identify documents that approximately share a pair of sliding windows which differ by no more than τ tokens. Our problem is technically challenging because for sliding windows the tokens to be indexed are less selective than entire documents, rendering set similarity join-based algorithms less efficient. Our proposed method is based on enumerating token combinations to obtain signatures with high selectivity. In order to strike a balance between signature and candidate generation, we partition the token universe and for different partitions we generate combinations composed of different numbers of tokens. A cost-aware algorithm is devised to find a good partitioning of the token universe. We also propose to leverage the overlap between adjacent windows to share computation and thus speed up query processing. In addition, we develop the techniques to support the large thresholds. Experiments on real datasets demonstrate the efficiency of our method against alternative solutions.

Keywords

local similarity search; unstructured text; prefix filtering; k-wise signature

1. INTRODUCTION

One of the main issues accompanying the growing popularity of electronic documents is the existence of replication. People may borrow or plagiarize text segments from various sources and make modifications. Due the need in many applications, e.g., plagiarism detection and near-duplicate Web page detection, identifying replications between documents has attracted remarkable attention from research community, and many approaches were proposed in the last two decades, e.g., by similarity search and join [27, 10, 3, 4, 35, 33] or document fingerprinting [25, 6, 8, 29, 30, 18, 31]. For the body of work in similarity search and join, documents are regarded as (multi)sets of tokens or strings, and pairs of documents are identified if they satisfy a similarity constraint. For document fingerprinting, documents are usually divided into overlapping or non-overlapping text segments and then identical or similar segments are identified. However, in many cases of replication, only a small part of a document is copied and text laundering may happen, e.g., by reorganizing sentences, replacing words with synonyms, changing word order, etc. These replications are hardly detected by similarity search and join approaches since these methods measures the similarities of entire documents, which are relatively low when only a small part is replicated. Document fingerprinting approaches are also likely to miss these results because they are either susceptible to small modifications [25, 6, 8] or do not have any guarantee when detecting similar segments [30, 29, 18, 31].

Seeing the limitations of the existing work, we propose to study the local similarity search problem for unstructured text. Given a collection of data documents and a query document, our goal is to find the data documents that share with the query a sliding window of size w but differ by a small number of tokens which are constrained by a threshold τ. Sliding windows can effectively capture partial replications. We regard sliding windows as multisets of tokens and tolerate errors so that replications can be detected in spite of text laundering. Unlike the document fingerprinting methods without guarantee when modifications exist, we investigate exact solutions to the local similarity search problem.

An immediate solution to the problem is materializing all the windows in the documents as individual objects and invoking a set similarity join on two sets of windows, one from the data documents and the other from the query document. Many prevalent set similarity join methods are based on prefix filtering [10, 4, 35, 33] to find candidates that satisfy a necessary but less strict condition of the similarity constraint and then verify these candidates. Tokens in each object are sorted by a global order, and two objects must share a required amount of tokens in their first few tokens, called prefix, to become a candidate. To find candidates, an inverted index is built to map each token to a list of objects that have this token in their prefixes. Tokens are usually sorted by the order of increasing document frequency so that prefixes are composed

*corresponding author

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

ISBN 978-1-4503-3531-7/16/06...$15.00

DOI: http://dx.doi.org/10.1145/2882903.2915211

of rare and hence selective tokens for fast query processing. However, for local similarity search, windows are much shorter than entire documents and thus the tokens in their prefixes are not so selective. In this case, they either have to use short prefixes but end up with large number of candidates, or use long prefixes but spend more in candidate generation due to the access to long postings lists in the inverted index. Either will result in poor performance. Another issue is that it is unknown how to share computation between overlapping windows by these methods, because prefixes can be different for adjacent windows though they share most of the tokens.

In this paper, we propose to solve the local similarity search problem by a novel way of leveraging prefix filtering. Unlike all the existing prefix filtering-based methods that take single tokens to index, we propose to index *k-wise signatures* which are combinations of k tokens in the prefix. Combining tokens significantly improves the selectivity and thus enables us to use long prefixes while accessing short postings lists. Since enumerating all possible k token combinations in the prefix results in large combination number and hence time-consuming signature generation, we divide the token universe into several partitions according to frequency and use different k's across these partitions. The corresponding prefix filtering condition is developed for k-wise signatures with the partitioning technique. The query processing cost is analyzed, and based on the cost model we propose a practical algorithm to find a good partitioning of the token universe. To take advantage of overlap between adjacent windows, we study how prefixes and candidates change for sliding windows, thereby developing an *interval sharing* technique to avoid unnecessary prefix computation and candidate generation as well as verification. For the case of large thresholds which may cause large number of combinations, we propose to further divide partitions and thus the combination number is reduced to be proportional to $\tau + 1$. Experiment results on publicly available datasets show that our method has superior performance to alternative solutions with up to 12x speedup.

We also note that tolerating errors in sliding windows will increase false positive for the task of detecting partial replications, whereas we aim at developing an efficient method to increase the recall. Additional post processing methods can be applied for the sake of high precision.

Our contributions can be summarized as:

- We study the problem of local similarity search to find sliding windows with a small amount of differences in unstructured text. It can capture partial replications with minor modifications which are hard to be detected by existing methods.
- We propose a prefix filtering-based method and address the major technical issue in adapting prefix filtering for local similarity search. Combinations of tokens are utilized for fast query processing and the token universe is partitioned to reduce the combination number. We propose a cost model based on which a practical partitioning algorithm is devised.
- We exploit the sharing of computation between adjacent windows to efficiently compute prefixes and candidates and perform verification for sliding windows.
- We conduct extensive experiments on real datasets. The proposed method is shown to be faster than alternative methods by up to an order of magnitude.

The rest of the paper is organized as follows: Section 2 defines the problem and introduces preliminaries. Section 3 presents the k-wise signature method with partitioning. Section 4 elaborates the interval sharing technique to share computation for overlapping windows. Cost analysis and token universe partitioning are covered by Section 5. Section 6 presents the technique to cope with large thresholds. Experimental results and analyses are reported in Section 7. Section 8 reviews related work. Section 9 concludes the paper.

2. PRELIMINARIES

2.1 Problem Definition

A document is defined as a sequence of tokens drawn from a finite universe $\mathcal{U} = \{ t_1, \ldots, t_{|\mathcal{U}|} \}$. A token can be a word, a q-gram, etc. In our examples, we tokenize documents with whitespace as delimiters, but our algorithms are independent of the tokenization scheme. A window of size w is w consecutive tokens in a document d. $d[i]$ denotes the i-th token in d. $W(d, i)$ denotes the window starting with $d[i]$. We use the notation $x \sqsubseteq d$ to denote that x is a window of d. By neglecting the order of tokens in the original document, a window is transformed into a *multiset* of tokens. The overlap similarity measures the intersection of tokens in two windows x and y; i.e., $O(x, y) = |x \cap y|$. Note that multiplicities are considered when computing the overlap similarity. Let $mul(t, x)$ denote the multiplicity (number of occurrences) of a token in a window x. The multiplicity of t in $x \cap y$ is the smaller of $mul(t, x)$ and $mul(t, y)$. E.g., $\{ A, A, A, B \} \cap \{ A, A, B, B \} = \{ A, A, B \}$. If a window is drawn from a data document we call it a data window, and if it is drawn from a query document we call it a query window. The problem of local similarity search is defined as follows.

PROBLEM 1. *Given a collection of data documents $\mathcal{D}$, a query document q, a window size w, and a threshold θ, the problem of local similarity search is to find all pairs of windows $\langle x, y \rangle$, such that x is a data window, y is a query window, and their intersection is at least θ; i.e., $\{ \langle x, y \rangle \mid x \sqsubseteq d_i, d_i \in \mathcal{D}, y \sqsubseteq q, O(x, y) \geq \theta \}$.*

We may also define the threshold with dissimilarity; i.e., $\tau = w - \theta$, and our goal is to find the pair of windows $\langle x, y \rangle$ such that $w - O(x, y) \leq \tau$. For ease of exposition, we use the τ threshold instead of θ in the rest of the paper.

EXAMPLE 1. *Consider a data document d and a query document q,*

$$d = \text{“the lord of the rings”},$$
$$q = \text{“the lord and the kings”}.$$

$w = 4$, and $\tau = 1$. With the word-to-token mapping table,

Word	*the*	*lord*	*of*	*rings*	*and*	*kings*
Token	*A*	*B*	*C*	*D*	*E*	*F*
Window Freq.	*2*	*2*	*2*	*1*	*0*	*0*

The data document has two windows

$$W(d, 1) = \{ A, B, C, A \},$$
$$W(d, 2) = \{ B, C, A, D \}.$$

The query document has two windows

$$W(q, 1) = \{ A, B, E, A \},$$
$$W(q, 2) = \{ B, E, A, F \}.$$

$\langle W(d, 1), W(q, 1) \rangle$ is returned as the result of local similarity search because the $w - O(x, y) = 4 - 3 = 1 \leq \tau$.

2.2 Prefix Filtering

Similarity join [10] is an operation to take two relations as input and return pairs of objects from each relation that satisfy a similarity constraint. One may regard each window as an individual object and convert the local similarity search to a set similarity join on two relations $\mathcal{R}$ and $\mathcal{S}$, which consist of all the windows from the data and the query documents, respectively. Since similarity computation for all pairs of objects is time-consuming, many prevalent set similarity join algorithms are based on the filter-and-refine scheme to generate a set of promising candidates that satisfy necessary conditions for the similarity constraint and then verify them by similarity computation. Many of them [10, 4, 35, 33] utilize the prefix filtering principle for fast query processing:

LEMMA 1 (PREFIX FILTERING). *Consider two multisets x and y of size w. Both are sorted in a global order $\mathcal{O}$. Let the prefix of x be the first $(\tau+1)$ tokens in x. If $w - O(x,y) \leq \tau$, the prefixes of x and y must share at least one token.*

For the global order $\mathcal{O}$, prefix filtering-based methods suggest sorting by increasing order of document frequency. In this way, prefixes tend to be composed of rare tokens, and thus the number of objects that share at least a token in prefixes (called candidates) tends to be small. For local similarity search, since each window is treated as an object, we sort the tokens in increasing order of window frequency, i.e., the number of windows in *data documents* that contain the token, and break tie by lexicographical order (after tokenization) and then increasing order of their positions in the original document. In this paper, we let $\mathcal{O}$ be this order unless otherwise stated. Consider a window x whose tokens are sorted by $\mathcal{O}$. $x[i]$ denotes the i-th token in x. $x[i\,..\,j]$ denotes the multiset of tokens from the i-th token to the j-th token in x. Given two tokens t_1 and t_2, $t_1 < t_2$ if t_1 precedes t_2 in $\mathcal{O}$.

In [35, 33], the prefix filtering is extended to k-prefix:

LEMMA 2 (EXTENDED PREFIX FILTERING). *Consider two multisets x and y of size w. Both are sorted in a global order $\mathcal{O}$. Let the k-prefix of x be the first $(\tau+k)$ tokens in x. If $w - O(x,y) \leq \tau$, then the k-prefixes of x and y must share at least k tokens.*

The condition of the k-prefix case is stricter than the 1-prefix case, and hence reduces the candidate number. In addition, an adaptive approach was proposed in [33] to optimize query processing performance by selecting an appropriate prefix length for each object using a cost model.

EXAMPLE 2. *Consider the windows and the window frequency table (note that window frequency only counts occurrences in data windows) in Example 1. $\mathcal{O}$ is $E < F < D < A < B < C$. We sort the tokens in each window in this order:*

$$W(d,1) = [\underline{A, A, B}, C],$$
$$W(d,2) = [\underline{D, A, B}, C],$$
$$W(q,1) = [\underline{E, A, A}, B],$$
$$W(q,2) = [\underline{E, F, A}, B].$$

The underlined tokens are 2-prefixes of these windows. $W(d,1)$ and $W(q,1)$ satisfy the similarity constraint. They share two tokens (two A's) in their 2-prefixes.

With the (extended) prefix filtering principle, one can design a similarity join-based algorithm [1] for local similarity search. The algorithm consists of two parts: the indexing part and the query processing part. In the indexing part, for each window in $\mathcal{R}$, the tokens in the prefix (can be 1-prefix, k-prefix, or adaptive prefix) are extracted, each token regarded as a *signature*. An inverted index is built offline, mapping each signature s to a list (called postings list) of windows whose prefixes contain s. In the query processing part, the windows in $\mathcal{S}$ are processed one by one in an index nested loops join manner, and there are three phases: (1) In the *signature generation phase*, signatures are generated in the same way as in the indexing part. (2) In the *candidate generation phase*, the inverted index is probed to find *candidate windows*, i.e., the windows in $\mathcal{R}$ that share required amount of tokens with the query window in their prefixes. (3) In the *verification phase*, candidate windows are verified and added to the result if they meet the similarity condition.

There are two main drawbacks of the similarity join-based algorithm: (1) Compared with similarity join on entire documents, windows size is smaller in local similarity search. Prefixes are likely to contain frequent tokens that are *not selective*, and this will result in the following dilemma: We either use short prefixes but end up with a time-consuming verification phase due to large number of candidate windows, or use long prefixes but need to access long postings lists which poses considerable overhead in the candidate generation phase. (2) Overlap exists between adjacent windows but they are processed individually without any share of computation, e.g., common tokens in prefixes as well as verification of adjacent windows. We proposed a new method to address the two issues in the next two sections.

3. k-wise SIGNATURE SCHEME

3.1 Combination of k Tokens

Unlike the similarity join-based algorithm that regards *single tokens* as signatures, e.g., [35, 33], we apply the prefix filtering in another way. Recall that extended prefix filtering requires that candidate windows share at least k tokens in their prefixes. For the k-prefix of each window, we pick the combination of k tokens in every possible way and generate signatures. An inverted index is built to map each signature to a list of windows that contain the signature, i.e., all the k tokens, in their prefixes. We use the index to find windows that share a common signature, hence at least k tokens in their prefixes. Since there are $(\tau+k)$ tokens in the k-prefix, the number of signatures for each window is $\binom{\tau+k}{k}$. We call this type of signatures *k-wise signatures*. Compared to single tokens, k-wise signatures are usually more selective and yield shorter postings lists in the index, thereby reducing the cost in the candidate generation phase. When $k = 1$, k-wise signatures become single tokens and the method is equivalent to standard prefix filtering (Lemma 1).

EXAMPLE 3. *Consider the four windows in Example 2. $\tau =$*

[1]Despite solving a search problem, we call it a join-based algorithm because it converts the search problem to a problem of joining two relations of windows.

1, *and* $k = 2$. *The signatures for these windows are* [2]*:*

$$S_{W(d,1)} = \{ AA, AB, AB \},$$
$$S_{W(d,2)} = \{ DA, DB, AB \},$$
$$S_{W(q,1)} = \{ EA, EA, AA \},$$
$$S_{W(q,2)} = \{ EF, EA, FA \}.$$

$W(d,1)$ *and* $W(q,1)$ *share a common signature* AA*, and therefore share at least two tokens in their prefixes.*

We compare the cost in the candidate generation phase. Using single tokens, the postings list of token A *has two entries* $W(d,1)$ *and* $W(d,2)$*;* E *and* F *are not in the index. To process* $W(q,1)$ *and* $W(q,2)$*,* $2 + 2 = 4$ *entries are accessed. Using 2-wise signatures, the postings list of* AA *has one entry* $W(d,1)$*;* EA*,* EF*, and* FA *are not in the index. To process* $W(q,1)$ *and* $W(q,2)$*, 1 entry is accessed, and the cost is reduced by 3 from the single token case.*

For the choice of k, a larger k decreases candidate generation cost as well as verification cost because signatures are more selective, resulting in shorter postings lists and less number of candidates. On the other hand, it increases signature generation cost due to more token combinations. According to our experiment results, setting k to 3 yields the best runtime performance for most w and τ settings.

3.2 Partition of Token Universe

Although using k-wise signatures reduces cost in the candidate generation phase, it increases the cost in the signature generation phase due to the enumeration of token combinations in prefixes. This renders it unable to scale up when τ or k increases. To remedy this, we observe that due to the power law distribution of token frequencies, the rarest tokens in prefixes are selective enough, and there is no need to combine them to k-wise signatures. Only the relatively frequent tokens (they are still uncommon when compared with the most popular tokens in a window) in prefixes need to be combined, and considering their frequencies we may use different k's for different tokens. This inspires our idea of partitioning token universe.

Consider a token universe $\mathcal{U}$ sorted by $\mathcal{O}$. It is divided into $k_{\max}$ disjoint partitions (empty partitions are allowed). For the tokens in the i-th partition, i-wise signatures are used and combinations are only generated from within. Note that if a window has less than i tokens in the i-th partition, we do not generate signatures for these tokens in this window. Intuitively, the rarest tokens are indexed in single tokens, while the most frequent tokens are indexed in $k_{\max}$-wise signatures. We say a token t's class (denoted by $class(t)$) is i if t is in the i-th partition of $\mathcal{U}$. We call this type of signatures partitioned k-wise signatures.

For the above partitioned k-wise signatures, we define candidate windows as the data windows that share with the query window at least one common signature generated from their respective prefixes. The prefix length for partitioned k-wise signatures needs to be computed, considering the possibility that a window may contain tokens in different classes. Since a pair of windows $\langle x, y \rangle$ satisfying the similarity constraint may differ by at most τ tokens, if a token of x is not in y, we call the token an *error* in x. Our goal is to find the shortest lengths l_x of x and l_y of y such that if there is no common signature generated from the tokens in $x[1..l_x]$ and $y[1..l_y]$, it will incur at least $\tau + 1$ errors (in both x and y).

[2]We do not remove duplicate signatures because they are necessary to the correctness of the interval sharing technique which will be presented in Section 4.

Algorithm 1: PrefixLength (x, τ)

```
1 cov = 0, n_i ← 0(1 ≤ i ≤ k_max);
2 for l = 1 to w do
3     i ← x[l]'s class;
4     n_i ← n_i + 1;
5     if n_i ≥ i then
6         cov ← cov + 1;
7         if cov = τ + 1 then break;
8 return l
```

We say a signature is *affected* by an error if the signature contains the error. Given a multiset of tokens, its *coverage* is defined as the minimum amount of errors required to affect all the signatures enumerated from these tokens.

LEMMA 3. *Consider* n_i *tokens in class* i*. The coverage of these tokens is captured by the following equation*

$$cov(i) = \begin{cases} n_i - i + 1 & , \text{if } n_i \geq i \\ 0 & , \text{otherwise.} \end{cases} \quad (1)$$

PROOF. For the case when $n_i \geq i$, if the number of errors is less than $(n_i - i + 1)$, there will be at least i common tokens, hence at least one common signature not affected. For the other case, since no signature can be generated, no error is needed. □

It can be seen that two signatures generated from different classes do not have any common tokens. Hence we have the following lemma.

LEMMA 4. *Consider a multiset which has* n_i *tokens in each class* i *(*$1 \leq i \leq k_{\max}$*). The coverage of these tokens is* $\sum_{i=1}^{k_{\max}} cov(i)$*.*

With the above lemma, we can design an algorithm (Algorithm 1) to compute the prefix length of a window x, whose tokens are already sorted by $\mathcal{O}$. First, the algorithm initializes as zero the coverage and a counter n_i for each class i. Then it iterates through the tokens in x. For each token $x[l]$ and its class i, it increments the corresponding counter n_i. The coverage is incremented if $n_i \geq i$. When the coverage reaches $\tau + 1$, it returns the current value of l as the prefix length. Although when $n_i < i$ no i-wise signatures are generated from the n_i tokens, these tokens are included in the prefix so that the prefix is still continuous [3]. The algorithm correctly computes the prefix length, as stated by the following theorem (proof is provided in Appendix B).

THEOREM 1. *Let* l_x *denote the prefix length of a window* x*, as output by Algorithm 1, and* S_x *be the set of signatures generated with the tokens in* $x[1..l_x]$*. If* $w - O(x, y) \leq \tau$*,* $S_x \cap S_y \neq \emptyset$*.*

EXAMPLE 4. *Consider the window in Figure 1.* $\tau = 3$*. We need a total coverage of 4. The number of tokens in the first three classes are 1, 3, and 1, respectively. Their coverages are 1, 2, and 0, respectively, which sum up to 3. Besides these five tokens, we need the first four tokens in class 4 to make the total coverage* $\tau + 1$*. Therefore the prefix length is 9.*

[3]Including these tokens into the prefix is also necessary to guarantee the correctness of the interval sharing in Section 4.

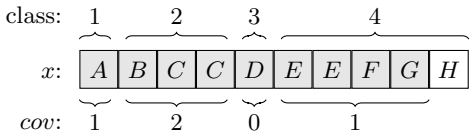


Figure 1: Example of Prefix Length (prefix tokens are shaded)

Algorithm 2: PartitionedKWise $(\mathcal{R}, \mathcal{S}, w, \tau, k_{\max})$

```
1  T ← ∅, I_i ← ∅;
2  foreach x ∈ R do
3      S ← GenSignature(x, τ, k_max);
4      foreach s ∈ S do
5          I_s ← I_s ∪ { x } ;            /* insert into index */
6  foreach y ∈ S do
7      A ← ∅;
8      S ← GenSignature(y, τ, k_max);
9      foreach s ∈ S do
10         foreach x ∈ I_s do
11             A ← A ∪ { x } ;            /* find a candidate */
12     foreach x ∈ A do
13         if w − O(x, y) ≤ τ then
14             T ← T ∪ { ⟨x, y⟩ };
15 return T
```

Supposing checking a token's class spends $O(1)$ time, the time complexity of Algorithm 1 is $O(l)$. Moreover, the prefix length of a window is upper-bounded by the following corollary, as derived from Lemmas 3 and 4.

COROLLARY 1. *A window's prefix length does not exceed* $\tau + 1 + \frac{k_{\max}(k_{\max}-1)}{2}$.

The upper bound is tight, because when there are $i-1$ tokens in class i ($1 \le i \le k_{\max}-1$), the prefix length reaches the upper bound. Corollary 1 also yields an upper-bound of the time complexity of Algorithm 1, which is $O(\tau + k_{\max}^2)$.

We call a token t a *covering* token if the coverage of the tokens in $class(t)$ is above zero, or a *non-covering* token otherwise. The following corollary states that the tokens in the highest class in the prefix are covering tokens.

COROLLARY 2. *Let h be the highest class in the prefix: $h = \max\{\, i \mid n_i > 0, 1 \le i \le k_{\max} \,\}$. The coverage of the tokens in class h is above zero.*

By replacing single tokens with partitioned k-wise signatures, we devise an algorithm (Algorithm 2) for local similarity search. Partitioned k-wise signatures are generated for indexing (Line 3) and signature generation (Line 8) by calling Algorithm 3, which computes the prefix length of a window and then combines tokens in each class i as i-wise signatures. In candidate generation, for each signature generated from a query window's prefix, we probe the inverted index and store the candidate windows in a set (Line 11). Then the candidate windows are verified by the similarity constraint (Line 13).

By Corollary 1, the completeness of the algorithm is stated by the following theorem.

THEOREM 2. *Algorithm 2 is complete and does not miss any result of local similarity search when* $w \ge \tau + 1 + \frac{k_{\max}(k_{\max}-1)}{2}$.

When $k_{\max} = 1$, only single tokens are used to generate signatures, and the prefix length returned by Algorithm 1 is exactly $\tau + 1$. Therefore, using standard prefix filtering is a special case of the partitioned k-wise algorithm.

Algorithm 3: GenSignature $(x, \tau, k_{\max})$

```
1  S ← ∅;
2  l ← PrefixLength(x, τ);
3  for i = 1 to k_max do
4      P_i ← { t | t ∈ x[1..l] ∧ class(t) = i } ;   /* a multiset */
5      S ← S ∪ { i-wise signatures generated from tokens in P_i };
6  return S
```

We analyze the cost of the partitioned k-wise algorithm and compare it with the algorithm generating the same set of candidate windows using single tokens, i.e., building index with single tokens for all the classes and finding the pairs of windows such that there exists a token class i in their prefixes where at least i tokens are shared. Assume that the cost of generating an i-wise signature is i and the cost of accessing each entry in a postings list is 1. For signature generation, the cost of the single token algorithm is $\sum_{i=1}^{k_{\max}} n_i$ and the partitioned k-wise algorithm is $\sum_{i=1}^{k_{\max}} i \cdot \binom{n_i}{i}$. Assume tokens are independent and the average length of postings list is $|\mathcal{R}|f_i$ for a single token in class i. For an i-wise signature, the expected length of its posting list is $|\mathcal{R}|(f_i)^i$. Hence for candidate generation, the cost of the single token algorithm is $\sum_{i=1}^{k_{\max}} n_i \cdot |\mathcal{R}|f_i$, and the partitioned k-wise algorithm is $\sum_{i=1}^{k_{\max}} \binom{n_i}{i} \cdot |\mathcal{R}|(f_i)^i$. The partitioned k-wise algorithm spends more on signature generation than the single token algorithm when $k_{\max} > 1$ and $n_i > i$, but saves candidate generation cost when $f_i < (\frac{n_i}{\binom{n_i}{i}})^{\frac{1}{i-1}}$.

The query processing performance of the partitioned k-wise algorithm depends on the partitioning of token universe. We leave this problem to Section 5 and investigate how to utilize the overlap of adjacent windows first.

4. INTERVAL SHARING

4.1 Signature Generation

Two adjacent windows $w(d,i)$ and $w(d,i+1)$ share $w-1$ tokens. It is very likely that they share most tokens in the prefixes. This motivates us to share signature generation for the adjacent windows.

We say a window x contains a signature s if s is generated from x's prefix. Instead of mapping a signature to a list of windows in the index, we choose to map a signature to a list of window intervals that contain this signature. A window interval is in the form of $d[u,v]$, representing that all the windows between $W(d,u)$ and $W(d,v)$, including both, contain the signature. For the sake of query processing performance, an interval is to be maximal; i.e., neither $W(d,u-1)$ nor $W(d,v+1)$ contains the signature. We slide through the document, open an interval of the signature when we reach $W(d,u)$, and close the interval when we leave $W(d,v)$. Then the interval $d[u,v]$ is inserted to the postings list of the signature.

For the first window of a document, its prefix is computed and signatures are generated by Algorithm 3. An interval is opened for each of these signatures. When a window slides to the next one, it can be observed that if none of the tokens in the prefix changes, no signature changes. Therefore intervals are only opened or closed when any token changes in the prefix. The intervals of signatures are all closed when we leave the last window of a document.

Based on the above observation, we design a prefix maintenance algorithm (pseudo-code provided in Appendix A). The basic idea is that when the window slides, we can exploit the current prefix to compute the new one rather than starting from scratch. Suppose x and x' are two adjacent windows. P denotes the prefix of x, whose length is l, and its coverage is denoted by $cov(P)$. Let t_1 be the first token of x and t_2 be the last token of x'; i.e., t_1 is the outgoing token and t_2 is the incoming token when the window slides. The algorithm takes x, l, t_1, and t_2 as input, and outputs a quadruple (x', l', S_o, S_c), where l' denotes the prefix length of x', and S_o and S_c denote the multisets of the signatures whose intervals are opened and closed while x slides to x', respectively. To compute the new prefix, our idea is to delete t_1 if it is in the current prefix, and insert t_2 if it precedes some token in the current prefix. Then we compute the coverage (either τ, $\tau+1$, or $\tau+2$) of the resulting prefix, and recover it to $\tau+1$ if not equal. To this end, we propose the notion of *temporary* prefix, denoted by P', to capture how the prefix changes with the slide. P' is first initialized as P. Then t_1 is deleted from P' if t_1 is a token of P', and t_2 is inserted into P' if it precedes the last token of P' in terms of the global order $\mathcal{O}$. Then we cope with the coverage of the resulting P'. The procedure is divided into the following cases.

- If $cov(P \backslash t_1) < \tau + 1$, the coverage is below $\tau+1$ when t_1 is deleted. We check whether t_2 recovers the coverage to $\tau+1$.

  If so, it means that t_1 is replaced by t_2 in the prefix. Due to Corollary 2, we remove from P' the tokens in the highest class if they are non-covering, because t_1 may be in the highest class of P and its removal may cause the other tokens in this class to be non-covering. If $t_1 \neq t_2$, we close intervals for the signatures that contain t_1, generate signatures by combining t_2 and the tokens in the same class in P', and open intervals for them.

  Otherwise, it means that the removal of t_1 reduces the coverage but the inclusion of t_2 does not recover it. We need to include more tokens (denoted by ΔP) into the prefix to increase the coverage to $\tau+1$. If ΔP has tokens other than t_1 (note that t_1 may be a token of x' as well due to the multiplicity), we close intervals for the signatures that contain t_1, generate signatures by combining every token in ΔP and those in the same class in P', and open intervals for them.
- If $cov(P \backslash t_1) = \tau + 1$, either t_1 is not in the prefix or the coverage remains $\tau+1$ when it is deleted. We check whether the coverage of P' exceeds $\tau+1$ due to t_2.

  If so, it means that the removal of t_1 does not affect the coverage but the inclusion of t_2 make it exceed $\tau+1$. We need to remove tokens (denoted by ΔP) from P' to decrease the coverage to $\tau+1$. The tokens in the highest class of P' are then removed if they are non-covering tokens. If ΔP has tokens other than t_2, we close intervals for the signatures that contain any token in ΔP, generate signatures by combining t_2 and the tokens in the same class in P', and open intervals for them.

  Otherwise, it means that neither t_1 and t_2 affects the coverage. No interval changes in this case.

EXAMPLE 5. *Figure 2 shows an example of the maintenance of prefix. $w = 4$, and $\tau = 1$. d is a document consisting of seven tokens. Suppose tokens are sorted in alphabetical order. A, B, C, and D are class 1 tokens. E, F, and G are class 2 tokens. Tokens in the prefix are underlined for each window.*

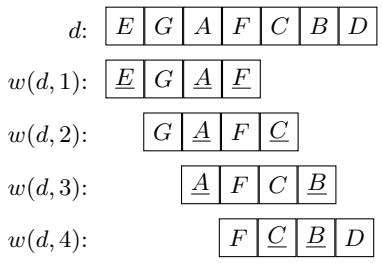


Figure 2: Example of Prefix Maintenance

Starting with $W(d,1)$, the prefix is $\{A, E, F\}$. Signatures A and EF are generated, and intervals are opened for them. When the window slides to $W(d,2)$, E leaves the window. Since it is in the prefix of $W(d,1)$, the temporary prefix becomes $\{A, F\}$. C enters the window, and it is inserted into the temporary prefix because $C < F$. Now the coverage is $\tau+1$. We remove the highest class which consists of a non-covering token F. Hence the prefix of $W(d,2)$ is $\{A, C\}$. Because $E \neq C$, the interval of EF is closed, and the interval of C is opened. For $W(d,3)$, G leaves the window and it is not in the prefix of $W(d,2)$. The new token B is inserted into the temporary prefix because $B < C$. Since the coverage is $\tau+2$, C is removed to recover the coverage to $\tau+1$. Hence the prefix of $W(d,3)$ is $\{A, B\}$. The interval of C is closed, and the interval of B is opened. For $W(d,4)$, A leaves the window and it is in the prefix of $W(d,3)$. The temporary prefix becomes $\{B\}$. Then D is not inserted because $D \not< B$. Since the coverage is τ, C is included to recover it to $\tau+1$. Hence the prefix of $W(d,4)$ is $\{B, C\}$. Because $A \neq C$, the interval of A is closed, and the interval of C is opened. When reaching the end of d, we close the intervals of B and C.

The signatures and their corresponding intervals are A : $\{d[1,3]\}$, EF : $\{d[1,1]\}$, C : $\{d[2,2], d[4,4]\}$, B : $\{d[3,4]\}$.

To efficiently implement the prefix maintenance algorithm, the tokens of window x can be stored in a binary search tree, so that the deletion of t_1 and the insertion of t_2 take $O(\log w)$ time. We do not materialize the prefix P or P' but record the prefix length l and l'. Hence checking if a token is in P' takes $O(\log w)$ time and any update on P' takes $O(1)$ time. By keeping track of the number of tokens in each class, any coverage-related operation takes $O(1)$ time. The loop of deleting the highest class from the temporary prefix is repeated at most ($k_{\max}$ - 1) times, hence taking $O(k_{\max})$ time in the worst case and $O(1)$ time in the average case. Ignoring the cost of generating signatures, the time complexity is $O(k_{\max}+\log w)$ in the worst case and $O(\log w)$ in the average case.

A subtle case is that the multiplicity of a token in the prefix may cause duplicate signatures in a window, and hence multiple opens or closes of an interval. E.g., $k_{\max} = 1$, and the prefixes of windows $W(d,1)$ to $W(d,5)$ are $\{A, B\}$, $\{A, A\}$, $\{A, B\}$, $\{A, B\}$, and $\{B, C\}$, respectively. The interval of signature A is opened at $W(d,1)$ and $W(d,2)$, and closed at $W(d,2)$ and $W(d,4)$, hence resulting in two overlapping intervals $d[1,4]$ and $d[2,2]$ which violate the maximality of an interval. To handle this case, we use a counter γ to store the times an interval has been opened. When it is opened for the first time, γ is initialized as 1. It is increased by 1 when an open occurs, and decreased by 1 when a close occurs. Only the first open (when $\gamma = 1$) and the last close (when $\gamma = 0$) are treated as "true" open and close of an interval, while the others are treated as "false" opens or closes. We

only insert the interval into the index for a true close. In the above example, a true open occurs at $W(d,1)$ and a true close occurs at $W(d,4)$, hence resulting in a maximal interval $d[1,4]$. In the rest of the paper, we mean a true open or close when referring to an open or close of an interval unless otherwise noted, and the output of the prefix maintenance algorithm only involves signatures with true open or close intervals.

4.2 Candidate Generation

By utilizing index on window intervals, for a query window, we generate candidates in the form of $d[u,v]$ (called *candidate interval*), indicating that all the windows between $W(d,u)$ and $W(d,v)$, including both, share at least a signature with the query window in their prefixes.

We use the same method to generate signatures for both indexing and query processing. It can be seen that for two consecutive query windows $W(q,i)$ and $W(q,i+1)$, their candidate intervals are the same if the two windows generate the same set of signatures. We exploit this property and devise the partitioned k-wise algorithm equipped with interval sharing. The pseudo-code is shown in Algorithm 4.

To index data documents, it iterates through the data documents in $\mathcal{D}$, computes prefix and generates signatures for the first window of each document by Algorithms 1 and 3 (Line 4), and processes the other windows by the prefix maintenance algorithm (Line 6). Signatures are generated and corresponding window intervals are indexed (Line 8).

To process queries, we also call Algorithms 1 and 3 to compute prefix and generate signatures for the first query window (Line 10), and call the prefix maintenance algorithm to handle the other windows (Line 13). The candidate intervals of a query window $W(q,i)$ is stored in a multiset A_i. With the set of signatures whose intervals are open for the first query window, as returned by Algorithm 3, for each signature s in the set we probe the index to retrieve its (data) window intervals and insert them into A_1 (Line 11). For the other query windows, we monitor S_o and S_c output by the prefix maintenance algorithm, which are the multisets of the signatures whose intervals are opened and closed, respectively, while the query window slides. If S_o and S_c are both empty, indicating that $W(q,i)$ and $W(q,i+1)$ generate the same set of signatures, then $A_{i+1} = A_i$ (Line 14). Otherwise, we let $A_{i+1} = A_i$, probe the index for each signature in S_o (resp. S_c), and then insert into (resp. delete from) A_{i+1} the (data) window intervals retrieved from the index (Lines 15 – 16). Finally, we merge intervals in each A_i to eliminate the overlap among candidate intervals (Line 18) and perform verification (Line 20).

EXAMPLE 6. *Consider an index mapping two signatures s_1 and s_2 to the window intervals of d_1 and d_2:*

$$I_{s_1} = \{\, d_1[11,13], d_2[13,15] \,\},$$
$$I_{s_2} = \{\, d_1[12,14], d_2[11,14] \,\}.$$

Consider a query of three windows. Suppose both $W(q,1)$ and $W(q,2)$ generate a signature s_1, and $W(q,3)$ generates two signatures s_1 and s_2. Before merging, the candidate intervals of the three query windows are:

$$A_1 = \{\, d_1[11,13], d_2[13,15] \,\},$$
$$A_2 = \{\, d_1[11,13], d_2[13,15] \,\},$$
$$A_3 = \{\, d_1[11,13], d_2[13,15], d_1[12,14], d_2[11,14] \,\}.$$

We obtain A_1 by probing the postings list of s_1, and let $A_2 = A_1$ because they generate the same signature. For A_3, we probe the index and insert the window interval of s_2. After merging, A_3 becomes $\{\, d_1[11,14], d_2[11,15] \,\}$.

Algorithm 4: PartitionedKWiseInterval $(\mathcal{D}, q, w, \tau, k_{\max})$

```
1  T ← ∅, I_i ← ∅;
2  foreach d ∈ D do
3      x ← W(d,1);
4      l ← PrefixLength(x,τ), S_o ← GenSignature(x,τ,k_max);
5      for i = 1 to |d| do
6          (x,l,S_o,S_c) ← MaintainPrefix(x,l,d[i],d[i+w]);
7          foreach s ∈ S_c and its interval [u,v] do
8              I_s ← I_s ∪ { d[u,v] };
9  y ← W(q,1);
10 l ← PrefixLength(y,τ), S_o ← GenSignature(y,τ,k_max);
11 A_1 ← ⋃_{s∈S_o} I_s;
12 for i = 1 to |q| do
13     (y,l,S_o,S_c) ← MaintainPrefix(y,l,q[i],q[i+w]);
14     A_{i+1} ← A_i;
15     if S_o ≠ ∅ or S_c ≠ ∅ then
16         A_{i+1} ← (A_{i+1} \ ⋃_{s∈S_c} I_s) ∪ ⋃_{s∈S_o} I_s;
17 for i = 1 to |q| do
18     MergeInterval(A_i);
19     foreach d[u,v] ∈ A_i do
20         T ← T ∪ VerifyInterval(W(q,i), W(d,u..v));
21 return T
```

4.3 Verification

Because of the overlap between adjacent windows, the verification of a query window against a candidate interval can be performed in a rolling fashion. To compute the intersection of a query window y and a data window x, we use two hash tables to count the multiplicities of their tokens, and the intersection $O(x,y) = \sum_{t \in y} \min(mul(t,x), mul(t,y))$. For the next data window x', since only the multiplicities of the outgoing and the incoming tokens change, we can compute $O(x',y)$ with four operations on the two hash tables, including a deletion, an insertion, and two lookups. Similarly, for the next query window y', we can obtain its hash table by two operations on the existing hash table of y instead of counting multiplicities by starting from scratch.

The above method spends w hash table operations for the first query window, 2 for any other query window, and $2w + 4(v-u)$ for a candidate interval $d[u,v]$. Based on this observation, candidate intervals can be further merged if they are close to each other. Consider two candidate intervals $d[u_1,v_1]$ and $d[u_2,v_2]$, where $u_2 > v_1$. If $u_2 - v_1 < \frac{w}{2}$, they will be merged into $d[u_1,v_2]$ and verified. This is because there are $4w + 4(v_2 + v_1 - u_2 - u_1)$ hash table operations to process the two separate intervals, and $2w + 4(v_2 - u_1)$ for a merged interval. The latter is less than the former when $u_2 - v_1 < \frac{w}{2}$, even if the windows between $W(d, v_1+1)$ and $W(d, u_2-1)$ are not candidates. We integrate this method to the interval merge step (Line 18) in Algorithm 4.

Another optimization is that we can early terminate the verification of an interval seeing an already computed intersection. Consider a query window x and a window $W(d,j)$ from an interval $d[u,v]$. If $w - O(x, W(d,j)) = \tau + \delta$, where $\delta > 0$, then none of the windows between $W(d,j)$ and $W(d, j+\delta-1)$ is a result, because they differ by at most $\delta - 1$ tokens from $W(d,j)$ and thus at least $\tau + 1$ tokens from x. To exploit this observation, we skip verifying the remaining windows of the interval if $j + \delta > v$.

5. COST ANALYSIS AND TOKEN UNIVERSE PARTITIONING

We analyze the cost of Algorithm 4 and based on the analysis we devise the token universe partitioning algorithm.

5.1 Cost Analysis

For a given query q, the query processing cost consists of the costs in three phases: signature generation, candidate generation, and verification, i.e.,

$$C_{query\_proc}(q) = C_{sig\_gen}(q) + C_{cand\_gen}(q) + C_{verify}(q).$$

For the signature generation phase, we ignore the prefix computation costs of the algorithms compared and consider the costs of generating signatures only. A signature is generated when its interval is opened or closed (including false open and close). We assume that the cost of generating a signature s is $c_{comb} \cdot |s|$, i.e., the cost of combining a token multiplied by the number of constituent tokens in s. Then the signature generation cost of Algorithm 4

$$C_{sig\_gen}(q) = c_{comb} \cdot 2 \sum_{s \in S_{all}} |s|, \tag{2}$$

where S_{all} denotes the multiset [4] of signatures generated in the signature generation phase. We compare with Algorithm 2 which processes each window individually. Let $s.u$ and $s.v$ denote the two ends of a window interval that contains s. Algorithm 2's signature generation cost is $c_{comb} \cdot \sum_{s \in S_{all}} (s.v - s.u + 1)|s|$. Algorithm 4 saves signature generation cost for the signatures such that $s.v - s.u > 1$. In the worst case, nothing is shared in the prefixes of adjacent windows, and the signature generation cost of Algorithm 4 is twice as much as Algorithm 2.

The candidate generation cost of Algorithm 4 consists of two parts: accessing inverted index and merging candidate intervals. Index is accessed when the interval of a signature s is opened or closed (true open and close only). Merging only occurs if two adjacent windows have different candidate intervals. Assuming the cost of accessing an interval is c_{int}, the candidate generation cost

$$C_{cand\_gen}(q) = c_{int} \cdot (2 \sum_{s \in S_{true}} |I_s| + \sum_{i=1}^{|q|} 1_{S_i \neq S_{i-1}} \cdot \sum_{s \in S_i} |I_s|). \tag{3}$$

S_{true} denotes the multiset of signatures generated by a true open in the signature generation phase. $|I_s|$ denotes the number of window intervals in the postings list of s. S_i denotes the signatures whose intervals are open when the window slides to $W(q,i)$. $1_{S_i \neq S_{i-1}}$ is the indicator function that returns 1 if $S_i \neq S_{i-1}$ or 0 otherwise (S_0 is defined as an $\emptyset$). For Algorithm 2, the candidate generation cost is $c_{int} \cdot \sum_{i=1}^{|q|} \sum_{s \in S_i} |I'_s|$, where I'_s denotes the number of windows in the postings list of s. Hence Algorithm 4 saves cost for the signatures that are shared by more than two consecutive windows in both the query and the data windows. In the worst case, the candidate generation cost of Algorithm 4 is three times as much as Algorithm 2.

For verification, the first query window spends w operations to count its token multiplicities, and every other query window spends 2 operations, hence $2|q| - w$ operations in total. To count the token multiplicities of a candidate interval $d[u,v]$, there are $2w + 4(v-u)$ operations (early termination not considered). Assuming the cost of a hash table operation is c_{hash}, the verification cost of Algorithm 4

$$C_{verify}(q) = c_{hash} \cdot (2|q| - w + \sum_{i=1}^{|q|} \sum_{d[u,v] \in A_i} 2w + 4(v-u)), \tag{4}$$

where A_i denotes the set of candidate intervals of $W(q,i)$ after merging. When $w > 1$, this cost is always less than the cost of Algorithm 2, which is $c_{hash} \cdot (w|q| + \sum_{i=1}^{|q|} \sum_{d[u,v] \in A_i} 2w(v - u + 1))$, even if in the worst case.

5.2 A Greedy Partitioning Algorithm

Since the token universe partitioning can be done offline and queries are processed online, our goal is to optimize the query processing for multiple queries (denoted by a query workload $\mathcal{Q}$) rather than a single query. Instead of resorting to a straightforward equi-width partitioning, we leverage the above cost model and can formulate the token universe partitioning as an optimization problem.

Given a query workload $\mathcal{Q}$, the processing cost of each query is summed up to the total processing cost of the workload:

$$C_{workload}(\mathcal{Q}) = \sum_{q=1}^{|\mathcal{Q}|} C_{query\_proc}(q).$$

PROBLEM 2. *Given a token universe $\mathcal{U}$, a collection of data documents $\mathcal{D}$, a query workload $\mathcal{Q}$, and parameters w, τ, and $k_{\max}$, determine the global order $\mathcal{O}$ and divide the token universe into $k_{\max}$ partitions such that $C_{workload}(\mathcal{Q})$ is minimized.*

When $k_{\max} = 1$, this problem becomes finding the optimal global order for standard prefix filtering. It is likely to be intractable and hence most existing prefix filtering-based algorithms sort by increasing document frequency as a heuristic [10, 4, 35, 33] . The problem is harder when $k_{\max} > 1$. Moreover, the computation of $C_{workload}(\mathcal{Q})$ for a partitioning $\mathcal{P}$ incurs considerable overhead because we need to build index for $\mathcal{D}$ with respect to $\mathcal{P}$ and then process the queries in $\mathcal{Q}$ to sum up the cost. Seeing these factors, we design an algorithm to find a good partitioning while bounding the number of times computing $C_{workload}(\mathcal{Q})$.

Our algorithm is based on a greedy and two-level blocking strategy. We first choose to sort in $\mathcal{U}$ by increasing order of window frequency. Then we divide $\mathcal{U}$ into 1-wise and 2-wise tokens, find a best border which yields the smallest $C_{workload}(\mathcal{Q})$, and then divide the partition of 2-wise tokens into 2-wise and 3-wise tokens. This is repeated until $k_{\max}$ is reached. For a border of i-wise and $(i+1)$-wise tokens, there can be at most $|\mathcal{U}| + 1$ possibilities. Since computing $C_{workload}(\mathcal{Q})$ for such number of times is prohibitive for large $|\mathcal{U}|$, we choose to divide $\mathcal{U}$ into blocks of size B_1, and pick the block boundary which yields the smallest $C_{workload}(\mathcal{Q})$ if we divide the partition there. Denoting this block boundary by b_i, then we divide two adjacent blocks, $[b_{i-1}, b_i]$ and $[b_i, b_{i+1}]$, into sub-blocks of size B_2, and pick the best sub-block boundary as the partition boundary. The computation of $C_{workload}(\mathcal{Q})$ is invoked no more than $(k_{\max} - 1)(\lceil \frac{|\mathcal{U}|}{B_1} \rceil + 2\lceil \frac{B_1}{B_2} \rceil - 1)$ times.

Note that empty partitions are allowed by the greedy algorithm, and hence it is not mandatory to have i-wise tokens

[4] A signature may be generated multiple times. It is inserted into S when its interval is opened, including a false open.

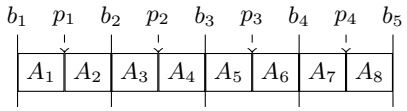


Figure 3: Greedy-and-Blocking Partitioning

for any $i \in [1, k_{\max}]$. We also argue that the reason why we choose to find a good partitioning by the cost model instead of simply setting a selectivity threshold is that combining any two tokens does not always make the selectivity approach the threshold due to the existence of correlations between tokens, e.g., "Kuala" and "Lumpur".

EXAMPLE 7. *Consider a universe consists of eight tokens, $k_{\max} = 3$, $B_1 = 2$, and $B_2 = 1$. The universe is divided into 4 blocks. Figure 3 shows the tokens and the block boundaries. We first divide the universe into 1-wise and 2-wise tokens. Suppose the costs of dividing at $b_1 \cdots b_5$ are 10, 8, 9, 10, 11, respectively. We pick the smallest one, b_2, and then divide $[b_1, b_2]$ and $[b_2, b_3]$ into sub-blocks of size B_2. Suppose the total costs of dividing at p_1 and p_2 are 9 and 7, respectively. p_2 is chosen to divide the universe, and thus $A_1 \cdots A_3$ are 1-wise tokens, and $A_4 \cdots A_8$ are 2-wise tokens. Then we proceed to divide the latter into 2-wise and 3-wise tokens. Suppose the total costs of dividing at p_2 and $b_3 \cdots b_5$ are 8, 8, 5, 7, respectively. We pick the smallest one, b_4, and then consider sub-block boundary p_3 and p_4. Suppose the costs of dividing at p_3 and p_4 are 4 and 6, respectively. p_3 is chosen to divide $A_4 \cdots A_8$. Finally, $A_1 \cdots A_3$ are 1-wise tokens, $A_4 \cdots A_5$ are 2-wise tokens, and $A_6 \cdots A_8$ are 3-wise tokens. The computation of $C_{workload}(\mathcal{Q})$ is invoked 13 times.*

In case that a historical query workload is not available, a portion of data documents can be sampled as a surrogate, denoted by $\mathcal{Q}'$. Its size is controlled by a sample ratio ρ; i.e., $|\mathcal{Q}'| = \rho \cdot |\mathcal{D}|$. We choose this option in our experiments.

6. COPING WITH LARGE THRESHOLDS

A large τ may cause large number of combinations in the signature generation. Assume in a window x, the $\tau+1$ coverage is equally distributed to the $k_{\max}$ partitions; i.e., each class in its prefix has a coverage of $\frac{\tau+1}{k_{\max}}$. For each class, there are $(\frac{\tau+1}{k_{\max}} + i - 1)$ tokens and thus $\binom{\frac{\tau+1}{k_{\max}}+i-1}{i}$ combinations generated. When $\tau = 99$, the total number of combinations generated from the prefix is 23,750 when $k_{\max} = 4$.

To scale up for large τ, recall in Section 3.2 we partition the token universe and tokens are only combined with others from the same partition. We further exploit this idea and divide each i-wise partition in the token universe into m equi-width sub-partitions. All these m sub-partitions use i-wise signatures, and combinations are only generated within each sub-partition. Since single tokens are used in 1-wise partition, we choose not to divide it into sub-partitions. To compute the prefix length, Lemmas 3 and 4 also apply for sub-partitions. Thus we modify Algorithm 1 by summing up the coverage in each sub-partition until it reaches $\tau + 1$.

EXAMPLE 8. *Consider the token universe in Figure 4. $k_{\max} = 3$, and the solid lines separate the three partitions. $\tau = 5$. Consider a window x. Before further partitioning, the prefix length is 4 + 5 = 9 and there are 6 + 10 = 16 combinations generated. When $m = 3$, 2-wise and 3-wise partitions are further divided into 3 sub-partitions, respectively, as shown by the dashed lines. The arrows show which classes and which sub-partitions its tokens belong to. By further partitioning, the prefix length is increased to 2 + 2 + 3 + 3 + 4 = 14, but the number of combinations is reduced to 1 + 1 + 1 + 1 + 4 = 8.*

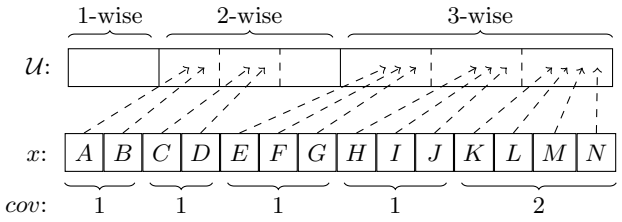


Figure 4: Example of Further Partitioning

Assume the above mentioned $\frac{\tau+1}{k_{\max}}$ coverage is equally distributed to the m sub-partitions. We need $(\frac{\tau+1}{m \cdot k_{\max}} + i - 1)$ tokens for each sub-partition of an i-wise partition, and thus in total $m(i-1)$ more tokens in each i-wise partition in the prefix. On the other hand, there are $m\binom{\frac{\tau+1}{m \cdot k_{\max}}+i-1}{i}$ combinations generated in each i-wise partition. In consequence, we trade prefix length for combination number. If we set m as $\alpha(\tau+1)$, the number of combinations will be $\alpha(\tau+1)\binom{\frac{1}{\alpha \cdot k_{\max}}+i-1}{i}$, hence proportional to $\tau + 1$ for a fixed i, as opposed to the exponential increase with τ before further partitioning. According to the experiments, $\alpha = 0.2$ gives best performance and we use this setting to compute token universe partitioning and process queries when τ is large.

To adapt Corollary 1 for sub-partitions, the upper bound of prefix length becomes $\tau + 1 + m\sum_1^{k_{\max}-1}$. For Corollary 2, in the highest class of a prefix, the last non-empty sub-partition of tokens has a coverage above zero.

7. EXPERIMENTS

7.1 Experiment Setup

The following algorithms are compared in the experiment.

- **Adapt** is a state-of-the-art algorithm for set similarity search and join [33]. It leverages extended prefix filtering and computes an appropriate prefix length for each query object using a cost model. To adapt the algorithm to local similarity search, we materialize the windows of data and query documents as its data and query objects, respectively.
- **Faerie** is a state-of-the-art algorithm for approximate dictionary-based entity extraction [13]. It finds approximate occurrences of the indexed entities in a query document. We materialize the windows of data documents as entities. The specific implementation we use considers only candidate windows of size w, and our overlap constraints are converted into corresponding equivalent Jaccard constraints.
- **FBW** is a Winnowing-family algorithm [31] which returns approximate answers to the problem of finding documents that share $w - q + 1$ consecutive token q-grams while tolerating $q\tau$ errors, where q is the q-gram length. We use its fingerprinting scheme to generate candidates and they are verified against our similarity constraint. q-gram length is set to 2 to balance the number of results and query processing time.
- **pkwise**, short for **p**artitioned k-wise, is our proposed algorithm equipped with interval sharing for adjacent windows. We hash signatures into 4-byte integers. The number of sub-partitions m is set to 1 unless otherwise specified.

Table 1: Dataset Statistics

Dataset	$\|\mathcal{D}\|$	$\|\mathcal{Q}\|$	avg. $\|d\|$	avg. $\|q\|$	$\|\mathcal{U}\|$
REUTERS	7,791	1,000	237.2	231.1	33,260
TREC	185,666	1,000	198.2	214.1	148,244
PAN	10,483	1,000	27,026.8	721.6	1,846,623

Other methods for similarity search and join, e.g., [3, 4, 21, 35], and approximate entity extraction, e.g., [9], are not compared since prior work [35, 33, 13] showed they are outperformed by the above selected methods. We do not consider the method in [2] developed for approximate entity extraction because it relies on WtEnum [3] which enumerates minimal subset of entities whose sum of weights is no less than the threshold. By materializing query windows as entities, the number of subsets per window is $\binom{w}{w-\tau}$, which is prohibitive for local similarity search, e.g., 5.4×10^{20} when $w = 100$ and $\tau = 20$.

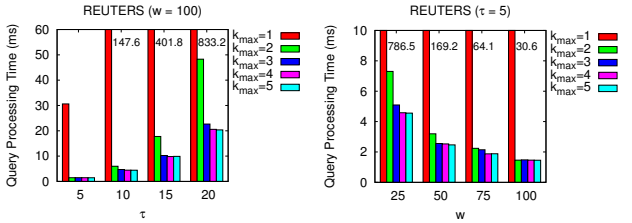
We select three publicly available datasets which were used in prior related studies:

- **REUTERS** is a set of 19K Reuters news stories [5]. We extract news body as documents.
- **TREC** is a set of references from MEDLINE. It is used for the TREC-9 Filtering Track Collections [6]. We extract the 233K paper abstracts as documents.
- **PAN** is used in the plagiarism detection task of PAN Workshop and Competition of 2010 (PAN-PC-10) [7]. It contains about 11K source documents and 16K suspicious documents that may contain plagiarism.

In order to make the numbers of documents in all the settings are the same, short documents with less than 100 tokens are removed from the corpora. For PAN, we use source documents as data documents and sample 1,000 queries from suspicious documents. Each query is composed of a number of paragraphs which contain true plagiarism. Non-English documents are removed. For REUTERS and TREC, we sample 1,000 documents as queries and take the rest as data documents. Table 1 shows statistics about the datasets.

Average query processing time is measured by varying τ while fixing w as 100, and by varying w while fixing τ as 5. For Adapt and Faerie, window materialization is not counted towards their index construction or query processing time.

The experiments were carried out on a PC with an Intel Xeon E5620 2.4GHz Processor and 96GB RAM, running Ubuntu 14.04.3. The algorithms were implemented in C++ and in a main memory fashion.



(a) REUTERS, vary τ (b) REUTERS, vary w

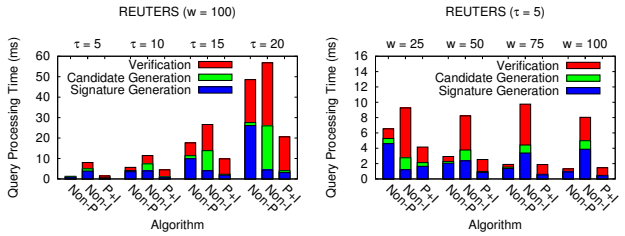
Figure 5: Effect of $k_{\max}$

We sample a subset of data documents as query workload $\mathcal{Q}'$ and partition token universe with the algorithm proposed in

[5]http://www.daviddlewis.com/resources/testcollections/reuters21578/

[6]http://trec.nist.gov/data/t9_filtering.html

[7]http://www.uni-weimar.de/en/media/chairs/webis/corpora/corpus-pan-pc-10/



(a) REUTERS, vary τ (b) REUTERS, vary w

Figure 6: Effect of Partition & Interval

Section 5.2. The default value of $k_{\max}$ is 4. The values of c_{comb}, c_{int}, c_{hash} in Equations 2 – 4 are 10, 2, and 1, respectively. $B_1 = 0.1|\mathcal{U}|$ and $B_2 = 0.01|\mathcal{U}|$. Then we process the queries Q (different from Q') with the obtained partitioning results. As for the sample ratio of query workload, the experiment results show its effect on the query processing performance is not obvious. E.g., the average query processing times are 4.64, 4.44, 4.41, 4.39, and 4.39 milliseconds, respectively, when the sample ratio changes from 0.5% to 2.5% on REUTERS when $w = 100$ and $\tau = 10$. We choose 1% as the sample ratio.

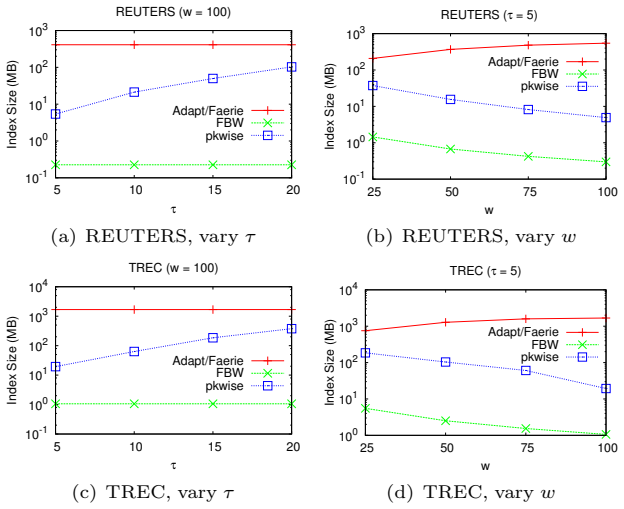
7.2 Effect of Partitioned k-wise Signatures

Figures 5(a) – 5(b) show the average query processing times with varying τ and w on REUTERS for $k_{\max} \in [1, 5]$. Note that the algorithm becomes standard prefix filtering when $k_{\max} = 1$. $k_{\max} = 1$ has the worst performance and is up to two orders of magnitude slower than the other $k_{\max}$ settings, especially for large τ or small w. The reason is that there are frequent tokens in its prefix and they cause large candidate numbers. When $w = 100$ and $\tau = 5$, a $k_{\max}$ of 2 or 3 is as good as 4 or 5. But when w decreases or τ increases, setting $k_{\max}$ as 4 or 5 runs 2 times faster than 2 or 3 because of looser similarity constraints which call for signatures with better selectivity. The results demonstrate that query performance can be improved by combining tokens. We set $k_{\max}$ as 4 in the rest of the experiments (on par with $k_{\max} = 5$ on query processing but faster on token universe partitioning).

We evaluate the effect of partition by comparing partitioned k-wise with non-partitioned k-wise, i.e., all signatures with a fixed number of tokens. The average query processing times with varying τ and w on REUTERS are shown in Figures 6(a) – 6(b). The running times are decomposed into three phases. The partitioned and the non-partitioned algorithms are denoted by P+I and Non-P, respectively. For partitioned k-wise, we set $k_{\max} = 4$. For non-partitioned k-wise, we choose $k = 3$ since it gives best performance for most w and τ settings. As seen from the figures, partitioned k-wise substantially saves signature generation time but spends more on verification time since 3-wise signatures are more selective than the mixture of 1 to 4-wise signatures. Considering the two effects, partitioned k-wise exhibits a reduction in overall query processing time in most cases. The speedup is more significant for larger τ or smaller w, and can be up to 2.4 times. We also observe an exception that non-partitioned k-wise spends slightly less time when $w = 100$ and $\tau = 5$. The reason is that both algorithms spend very short time on signature generation, and hence the gain of partitioned k-wise in this phase is small, resulting in an overall slower query processing.

7.3 Effect of Interval Sharing

We evaluate the effect of the interval sharing technique



(a) REUTERS, vary τ (b) REUTERS, vary w

(c) TREC, vary τ (d) TREC, vary w

Figure 7: Comparison with Alternatives - Index

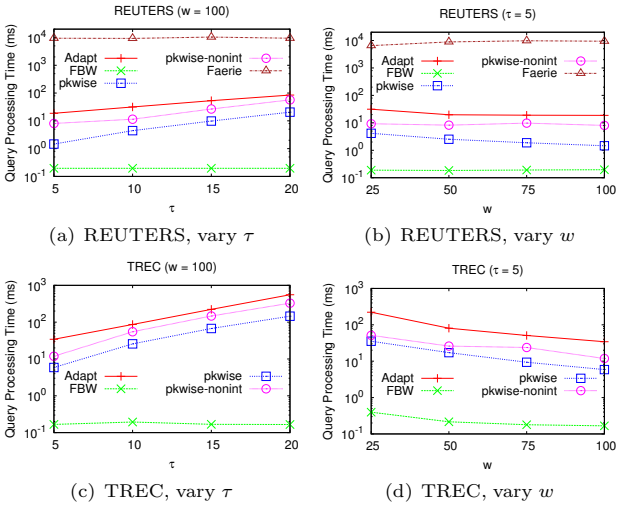
proposed in Section 4 and plot the average query processing times with varying τ and w on REUTERS in Figures 6(a) – 6(b). The algorithms with and without interval sharing are denoted by P+I and Non-I, respectively. By taking advantage of overlapping windows, the query processing time is reduced by 2.6 to 5.5 times by varying τ. By varying w, the speedup is 2.2 to 5.5 times, and different trends are observed for the algorithms with and without interval sharing. This is because for larger w and a fixed τ, the similarity constraint becomes tighter and thus candidate number decreases, but processing longer windows spends more time. The two factors result in the fluctuation of query processing time without interval sharing. When interval sharing is applied, processing continuous windows becomes faster and thus the effect of the first factor dominates. We also measure the average sharing by computing the Jaccard similarity between the prefixes of every two adjacent windows and taking the average. When $w = 100$ and τ grows from 5 to 20, the sharing in query windows slightly decreases from 0.966 to 0.963. When $\tau = 5$ and w grows from 25 to 100, the sharing in query windows increases from 0.872 to 0.966. A similar result is observed on the sharing in data windows. We also notice that when $w = 25$ and $\tau = 5$, the algorithm with interval sharing spends more time on signature generation. The reason is that window size is small and sharing is relatively low, and thus the gain from sharing does not counteract the overhead on maintaining prefixes and intervals. Nonetheless, candidate generation and verification still benefit from interval sharing, hence resulting in less overall query processing time in this setting. Another interesting result is that by indexing intervals instead of individual windows, index size is reduced by 3 to 14 times (e.g., from 77.4MB to 5.4MB when $w = 100$ and $\tau = 5$), and the reduction is more remarkable for larger w.

7.4 Comparison with Alternative Methods

Index sizes are compared first. Figures 7(a) – 7(d) show the index sizes of the algorithms with varying τ and w on REUTERS and TREC. Since Adapt and Faerie index all the tokens in each window, they have the same index sizes and they only vary with w. FBW's index size is significantly smaller than the exact algorithms due to its signature selection scheme. Among the exact algorithms, pkwise has the smallest index size, because (1) only the signatures generated from prefixes

Table 2: Index Construction Time (REUTERS)

w	τ	Adapt	Faerie	FBW	pkwise (part + index)
25	5	35s	28s	1.1s	303s + 10.3s
50	5	61s	47s	1.3s	172s + 4.9s
75	5	86s	66s	1.4s	147s + 3.7s
100	5	100s	72s	1.4s	133s + 2.9s
100	10	100s	72s	1.4s	377s + 4.7s
100	15	100s	72s	1.4s	859s + 9.5s
100	20	100s	72s	1.4s	2009s + 14.3s



(a) REUTERS, vary τ (b) REUTERS, vary w

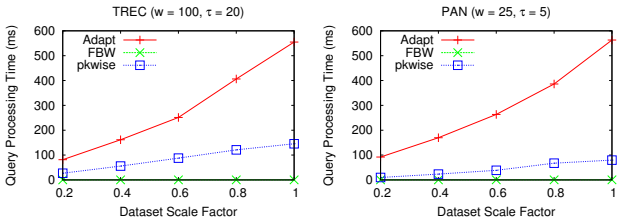
(c) TREC, vary τ (d) TREC, vary w

Figure 8: Comparison with Alternatives - Runtime

are indexed by pkwise, as opposed to Adapt and Faerie which index every token in every window, and (2) interval sharing further reduces pkwise's index size. The gap ranges from 3.5 to 76.1 times on REUTERS and 4.1 to 86.7 times on TREC.

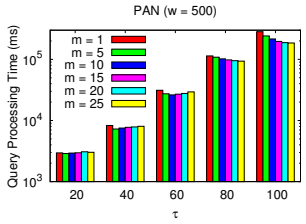
The corresponding index construction times on REUTERS are shown in Table 2. For pkwise we decompose the time into two parts: computing token universe partitioning and indexing data documents. We observe that the indexing times of Adapt and Faerie only change with w. Both parts of pkwise's index construction time increase with looser similarity constraint (smaller w or larger τ). Despite more time consumption on the computation of partitioning, pkwise spends less time indexing the data documents than the other two exact algorithms. We argue that the computation of partitioning can be done offline and the output can be used on data documents with approximately the same token frequency distribution.

Figures 8(a) – 8(d) show the average query processing times of the algorithms on REUTERS and TREC by varying τ and w. Since the other competitors are not equipped with interval sharing, for the purpose of fair comparison, we also show pkwise's performance when interval sharing is disabled (denoted by pkwise-nonint). Faerie is unable to finish processing the 1,000 queries on TREC in 24 hours, and thus its performance is not shown on TREC. On REUTERS, Faerie is the least competitive and two to three orders of magnitude slower than the other algorithms. Although its candidate number is very close to the result number, its filtering is very time-consuming due to the heap-based candidate generation. This result suggests that Faerie is not efficient for local similarity search where windows are much longer than normal entities (on average less than ten tokens). For Adapt and pkwise, both times increase with τ and decrease with w. pkwise is always faster than Adapt. The speedup is 4.1 to 12.8 times on REUTERS and 3.3 to 6.3 times



(a) TREC, vary $|\mathcal{D}|$ (b) PAN, vary $|\mathcal{D}|$

Figure 9: Scalability



(a) PAN, vary τ

Figure 10: Large Thresholds

on TREC. The speedup not only comes from interval sharing but also partitioned k-wise signatures, as it can be seen that pkwise-nonint also outperforms Adapt (by up to 4.3 times) in every setting. Although FBW is significantly faster than the exact methods, it returns only 10.1% to 42.7% results, and the percentage is low for small w.

We evaluate the scalability of the algorithms with varying dataset size. 20% to 100% data documents are sampled from TREC and PAN. The query processing times ($w = 100$ and $\tau = 20$ on TREC and $w = 25$ and $\tau = 5$ on PAN) are given in Figures 9(a) – 9(b). Faerie is not shown due to its much longer query processing time. The times of the algorithms grow approximate linearly with the dataset size. pkwise has a slower growth rate than Adapt (3.8 and 7.1 times faster on TREC and PAN, respectively).

7.5 Large Thresholds

Figure 10(a) shows the average query processing time of pkwise on PAN with varying large thresholds when $w = 500$. The comparison with Adapt and Faerie is excluded because loading and indexing the materialized windows exceeds the main memory, and they are slower than pkwise on a sampled subset of 100 data documents when $w = 500$ and $\tau = 100$ by 4.4 and 213.7 times, respectively. We set the number of sub-partitions m to 1, 5, 10, 15, 20, and 25. With larger m, the number of combinations in signature generation is reduced but prefix length grows and thus selectivity is compromised. This effect can be seen: in most cases, the time first drops with m and then rebounds. Another trend is that the best m increases with τ: 1, 5, 10, 25, and 25 when $\tau = 20$, 40, 60, 80, and 100, respectively. Based on the results, we suggest users choose $m = 1$ when $m \leq 20$ and $m = 0.25 \cdot \tau$ when $m > 20$.

8. RELATED WORK

Similarity search and join have been studied by many researchers due to its importance in many applications, including near-duplicate document detection. To answer (multi)set similarity queries, many existing methods adopted the prefix filtering framework, which was proposed by Chaudhuri *et al.* [10] and later improved by subsequent studies [4, 35, 33]. Other methods include merging postings lists [27, 17, 21] and partitioning data by pigeon-hole principle [3], etc. Approximate solutions were also investigated; e.g., MinHash [7] and LSH [16, 28]. Another body of work studied the processing of string similarity queries by regarding documents as strings; e.g., [22, 34, 26, 12, 36]. Edit distance is usually adopted to capture the string similarity. We refer readers to [19] for an experimental comparison of prevalent methods.

Document fingerprinting methods have been extensively studied, aiming at finding near-duplicate documents or reused contents. Most proposed solutions can be divided into two categories: overlapping methods and non-overlapping methods, namely, by selecting overlapping or non-overlapping text segments as fingerprints. Notable overlapping methods are 0 mod p [25], super-shingles [8], Winnowing [29], Hailstorm [18], etc. Non-overlapping methods include hash breaking[6], DCT fingerprinting [30], qSign [20], learning hash code [37], etc. There are also methods using other features; e.g., I-Match [11] uses medium document frequency tokens and SpotSigs [32] selects tokens around stopwords.

While our paper and the above related studies focus on dealing with unstructured data, there are also a few studies on detecting copies in structured data [15, 5, 14, 24].

Token combinations (token sets) have been used to solve approximate ad-hoc entity extraction [2], error-tolerant set containment [1], and similarity queries on multi-attribute data [23]. The idea of partition and enumeration is used for similarity join [3]. We briefly discuss our differences: (1) To ensure the correctness, in [2] and [1], every minimal subset of entities or queries with sum of weight no less than the threshold is covered by at least one token set selected by a cost model. [3] resorts to pigeon-hole principle. Our method extends prefix filtering to enumerate token combinations. (2) In [3], records are converted to vectors and divided into two-level equi-sized partitions, and partition combinations are enumerated on the second level. In our method, token universe is partitioned using a cost model and we combine tokens rather than partitions. (3) [23] uses standard prefix filtering, and tokens of different attributes are organized in a tree and checked one by one to find candidates. We use hash values of token combinations to find candidates for a single attribute problem. (4) We take advantage of overlap between adjacent windows by interval sharing, which is not covered by the other methods.

9. CONCLUSION

We study the problem of local similarity search which identifies documents that share a common sliding window with the query but differ by at most τ tokens. Our solution is based on two observations: (1) token combinations are more selective than single tokens, and (2) overlap exist between adjacent sliding windows. We partition the token universe and consider using different numbers of tokens in a combination. A practical algorithm is devised to compute a good partitioning of the token universe. The techniques to support large thresholds are developed. Extensive experimental evaluation on real datasets demonstrates the superior query processing performance of the proposed method to alternative solutions.

Acknowledgements. Pei Wang, Chuan Xiao, and Yoshiharu Ishikawa are supported by JSPS Kakenhi 25280039 and 16H01722. Jianbin Qin and Wei Wang are supported by D2DCRC Grants DC25002 and DC25003. We thank the authors of [33] and [13] for kindly providing their source codes.

10. REFERENCES

[1] P. Agrawal, A. Arasu, and R. Kaushik. On indexing error-tolerant set containment. In *SIGMOD Conference*, pages 927–938, 2010.

[2] S. Agrawal, K. Chakrabarti, S. Chaudhuri, and V. Ganti. Scalable ad-hoc entity extraction from text collections. *PVLDB*, 1(1):945–957, 2008.

[3] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[4] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[5] L. Blanco, V. Crescenzi, P. Merialdo, and P. Papotti. Probabilistic models to reconcile complex data from inaccurate data sources. In *CAiSE*, pages 83–97, 2010.

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

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

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

[9] K. Chakrabarti, S. Chaudhuri, V. Ganti, and D. Xin. An efficient filter for approximate membership checking. In *SIGMOD Conference*, pages 805–818, 2008.

[10] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, pages 5–16, 2006.

[11] A. Chowdhury, O. Frieder, D. A. Grossman, and M. C. McCabe. Collection statistics for fast duplicate document detection. *ACM Trans. Inf. Syst.*, 20(2):171–191, 2002.

[12] D. Deng, G. Li, and J. Feng. A pivotal prefix based filtering algorithm for string similarity search. In *SIGMOD Conference*, pages 673–684, 2014.

[13] D. Deng, G. Li, J. Feng, Y. Duan, and Z. Gong. A unified framework for approximate dictionary-based entity extraction. *VLDB J.*, 24(1):143–167, 2015.

[14] X. Dong, L. Berti-Equille, Y. Hu, and D. Srivastava. Global detection of complex copying relationships between sources. *PVLDB*, 3(1):1358–1369, 2010.

[15] X. L. Dong, L. Berti-Equille, and D. Srivastava. Truth discovery and copying detection in a dynamic world. *PVLDB*, 2(1):562–573, 2009.

[16] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[17] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, pages 267–276, 2008.

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

[19] Y. Jiang, G. Li, J. Feng, and W. Li. String similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.

[20] 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, 2009.

[21] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[22] G. Li, D. Deng, J. Wang, and J. Feng. Pass-Join: A partition-based method for similarity joins. *PVLDB*, 5(1):253–264, 2012.

[23] G. Li, J. He, D. Deng, and J. Li. Efficient similarity join and search on multi-attribute data. In *SIGMOD Conference*, pages 1137–1151, 2015.

[24] X. Li, X. L. Dong, K. B. Lyons, W. Meng, and D. Srivastava. Scaling up copy detection. In *ICDE*, pages 89–100, 2015.

[25] U. Manber. Finding similar files in a large file system. In *USENIX Winter*, pages 1–10, 1994.

[26] J. Qin, W. Wang, C. Xiao, Y. Lu, X. Lin, and H. Wang. Asymmetric signature schemes for efficient exact edit similarity query processing. *ACM Trans. Database Syst.*, 38(3):16, 2013.

[27] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, pages 743–754, 2004.

[28] V. Satuluri and S. Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. *PVLDB*, 5(5):430–441, 2012.

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

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

[31] Y. Sun, J. Qin, and W. Wang. Near duplicate text detection using frequency-biased signatures. In *WISE*, pages 277–291, 2013.

[32] M. Theobald, J. Siddharth, and A. Paepcke. Spotsigs: robust and efficient near duplicate detection in large web collections. In *SIGIR*, pages 563–570, 2008.

[33] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD Conference*, pages 85–96, 2012.

[34] W. Wang, J. Qin, C. Xiao, X. Lin, and H. T. Shen. Vchunkjoin: An efficient algorithm for edit similarity joins. *IEEE Trans. Knowl. Data Eng.*, 25(8):1916–1929, 2013.

[35] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *ACM Trans. Database Syst.*, 36(3):15, 2011.

[36] X. Yang, Y. Wang, B. Wang, and W. Wang. Local filtering: Improving the performance of approximate queries on string collections. In *SIGMOD Conference*, pages 377–392, 2015.

[37] Q. Zhang, Y. Wu, Z. Ding, and X. Huang. Learning hash codes for efficient content reuse detection. In *SIGIR*, pages 405–414, 2012.

APPENDIX

A. PSEUDO-CODE OF PREFIX MAINTENANCE ALGORITHM

The pseudo-code of the prefix maintenance algorithm for signature generation is shown in Algorithm 5.

B. PROOFS

The proof of Theorem 1 is given below.

PROOF. We compare $x[l_x]$ and $y[l_y]$, the last tokens in x's and y's prefixes. Assume $x[l_x] \leq y[l_y]$. If $S_x \cap S_y = \emptyset$, by Lemma 4, there must be at least $\tau + 1$ tokens in $x[1\,..\,l_x]$ but not in $y[1\,..\,l_y]$. Because $x[l_x] \leq y[l_y]$, these tokens are not in $y[l_y + 1\,..\,|y|]$ either. Therefore, there must be at least $\tau + 1$ tokens in x but not in y, hence contradicting $w - O(x, y) \leq \tau$. □

We give the proof of Corollary 1.

PROOF. $n_i - i + 1 \leq 0$ when $n_i < i$. By Lemma 3, $\forall i, n_i - i + 1 \leq cov(i)$. By Lemma 4 and Algorithm 1, $\sum_{i=1}^{k_{\max}} n_i - i + 1 \leq \sum_{i=1}^{k_{\max}} cov(i) = \tau + 1$. Therefore $\sum_{i=1}^{k_{\max}} n_i \leq \tau + 1 + \sum_{i=1}^{k_{\max}} (i - 1)$. The left side of the inequality is exactly the prefix length, and the right side equals to $\tau + 1 + \sum_{i=1}^{k_{\max}-1} i = \tau + 1 + \frac{k_{\max}(k_{\max}-1)}{2}$. □

We give the proof of Corollary 2.

PROOF. Assume that coverage of the tokens in class h is zero. Since the total coverage of the prefix is $\tau+1$, the tokens in class h can be removed from the prefix, and the total coverage is still $\tau + 1$. It contradicts that the prefix is shortest. □

C. EXTENSIONS TO WEIGHTED CASE

In the weighted case of local similarity search, each token t is assigned a weight $wt(t)$. Let $wt(x)$ denote the accumulated

Algorithm 5: MaintainPrefix (x, l, t_1, t_2)

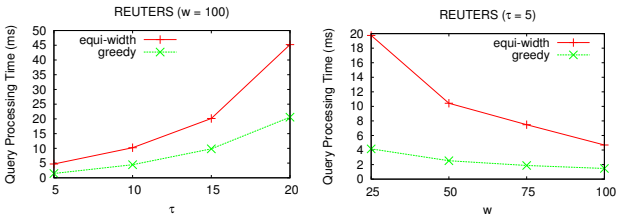
$P \leftarrow x[1\,..\,l]$, $S_o \leftarrow \emptyset$, $S_c \leftarrow \emptyset$;
if x is the last window **then**
  $S_c \leftarrow$ signatures generated from P;
  return $(\emptyset, 0, S_o, S_c)$
$x' \leftarrow (x \backslash t_1) \uplus t_2$, $P' \leftarrow P$;
if $t_1 \in P$ **then** $P' \leftarrow P' \backslash t_1$;
$l' \leftarrow |P'|$;
if $t_2 < x[l']$ **then** $P' \leftarrow P' \uplus \{t_2\}$, $l' \leftarrow l' + 1$;
if $cov(P \backslash t_1) < \tau + 1$ **then**
  if $cov(P') = \tau + 1$ **then**
    while $tail(P')$ are non-covering tokens **do**
      $P' \leftarrow P' \backslash tail(P')$;
    if $t_1 \neq t_2$ **then**
      $S_c \leftarrow$ signatures generated from P;
      $S_o \leftarrow$ signatures composed of t_2 and tokens in P';
  else
    $\Delta l \leftarrow \min\{\,\delta \mid cov(x'[l' + 1\,..\,l' + \delta]) = 1\,\}$;
    $\Delta P \leftarrow x'[l' + 1\,..\,l' + \Delta l]$;
    $P' \leftarrow P' \uplus \Delta P$;
    if $\Delta P \neq \{t_1\}$ **then**
      $S_c \leftarrow$ signatures generated from P and containing t_1;
      $S_o \leftarrow$ signatures composed of any token in ΔP and those in P';
else
  if $cov(P') > \tau + 1$ **then**
    $\Delta l \leftarrow \min\{\,\delta \mid cov(x'[l' - \delta\,..\,l']) = 1\,\}$;
    $\Delta P \leftarrow x'[l' - \Delta l\,..\,l']$;
    $P' \leftarrow P' \backslash \Delta P$;
    while $tail(P')$ are non-covering tokens **do**
      $P' \leftarrow P' \backslash tail(P')$;
    if $\Delta P \neq \{t_2\}$ **then**
      $S_c \leftarrow$ signatures generated from P and containing any token in ΔP;
      $S_o \leftarrow$ signatures composed of t_2 and tokens in P';
return (x', l', S_o, S_c)

weights of the tokens in x, i.e., $\sum_{t \in x} wt(t)$. Our goal is to find pair of windows such that the accumulated weights of their intersection is no less than a threshold; i.e., $\{\langle x, y\rangle \mid x \sqsubseteq d_i, d_i \in \mathcal{D}, y \sqsubseteq q, wt(O(x,y)) \geq \theta\}$. Given a multiset of tokens, we define its weighted coverage as the minimum accumulated weights of the errors required to affect all the signatures enumerated from these tokens. For n_i tokens in class i, since we need at least $n_i - i + 1$ errors to affect all the signatures, the weighted coverage is the sum of the $n_i - i + 1$ smallest weights among the n_i tokens. Lemma 4 also holds for weighted case. Hence to compute the prefix length of x, we use Algorithm 1 to sum up the weighted coverage of tokens until the value reaches $wt(x) - \theta + \epsilon$, where ϵ is a small positive real number. The prefix maintenance algorithm (Algorithm 5) is also modified by replacing $\tau + 1$ with $wt(x) - \theta + \epsilon$. We also modify the verification algorithm to compute the accumulated weights of intersection.

D. MORE EXPERIMENTS

D.1 Token Universe Partitioning

We evaluate the token universe partitioning by comparing the greedy partitioning algorithm proposed in Section 5.2 with the equi-width partitioning. For equi-width partitioning, we choose the $k_{\max}$ that yields the fastest query processing speed. The average query processing times with varying τ and w on



(a) REUTERS, vary τ (b) REUTERS, vary w

Figure 11: Token Universe Partitioning

Table 3: Precision/Recall on REUTERS and TREC

Algorithm	REUTERS (precision)	REUTERS (recall)	TREC (precision)	TREC (recall)
pkwise ($w = 25, \tau = 5$)	67.6%	86.1%	49.0%	85.7%
pkwise ($w = 50, \tau = 10$)	82.4%	53.5%	91.8%	57.1%
FBW ($w = 25, \tau = 5$)	81.5%	51.5%	75.5%	64.3%
FBW ($w = 50, \tau = 10$)	97.5%	10.0%	87.9%	28.6%

REUTERS are shown in Figures 11(a) – 11(b). The query processing with the greedy partitioning is always faster than the equi-width partitioning. The speedup varies from 2.0 to 4.7 times, and the gap is more significant when w is small.

D.2 Quality of Local Similarity Search

We evaluate the quality of local similarity search by running pkwise and FBW with two parameter settings: (1) $w = 25$ and $\tau = 5$, and (2) $w = 50$ and $\tau = 10$. Adapt and Faerie are also exact algorithms, and thus they have the same precision and recall as pkwise.

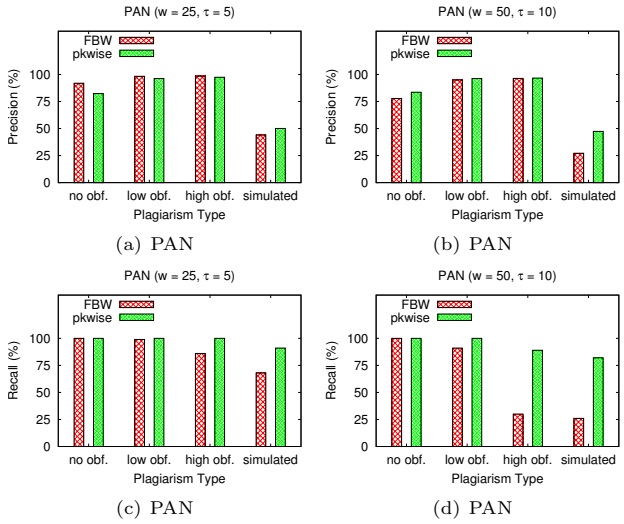
To label the ground truth (plagiarism or text reuse) in REUTERS and TREC, we first retrieve a set of candidate document pairs that share at least ten consecutive tokens. Afterwards the candidates pairs are manually checked by our volunteers. The ground truth in PAN has already been included in the dataset.

The ground truth pair is in the form of $\langle d[u,v], q[u',v']\rangle$, meaning that the text segment from the u'-th to the v'-th token of the query q is a plagiarism or reuse of the text segment from the u-th to the v-the token of a data document d. A ground truth pair $\langle d[u,v], q[u',v']\rangle$ is regarded as identified, if there exists a result pair of local similarity search $\langle W(d',i), W(q',j)\rangle$, such that $d = d'$, $q = q'$, $[i, i+w-1] \cap [u,v] \neq \emptyset$, and $[j, j+w-1] \cap [u',v'] \neq \emptyset$; i.e., the result pair overlaps the region of the ground truth pair in both the data and the query documents. The recall is defined as the percentage of identified ground truth pairs. To measure precision, we say a token $q[i]$ in the query is positive if it is covered by a result pair of local similarity search. It is a true positive if it is covered by an identified ground truth pair. The precision is defined as the ratio between the numbers of true positives and all positives, i.e., the percentage of correctly identified text length.

Table 3 shows the precision and recall on REUTERS and TREC. Using the setting $w = 25$ and $\tau = 5$ yields lower precision but much higher recall than $w = 50$ and $\tau = 10$. The recall of pkwise can be up to around 86% on both datasets. Although FBW has higher precision, its recall is rather low, missing at least half true results on REUTERS and one third on TREC.

For a fair comparison of precision on PAN, we use entire suspicious documents as queries (on average 16K tokens per query). There are four types of plagiarism in PAN: artificial plagiarism generated by a computer program with no, low, or high obfuscation, and simulated plagiarism purposefully made by a human. We plot the precisions of four plagiarism types with the two parameter settings in Figures 12(a) – 12(b), and the recalls in Figures 12(c) – 12(d). The following observations are observed:

- The precisions are similar when using two different parameter settings. Both algorithms exhibit high precision on artificial plagiarism. We notice that the precision on plagiarism without obfuscation is lower than that with obfuscation. The reason is that without obfuscation the plagiarism exactly matches original text, while the windows within τ tokens left or right to the plagiarized text are also identified due to the errors allowed in the parameter settings. On simulated plagiarism, pkwise's precision is up to 50%, and it is better than FBW.
- Using the setting $w = 25$ and $\tau = 5$ achieves higher recall, especially for simulated plagiarism. The recall can be 100% for all types of artificial plagiarism and 91% for simulated plagiarism. pkwise exhibits higher recall than FBW (by up to 59%), especially for simulated and highly obfuscated artificial plagiarism. To see why FBW does not perform well for these two types of plagiarism, we notice that in these two types of plagiarism, there are uncommon wording and grammatical errors (e.g., "had make") whose frequencies are zero in the data documents. Since FBW selects least frequent q-grams as signatures, the q-grams containing these errors are chosen, and this will make the plagiarism missed by FBW.



(a) PAN (b) PAN

(c) PAN (d) PAN

Figure 12: Precision/Recall on PAN

We note that the precisions (but not recalls) of the pkwise algorithm can be further enhanced by post-processing methods, e.g., machine learning-based and natural language processing-based techniques. In consequence, we suggest users choose $w = 25$ and $\tau = 5$.