FrepJoin: An Efficient Partition-based Algorithm for Edit Similarity Join

Paper id: 106

Abstract

String similarity join has become an essential operator in many applications where near-duplicate objects need to be found. This paper focuses on string similarity join with edit distance constraints. The existing prevalent algorithms for this problem usually adopt the *filter-and-refine* framework. However, these algorithms (a) are string-pair based, *i.e.,* they can't catch the dissimilarity between string subsets; (b) do not exploit fully the global information of strings, *e.g.,* the frequencies of characters and other statistics. Highlighted by these points, this paper investigates to develop a partition-based edit similarity join algorithm by utilizing the statistics to target the two time-consuming factors, *i.e.,* the enumeration of candidate pairs and the computation of edit distances. Firstly, the (combined) frequency vectors are used to partition datasets into data chunks with dissimilarity between them being able to be caught easily. Secondly, a novel partition-based algorithm is developed to generate candidate pairs efficiently by using the dissimilarity between data chunks to prune away a large proportion of candidate pairs directly, *i.e.,* without paying the price to enumerate them. Thirdly, a new filter is proposed to leverage the frequencies of characters to avoid computing edit distances for a noticeable proportion of candidate pairs which survive the existing filters. As a result, our algorithm outperforms alternative methods significantly and consistently, which is verified by extensive experiments on real datasets.

1. Introduction

The *string similarity join* has been recognized as an essential operator in a wide range of applications, *e.g.,* coalition detection [20], fuzzy keyword matching [15], data integration [8], data cleaning [6], near duplicate object detection [33], among many others. It has been applied to solve variety kinds of practical problems in industrial community. For example, Google uses it to detect near duplicate web pages [13] and Microsoft adopts it in the Data Debugger project [5]. With this comes the need to study effective and efficient algorithms for this operator, and many string similarity algorithms [2, 3, 10, 24, 32, 33] have been proposed.

Given two sets of strings, a similarity join between them aims to find out all pairs of similar strings from each of the sets according to a predefined similarity function such as Jacard distance [24, 34], Cosine distance [3], edit distance [32, 33], and their variants [4, 12, 30]. Among them, the *edit-distance* measures the similarity of two strings by the minimum number of edit operations (*i.e.,* insertion, deletion, and substitution of single characters) to transform one string to the other. Two distinctive properties of edit distance, *i.e.,* the ability to reflect the original ordering of tokens in strings and the ability to allow non-trivial alignment, make it a popular and important similarity function in many applications [33]. This paper focuses on the study of *string similarity join with edit distance constraint* (also referred to as *Edit Similarity Join*), *i.e.,* given two sets of strings, the edit similarity join returns all pairs of strings from each set such that the edit distance between them is upper bounded by a given threshold.

Edit similarity join is costly, which stems from two time-consuming factors. One is the computation of edit distances, and the other is the enumeration of all possible string pairs for which edit distance computation is needed. In fact, a standard dynamic programming scheme to compute edit distance leads to $O(n^2)$ time complexity [29], where n is the length of strings. And, a naive algorithm for edit similarity join enumerates all $O(N^2)$ string pairs, thus bears a complexity of $O(n^2N^2)$ [33]. This is prohibitive when string sets are of a huge size N.

In view of such a high cost, the past approaches [2, 3, 10, 24, 32, 33] mainly adopt a *filter-and-refine* framework to improve the efficiency of edit similarity join. In the filter stage, they generate signatures for each string and use various kinds of filters onto the signatures of strings to prune away string pairs which consist of dissimilar strings. In the refine stage, they verify the remaining string pairs via edit distance computation and output the final results.

However, there are some drawbacks in these approaches. Firstly, they generate candidate pairs inherently by enumerating string pairs and can not capture the dissimilarity between string subsets. Secondly, the signatures of each string are mainly some local structures such as (positional) q-grams and prefixes. This leads to the fact they can't catch the dissimilarity of strings from a global perspective of view. Thirdly, as a result, they usually generate a huge amount of candidate pairs. In fact, empirical evidence on several real datasets shows that their candidate size grows at a fast quadratic rate with the size of the dataset [33].

To address the issues mentioned above, in this paper we propose to take the (combined) frequencies of single characters, as well as other statistics, as global information of strings to target the two time-consuming factors in edit similarity join. Specifically, a novel *partition-based* algorithm is developed to utilize such information to enumerate smaller candidate set in a more efficient way by partitioning dataset into small chunks. And a new filter is proposed to make use of such information to further reduce the size of candidate set with low complexity. Unlike all the existing *filter-and-refine* approaches which have to access some q-grams before discarding a string pair, our method can directly prune away a large part of string pairs without accessing any q-grams. The following example illustrates our basic idea.

Example 1: Consider edit similarity self-join over string set $\mathcal{R}$ (shown in Fig. 1(a)) with edit distance threshold $\tau = 1$.

(1) The english alphabet can be partitioned into three subsets $\Sigma_1 = \{a,g,j,l,m,p,v,y,z\}$, $\Sigma_2 = \{b,d,h,i,n,q,s,u,x\}$ and

	string	chunk
s_1:	Steve Wilson	C_1
s_2:	Enrico Macii	
s_3:	Peter Bunemen	
s_4:	Peter Ponelli	C_2
s_5:	Takeo Kanade	
s_6:	Kate Michael	
s_7:	Karl Kurbel	
s_8:	Bart Preneel	
s_9:	Vipin Kumar	C_3
s_{10}:	David Maier	
s_{11}:	David Maner	
s_{12}:	Danny Dolev	
s_{13}:	Hanan Samet	
s_{14}:	Marianne Winslett	C_4
s_{15}:	Ernesto Damiani	
s_{16}:	Vladik Kreinovich	
s_{17}:	Daniel Thalmann	C_5
s_{18}:	David Sammon	

(a) data partition

chunk pair	distance
C_1, C_2	2
C_1, C_3	4
C_1, C_4	2
C_1, C_5	7
C_2, C_3	4
C_2, C_4	3
C_2, C_5	7
C_3, C_4	3
C_3, C_5	2
C_4, C_5	4

(b) Chunk distance

Figure 1: Motivation Example

$\Sigma_3 = \{c,e,f,k,o,r,t,w\}$. The combined frequency vector of string s refers to the vector $(f_1^{(\Sigma_1)}(s), f_2^{(\Sigma_2)}(s), f_3^{(\Sigma_3)}(s))$, where $f_i^{(\Sigma_i)}(s)$ counts the total times of appearances of characters of Σ_i in s. For example, the combined frequency vector of s_{10} and s_{18} are $(4, 4, 2)$ and $(5, 5, 1)$, respectively.

(2) The L_1-distance of combined frequency vectors being larger than 2τ implies the dissimilarity between strings, which will be proved in Lemma 5 later. For example, since the L_1-distance between $(4, 4, 2)$ and $(5, 5, 1)$ is 3 ($> 2\tau$), s_{10} is not similar to s_{18}.

(3) Via the combined frequency vectors, $\mathcal{R}$ can be partitioned into five chunks $C_1 \sim C_5$. Chunk distances are estimated lower bounds of L_1-distances between strings from different chunks. Data partition and chunk distance computation will be clarified in Section 4. Fig. 1(b) lists chunk distances between different chunks.

(4) Since many chunk distances are larger than 2τ, a remarkable proportion of string pairs can be pruned away without enumerating them, although strings in some pairs may share a common prefix or several q-grams. For example, strings in C_3 can be pruned away for each string in C_5, although s_{10}, s_{18} (and s_{11}, s_{18}) share a common prefix "David". Similarly, C_3,C_4,C_5 can be pruned directly for C_2, and so on. □

As suggested by the example, some statistics of the strings are useful to prune away candidate pairs and to capture the dissimilarity between string subsets by partitioning the string set into data chunks with distances between them estimated. As a result, a large part of candidate pairs can be pruned away without paying the price to enumerate them.

Technically, we obtain a new filter, Frequency Filter, to reduce the size of candidate set by analyzing the (combined) frequency vectors of strings. We also show that the new filter is complementary to the existing ones. In fact, the new filter can further prune away 30%-60% of candidate pairs which survive the existing filters. Further, a data partitioning procedure is developed to partition the input dataset into data chunks, by utilizing the statistics of the string set, such that more than 50% of chunk pairs can be directly pruned during similarity join. Two character selection strategies are also proposed to make the data partition procedure suitable for different string sets. Moreover, a novel *partition-based* edit similarity join algorithm is proposed to enumerate smaller candidate set in a more efficient way, compared with the past approaches. In fact, only 75%-90% of candidate pairs (generated by past approaches) are enumerated without effecting the correctness of edit similarity join. Putting together, the performance of edit similarity join algorithm can be improved more than 50%, as shown by our experimental results.

While some statistics have been adopted before (*e.g.,* string length being used in *Length Filter*, tokens' *idf* being utilized to void indexing rare tokens [3, 33] or to identify exact duplicates [7]), these methods are not intended to partition data and can not provide edit distance guarantee.

Although several edit similarity join algorithms [2, 24, 28, 31, 32] have also adopted the idea of data partition, our data partitioning procedure as well as the partition-based algorithm are totally new.

Contributions. This work makes a first effort to leverage the statistics of strings to partition string set into data chunks such that lower bounds of edit distances between data chunks are guaranteed. We provide different partitioning strategies, and develop a novel partition-based edit similarity join algorithm to improve the performance of edit similarity join significantly and consistently.

(1) We adopt a new filter, Frequency Filter, to improve the filtering effectiveness of filter stage. This filter prunes away candidate pairs by analyzing the frequency vectors of strings. Although, it is a special form of content-based filter [33], it is not adopted explicitly before and is complementary to the existing ones.

(2)We establish a general framework to partition dataset into small chunks with the lower bound of edit distance between strings from different data chunks being able to estimated. Moreover, we develop two different strategies to select characters for data partitioning with the expectation of the pruned data-chunk pairs being guaranteed.

(3) We propose a partition-based edit similarity join algorithm to improve the efficiency of the *filter-and-refine* framework by pruning a remarkable proportion of candidate pairs without paying the price to enumerate them. A two-level index structure is adopted to accelerate this procedure.

(4) We conduct an extensive experimental study to verify the efficiency of our partition-based edit similarity join algorithm. Using four real dataset (*i.e.,* AOL Query Log data, Author data, DBLP data and TREC data), we show that our partition-based algorithm outperforms consistently the existing algorithms on both short strings and long strings, with a cost of index size increased a little bit. we contend that our partition-based techniques yield a promising method for edit similarity join.

Organization. The rest of the paper is organized as follows: Section 2 introduces preliminaries and backgrounds. Sections 3 presents the new filter and discusses its independence. Section 4 presents our data partition techniques. Then, Section 5 develops the partition-based edit similarity join algorithm. Experimental results and analysis are given in Section 6, and Section 7 concludes the paper.

Related Work. Recent studies on edit similarity join, *e.g.,*PartEnum [2], AllPairs [3], SSJoin [6], Ed-Join [33], usually employ a q-gram based filter-and-refine framework which is also adopted in this paper. Algorithms based on this framework usually use q-grams to filter out most pairs of

dissimilar strings and further verify the remaining pairs via edit distance computation. Several filters have been developed, *e.g.,counting filter* [10, 6], *length filter* [10], *positional filter* [10, 34], *prefix filter* [2, 6, 34], *suffix filter* [34], *content-based mismatching filter* [33]. Although our *frequency filter* is a special form of content-based filter, it is not adopted explicitly before and can prune away with low complexity many candidate pairs which pass through the exiting ones.

Partition-based techniques are not new for string similarity join. PartEnum [2] generates signatures for each string by projecting its feature vector to randomly two-level partitioned q-gram sets. It needs to generate many candidate pairs to guarantee completeness. Suffix filter [34] adopts a *divide-and-conquer* framework, which works well for set-similarity join. Moreover, both [2] and [34] are still string-pair based and can not catch the dissimilarity between string subsets. Trie-Join [32] took a trie-structure as index to group together strings with same prefixes and its *subtrie pruning* rule was able to prune away whole groups of strings. As reported in [32], it works well only for short strings. In fact, it underperforms Ed-Join [33] (and our FrepJoin) when the average string length exceeds 30. Further, it spends a huge amount of memory when the size of join result is large. Sarawagi and Kirpal [24] proposed to reduce the index size by partitioning data into clusters of partially overlapping strings. [28] uses token (groups) to generate data partition for each computing node of mapreduce. Unlike our method, it allots each string into several data parts. Moreover, both [24] and [28] did not use global information of strings and couldn't provide lower bounds of edit distances. It is also a popular technique to partition strings into a group of substrings and transform approximate string matching into exact search [9, 21, 31]. Besides, it is proposed in [18] to partition long inverted lists into shorter ones and to skip irrelevant ones while processing approximate string matching. Although our method also leads to short inverted lists, it intends to partition the string setinto small chunks with performance guaranteed. The data partition techniques in these work are all orthogonal to that used in this paper.

Data partition techniques have also been applied extensively to deal with other tasks, *e.g.,* similarity join in high dimensional space [26], spatial-join [22], database design [1], to name a few of them. However, these tasks are totally different from edit similarity join, and all these methods are unsuitable for our purpose.

There are also other studies on string similarity join [12, 14, 17, 19, 27], approximate string similarity join [10], and (approximate) string matching [4, 11, 15, 16, 23, 30, 35]. Due to the large amount of related literatures, we only give an incomplete list here.

2. Preliminaries

Let $\Sigma = \{\alpha_1, ..., \alpha_m\}$ be a finite alphabet of size m. Each α_i is a character of Σ. A string s is an ordered array of characters drawn from Σ. Each string s is assigned an identifier $s.id$. The length of string s is denoted as $|s|$.

Definition 1: The *edit distance* between string x and y, denoted as $ed(x, y)$, is the minimum number of edit operations (insertion, deletion, and substitution of single characters) to transform x to y (and vice versa). □

$ed(x, y)$ can be computed via standard dynamic programming scheme in $O(n^2)$ time and $O(n)$ space [29].

Definition 2: Given two string sets $\mathcal{R}$ and $\mathcal{S}$, an *edit similarity join with edit distance threshold* τ returns pairs of strings from each set, such that their edit distance is no larger than τ, i.e., $\{(r, s)|ed(r, s) \leq \tau, r \in \mathcal{R}, s \in \mathcal{S}\}$. □

For the ease of exposition, we will focus on the self-join case in the paper, i.e., $\mathcal{R} = \mathcal{S}$.

Let s be a string of length n. We use $s[i : j]$, $1 \leq i \leq j \leq n$, to denote a *substring* of s of length $j - i + 1$ starting at position i. Interval $[i, j]$ is the *probing window* of substring $s[i : j]$. Substring $s[1 : i]$ and $s[i : n]$ $(1 \leq i \leq n)$ are referred to as the *i-prefix* of s and the *i-suffix* of s, respectively.

For a fixed positive integer q, each substring $s[i : i+q-1]$ $(1 \leq i \leq |s| - q + 1)$ of string s is often called as a *q-gram* of s with starting position i. A q-gram *token* together with its starting position *pos*, is called as a positional q-gram and usually represented in the form of $(token, pos)$ [10]. For simplicity, when there is no ambiguity, "positional q-gram" and "q-gram" are used interchangeably. A string s has $l = |s| - q + 1$ q-grams. All the positional q-grams of string s can be extracted to store in an array, which is referred to as the q-gram array of string s.

The *document frequency* of a q-gram w, denoted as $df(w)$, is the number of strings containing w. The *inverse document frequency* of w, denoted as $idf(w)$, is defined as $1/df(w)$. Intuitively, q-grams with high *idf* values are rare q-grams in the collection. [3, 33] use it to avoid indexing rare q-grams.

Each q-gram array can be sorted in decreasing order of their *idf* values and increasing order of their locations. As pointed out in [6], sorting q-gram arrays in this order is a good heuristic to speeding up similarity joins. In what follows, q-gram array means sorted q-gram array. Given a q-gram array x, $str(x)$ denotes its corresponding string. The i-th positional q-gram in x is denoted as $x[i]$; its q-gram and location are denoted as $x[i].token$ and $x[i].loc$, respectively.

An inverted index for q-grams is a data structure that maps each q-gram w to an array I_w containing entries in the form of (id, loc), where id identifies the string that contains w and loc is the starting location of w in the string identified by id. The entries in I_w are usually sorted in the increasing order of id and loc.

Besides these familiar concepts, a character's frequency in a string, as well as the frequency vectors of strings, are essential for our method.

Definition 3: Given $\alpha_i \in \Sigma$ $(1 \leq i \leq m)$ and string s, the *frequency* of character α_i in s, denoted as $f_i(s)$, is the times of α_i's appearances in s. The *frequency vector* of s, denoted as $f(s)$, refers to the vector $< f_1(s), \cdots, f_m(s) >$. □

The frequencies of each single character in all strings of $\mathcal{R}$ usually span an interval and reflect the features of $\mathcal{R}$. Our data partitioning will be implemented by splitting such intervals into small ones.

Definition 4: Given string set $\mathcal{R}$ and $\alpha_j \in \Sigma$, α_j's *frequency range associated with* $\mathcal{R}$ (or simply *frequency range*) is defined as the integer interval $[m_j, M_j]$, where $m_j = \min_{s \in \mathcal{R}} f_j(s)$ and $M_j = \max_{s \in \mathcal{R}} f_j(s)$. □

The L_1-distance, which is defined below, provides a tool to estimate the lower bound of edit distance between strings.

Definition 5: Let $u = (u_1, \cdots, u_m)$ and $v = (v_1, \cdots, v_m)$ be two real vectors. The *L_1-distance* between u and v, denoted as $\|u - v\|_{L_1}$, is defined as $\Sigma_{i=1}^{m} |u_i - v_i|$. □

3. Frequency Filtering and Its Independence

The results of *frequency analysis* indicate that the frequency of characters tend to vary by the subjects of strings, the purposes, the angles, and the styles of writers. It is widely used in cryptography and plays an fundamental role. We point out that frequency of characters in strings catches the similarity of strings and can be used to help edit similarity join, as shown in the following lemma.

Lemma 1: *For two strings x and y, $\|f(x) - f(y)\|_{L_1} > 2\tau$ implies $ed(x, y) > \tau$.* □

Lemma 1 states that the usefulness of the frequencies of characters in edit similarity join consists in the ability to reduce string pairs which need edit distance computed. The correctness of Lemma 1 has been observed before [33] and has been used to help edit similarity join (to be discussed soon later). However, to our best knowledge, no one adopts it as an independent filter explicitly.

We propose to take Lemma 1 as a new filter and name it as Frequency Filter, whose implementation is given in Algorithm 1. For string pair (x, y), It first invokes procedure FreStatus() to obtain the frequency vectors $f(x)$, $f(y)$ (Line 1). Then, it computes $\|f(x) - f(y)\|_{L_1}$ according to the definition of L_1-distance (Line 2-Line 4). If $\|f(x) - f(y)\|_{L_1} > 2\tau$, it returns *false* to indicate $ed(x, y) > \tau$. Else, it returns *true*.

The time complexity of Frequency Filter is $O(n+|\Sigma|)$, since obtaining the frequency vectors costs $O(n)$ and L_1-distance computation costs $O(|\Sigma|)$. If the frequency vector of each string is pre-computed, just like q-grams are extracted before edit similarity join, its time complexity becomes $O(|\Sigma|)$, which is irrelevant to the length of strings and can be viewed as a constant.

Algorithm 1: Frequency Filter (x, y)

Input : two strings x and y
Output: whether $\|f(x) - f(y)\|_{L_1} \leq 2\tau$ or not.

1 $f(x) \leftarrow$ FreStatus(x); $f(y) \leftarrow$ FreStatus(y);
2 $freDiffer \leftarrow 0$;
3 **for** $i = 1$ *to* m **do**
4 | $freDiffer = freDiffer + |f_i(x) - f_i(y)|$;
5 return $(freDiffer \leq 2 \cdot \tau)$;

Independence of the Frequency Filtering. Since computing edit distance is time-consuming in edit similarity join, several filters have been proposed to identify string pairs (s, t) with $ed(s, t) \leq \tau$. We show that Frequency Filter is independent of these existing filters by using the setting of the following example.

Example 2: Assume $\tau = 5$ and $q = 4$. Consider the following string pair (s_0, t_0). [1]

s_0="Petra Perner Case based reasoning for image interpretation"
t_0="Petra Perner Cbr based ultra sonic image interpretation"

With a little patience to check $\|f(s_0) - f(t_0)\|_{L_1} = 13 > 2\tau$, one can find out s_0 is not similar to t_0 according to Frequency Filter. □

Length Filtering mandates that $||s| - |t|| \leq \tau$.

[1]The strings are generated with the aid of an algorithm, since it is hard to build an example with either short strings or long strings.

Count Filtering mandates that s and t must share at least $LB_{s;t} = (\max(|s|, |t|) - q + 1) - q \cdot \tau$ common q-grams.

A positional q-gram $(token, loc)$ of string s are matched by a positional q-gram $(token', loc')$ of string t, if $token = token'$ and $|loc - loc'| \leq \tau$.

Position Filtering mandates that s and t must share at least $LB_{s;t}$ matching positional q-grams.

Prefix Filtering mandates that the $(q \cdot \tau + 1)$- prefix of s and the $(q \cdot \tau + 1)$-prefix of t must have at least one matching q-gram.

Example 3: Consider the strings s_0,t_0 in Example 2. Since $|s_0| - |t_0| = 3$, s_0, t_0 survives the length filtering. Further, Since s_0 and t_0 have a common prefix ($s_0[1:14] = t_0[1:14]$) and a common suffix ($s_0[38:58] = t_0[35:55]$), they share at least 11 4-grams in their common prefix and 18 4-grams in their common suffix (29 in total). Moreover, each of these 4-grams is a matching 4-gram and $LB_{s_0;t_0} = (58 - 4 + 1) - 4 \cdot 5 = 25$. Thus, s_0, t_0 also survive the count filtering, and positional filtering, the prefix filtering. □

Content-based Mismatching Filtering mandates that $\|f(s[i:j]) - f(t[i:j])\|_{L_1} \leq 2\tau$ holds for any probing window $[i, j]$, where $1 \leq i, j \leq \min(|s|, |t|)$.

Obviously, the content-based mismatching filter also comes from Lemma 1 and the Frequency Filter is a special form of it (to set the probing window to be $[1, |s|]$). Although [33] has observed its correctness, it was not adopted as an independent filter there. Instead, it is used to catch the local edit errors between strings, as shown below.

A q-gram $(token, loc)$ of string s is a mismatching q-gram from s to t, if it is not matched by any q-gram of t. The mismatching q-gram array Q from s to t contains all mismatching q-grams from s to t, sorted in increasing order of positions. For each q-gram $Q[i]$ and minimum j such that $j > i$ and $Q[j].loc - Q[j-1].loc > 1$, a probing window $[Q[i].loc, Q[j-1].loc+q-1] = [l, u]$ is specified. On each such probing window, the following condition is observed [33].

$$\|f(s[l:u]) - f(t[l:u])\|_{L_1} + \mathsf{SumRightErrs}(u+1) \leq 2\tau \quad (1)$$

where SumRightErrs($u + 1$) is the maximum number of non-overlapping mismatching q-grams from suffix $s[u+1:|s|]$ to suffix $t[u+1:|t|]$.

Content Filtering applies Condition (1) on each probing windows specified by the mismatching q-gram array Q from string s to string t.

Example 4: Consider the strings s_0,t_0 in Example 2. The mismatching q-gram array Q from s_0 to t_0 follows below. We refer readers to [33] for the computation of Q.

Q: | r␣Ca,12 | ␣Cas,13 | Case,14 | ase␣,15 | se␣b,16 | e␣ba,17 | ed␣r,22 | $\cdots$

Consider the first probing window $[12, 20]$ specified by q-grams (r␣Ca,12) and (e␣ba,17). Note that, $\|f(s_0[12:20]) - f(t_0[12:20])\|_{L_1} = 4$ and SumRighErrs(21) $= 5$ (see [33] for details of computation). Thus, Condition (1) is satisfied. Similarly, (s_0, t_0) can not be pruned away by any probing window specified by other 4-grams in Q. Finally, (s_0, t_0) survives the content filtering.

Together with Example 2 and Example 3, we find that s_0, t_0 are two dissimilar strings passing through all existing filters but Frequency Filter. □

As our experimental results show (see Section 6), a noticeable proportion of candidate pairs passing through the existing filters can be further pruned away by Frequency Filter, and vice visa.

Remarks. (1) The limited filtering effectiveness of the existing filters stems from the fact that they utilize only the local information of *positional* q-grams and ignore the global information of strings provided by statistics such as frequency vectors. More statistics will be used to avoid enumerating string pairs in next two sections.

(2) One question arises as to where to put Frequency Filter in Ed-Join, which integrates all existing filters and shorten the prefix length of prefix filtering. It is the most efficient existing edit similarity join algorithm (except Trie-Join for short strings). Being aware of the low cost, we argue that Frequency Filter should be put before counting filtering, content filtering and after length filtering, prefix filtering. Algorithm 2, with Line 15 and Line 16 ignored at present, is the Ed-Join algorithm with Frequency Filter added in Line 9. We refer readers to [33] for details of procedures Verify() and CalcPrefixLen(), which implements other existing filters and calculates the prefix length respectively.

Algorithm 2: Ed-Join (R_i, τ)

Input : String set R_i, edit distance upper bound τ
Output: Pair set S_i of strings in R_i with $ed(\cdot,\cdot) \leq \tau$.

```
   T ← ∅;
   foreach x ∈ R_i do
       A ← empty map from id to boolean;
       px ← CalcPrefixLen(x);
5      x.px ← px            /* store px for later use */
       for j = 1 to px do
           w ← x[j].token,   loc_x ← x[j].loc;
           foreach (y, loc_y) ∈ I_w^(i) do
9              if ||x| − |y|| ≤ τ and
               |loc_x − loc_y| ≤ τ and
               Frequency Filter(x, y) = true and
               A[y] is not initialized then
                   A[y] = true
           I_w^(i) ← I_w^(i) ∪ (x, loc_x)    /* set local index */
15         if I_w^G[|I_w^G|] ≠ i then   /* set global index */
16             Add i to the end of I_w^G;
       Verify (x, A);
   return S_i;
```

4. Partitioning Data with Frequency Vectors

4.1 Overview of Data Partition

Frequency Filter captures the dissimilarity between strings by exploiting the fact that L_1-distance between the frequency vectors of similar strings must be small. This fact also means that two different strings with a same frequency vector can not be filtered by Frequency Filter. But, the L_1-distance computation between such frequency vectors should be avoided. With this comes immediately a trivial data partitioning strategy, *i.e.,* to group together all strings with a same frequency vector into a data chunk. However, data chunks generated in this way tend to be very small and the number of data chunks is huge, even for small alphabet and small frequency ranges. As a result, the total number of saved L_1-distance computation is also small.

To address this issue, we propose to split the frequency ranges of characters into small intervals. And, strings with their frequency vectors falling into a same group of intervals, are grouped into a data chunk. The number of data chunks grows at an exponential rate with the number of split intervals, as well as the number of the characters used to partition the string set. Too many data chunks will result in small chunks, which is not expected. Thus, Two parameters are used to control the number of data chunks. Both can be adjusted according to τ and the size of dataset.

Parameter setting. θ is the number of characters used to partition string set. κ is the expected number of split intervals of each frequency range.

Data partition. Now, we assume $\alpha_1, \cdots, \alpha_\theta$ be partition characters and each α_j's frequency range $[m_j, M_j]$ be split into k_j small intervals by an array of split points $P_j[0 : k_j]$ such that (1)$P_j[0] = m_j$, $P_j[k_j] = M_j$; (2) all intervals, except the first one $[m_j, P_j[1]]$ and the last one $[P_j[k_j - 1], M_j]$, have an equal length l_j.

To partition string set $\mathcal{R}$, the algorithm FrePartition (as shown in Algorithm 3) processes each string s sequentially (Line 2-Line 15). For each string s, it first computes an $id = (v_1, \cdots, v_\theta)$ for s, such that the frequency $f_j(s)$ falls into α_j's v_jth interval (Line 4-Line 10). id is taken as the identifier of a data chunk. Then, the algorithm checks whether there is an existing data chunk R_k such that $R_k.id = id$ (Line 11). If yes, it puts s into R_k (Line 12). Else, it creates a new data chunk (Line 14), sets id as its identifier and puts s into it (Line 15). Finally, all generated data chunks $R_1, \cdots, R_p$ are returned.

Algorithm 3: FrePartition $(\mathcal{R})$

Input : string set R, selected characters $\alpha_1, \cdots, \alpha_\theta$
Output: a partition of R.

```
1  p ← 0;
2  foreach s ∈ R do
3      for j = 1 to θ do
4          if f_j(s) ≤ P_j[1] then
5              v_j = 1;
6          else if f_j(s) > P_j[k_j − 1] then
7              v_j = k_j;
8          else
9              v_j = ⌈(f_j(s) − P_j[1])/l_j ⌉ + 1;
10     id ← (v_1, ··· , v_θ);
11     if there is k ∈ [1, p] such that R_k.id = id then
12         R_k ← R_k ∪ {s};
13     else
14         p ← p + 1;
15         R_p ← {s};    R_p.id ← id;
16 return R_1, ··· , R_p;
```

Section 4.2 discusses splitting the frequency range of characters. Section 4.3 defines the chunk distance and presents a greedy strategy for character selection. Section 4.4 develops another strategy, *Z-folding-Combined-Character*, to enhance the pruning effectiveness of data partitioning.

4.2 Range Split

Let $\alpha_j \in \Sigma$, $\mathcal{R}$ be the string set, N be the number of strings in $\mathcal{R}$, $[m_j, M_j]$ be α_j's frequency range. In addition, let $h_j[m_j : M_j]$ store the document frequencies of α_j's each frequency value, *i.e.,* $h_j[i]$ is the total number of $\mathcal{R}$'s strings in each of which α_j appears exactly i times. Via $h_j[m_j :$

$M_j]$, α_j's average frequency in $\mathcal{R}$ can be calculated as $\mu_j = \sum_{i=m_j}^{M_j} i \cdot (h_j[i]/N)$, and α_j's frequency deviation is $\sigma_j^2 = \sum_{i=m_j}^{M_j}((i-\mu_j)^2 \cdot (h_j[i]/N))$.

To split $[m_j, M_j]$ into small intervals with equal length, one direct method is to view $h[m_j : M_j]$ as an usual histogram and use optimal interval length $l_j \approx 3.49(M_j - m_j)^{-1/3}\sigma_j$ given in [25]. However, like many other methods of the same kind, this optimal interval length is designed to produce another histogram such that the expectation of the errors between the new histogram and the original one is minimized. Applying this optimal length directly will lead to some very small data chunks, as well as the number of intervals being not under control.

Instead, we adopt a tailing strategy and a *try-and-refine* method. The tailing strategy is used to avoid generating small data chunks by guaranteeing that a large number of strings s have $f_j(s)$ fall into the the first interval and the last interval. The try-and-refine method is used to control the number of intervals. It tries to split $[m_j, M_j]$ into intervals of length $l_j \approx 3.49(M_j - m_j)^{-1/3}\sigma_j$. If the interval number is far from κ, it adjusts the interval length accordingly and tries again.

RangeSplit, as shown in Algorithm 4, is used to split frequency range $[m_j, M_j]$. To avoid summing values in $h_j[m_j : M_j]$ repeatedly, it first cumulates values in $h_j[m_j : i]$ into $H_j[i]$ for $(m_j \le i \le M_j)$(Line 1-Line 3). Then, it invokes IntervalSplit to split $[m_j, M_j]$ into intervals of initial length $3.49(M_j - m_j)^{-1/3}\sigma_j$ (Line 4-Line 5). The interval number k and an array $P[0:k]$ of partition points are returned. If k is too large or too small, compared with κ, then the adjusting step *inc* is set to be $+1$ or -1, respectively (Line 6-Line 9). Then, it tries to split with the new interval length repeatedly until k equals κ approximately (Line 10-Line 14) or the new try makes thing worse (Line 15-Line 16). At last, the algorithm returns the final interval length l_j, the interval number k_j, and the array $P[0:k_j]$ of partition points.

Algorithm 4: RangeSplit (α_j)

```
1  H_j[m_j] ← h_j[m_j];
2  for i = m_j + 1 to M_j do
3  |  H_j[i] ← H_j[i − 1] + h_j[i];
4  l_j ← 3.49(M_j − m_j)^{−1/3} σ_j;
5  k_j, P_j ← IntervalSplit(H_j, l_j);
6  if k_j > κ + 1 then
7  |  inc ← 1;
8  else if k_j < κ − 1 then
9  |  inc ← −1;
10 while |k_j − κ| > 1 do
11 |  l_j ← l_j + inc;
12 |  k, P ← IntervalSplit(H_j, l_j);
13 |  if |k − κ| < |k_j − κ| then
14 |  |  k_j ← k; P_j ← P;
15 |  else
16 |  |  l_j ← l_j − inc;
17 |  |  break;
18 return k_j, P_j[0 : k_j], l_j;
```

IntervalSplit, as shown in Algorithm 5, splits range $[m_j, M_j]$ into intervals with a given length l_j and returns the interval number k, the array $P[0:k]$ of partition points. The tailing strategy, *i.e.,* the *while conditions* is implemented in Line 3 and Line 6.

Algorithm 5: IntervalSplit ($H[0 : M_j], l_j$)

```
1 k ← 1, i ← m_j;   P[0] ← m_j;
2 while H_j[i] < (1/2)(H_j[i + l_j] − H_j[i]) do
3 |  i ← i + 1;
4 P[k] ← i,   k ← k + 1;
5 while H_j[i + l_j] − H_j[i] < 2(H_j[M_j] − H_j[i + l_j]) do
6 |  i ← i + l_j;   P[k] ← i;   k ← k + 1;
7 P[k] ← M_j;
8 return k, P[0 : k];
```

4.3 Character Selection and Chunk Distance

Given $\alpha_j \in \Sigma$ and the string set R, all statistics are known. Thus, $[m_j, M_j]$'s split, as well as its interval length l_j and interval number k_j, is determined by Algorithm 4. In what follows, l_j, k_j, and the interval $[P_j[i-1]+1, P_j[i]]$ will be called as α_j's interval length, α_j's interval number, and α_j's ith interval respectively, without any ambiguity.

For $s_1, s_2 \in \mathcal{R}$, if $f_j(s_1)$ and $f_j(s_2)$ fall in α_j's i_1th interval and α_j's i_2th interval, then α_j will contribute at least $\max(0, (|i_1 - i_2| - 1) \cdot l_j)$ to the L_1-distance between $f(s_1)$ and $f(s_2)$.

Definition 6: The *split distance* between α_j's i_1th interval and i_2th interval $(1 \le i_1, i_2 \le k_j)$ is defined as $\max(0, (|i_1 - i_2| - 1) \cdot l_j)$, denoted as $\delta_j(i_1, i_2)$.

Lemma 2: *If i_1, i_2 is chosen randomly and uniformly from $[1, k_j]$, then $E(\delta_j(i_1, i_2)) > \frac{(k_j-1)(k_j-2)}{3k_j} l_j$.* □

Proof: $P_r(\delta_j(i_1,i_2) < l_j) = \frac{1}{k_j} + \sum_{j=1}^{k_j-1} \frac{2}{k_j^2} = \frac{k_j+2(k_j-1)}{k_j^2}$,and $P_r(\delta_j(i_1,i_2) = kl_j) = \sum_{j=1}^{k_j-k} \frac{2}{k_j^2} = \frac{2(k_j-k)}{k_j^2}$ for $1 \le k \le k_j - 2$. Therefore, $E(\delta_j(i_1,i_2)) > \sum_{k=1}^{k_j-2} kl_j \cdot P_r(\delta_j(i_1,i_2) = kl_j) = \frac{(k_j-1)(k_j-2)}{3k_j} l_j$. □

Definition 7: The *pruning ability* of character α_j is defined to as $\frac{(k_j-1)(k_j-2)}{3k_j} l_j$.

Greedy-Selected-Character Algorithm. Sort characters of Σ in descending order of their pruning ability as $\alpha_1, \cdots, \alpha_m$. Return $\alpha_1, \cdots, \alpha_\theta$ as well as their interval lengths, interval numbers and division point arrays.

Chunk Distance. Now, let $\alpha_1, \cdots, \alpha_\theta$ be selected as partition characters, and $R_1, \cdots, R_p$ be data chunks returned by FrePartition. We consider the lower bound of edit distance between any strings from two data chunks.

Consider data chunks R_i and R_j, with $R_i.id = (v_{i1}, \cdots, v_{i\theta})$ and $R_j.id = (v_{j1}, \cdots, v_{j\theta})$ respectively. For any $s \in R_i$ and $t \in R_j$, each α_k $(1 \le k \le \theta)$ contributes at least $\max(0, (|v_{ik} - v_{jk}| - 1) \cdot l_k)$ to $\|f(s) - f(t)\|_{L_1}$. Thus, the total contribution of $\alpha_1, \cdots, \alpha_\theta$ gives a lower bound of $\|f(s) - f(t)\|_{L_1}$.

Definition 8: The *chunk distance* $dis(R_i, R_j)$ between data chunks R_i and R_j, with $R_i.id = (v_{i1}, \cdots, v_{i\theta})$ and $R_j.id = (v_{j1}, \cdots, v_{j\theta})$ respectively, is defined as $dis(R_i, R_j) = \sum_{k=1}^{\theta} \max(0, (|v_{ik} - v_{jk}| - 1) \cdot l_k)$.

Proposition 3: *Let R_i, R_j be data chunks, and $s \in R_i, t \in R_j$. Then, $dis(R_i, R_j) > 2\tau$ implies $ed(s,t) > \tau$.* □

Proposition 4: *If R_i, R_j be data chunks drawn from $R_1, \cdots, R_p$ randomly and uniformly, then $E(dis(R_i, R_j)) > \sum_{j=1}^{\theta} \frac{(k_j-1)(k_j-2)}{3k_j} l_j$.* □

Discussion. (1) $E(dis(R_i, R_j)) >> 2\tau$ implies there are many chunk-pairs with chunk distances larger than 2τ. All string pairs drawn from such data chunks are dissimilar.

(2) For small τ and reasonable θ, partition characters chosen by *Greedy-Selected-Character* algorithm enable $E(dis(R_i, R_j))$ large enough. For example, $k_j = 6$, $l_j = 2$ and $\theta = 5$ enable $E(dis(R_i, R_j)) > 11$, which is larger enough for $\tau = 1, 2, 3, 4$.

(3) l_j is determined by α_j's frequency deviation σ_j^2 to a large extent. In one hand, the larger σ_j^2 is, the wider the frequency range $[m_j, M_j]$ spans. In the other hand, the initial value of l_j is $3.49(M_j - m_j)^{-1/3}\sigma_j$. The experimental results show that Greedy-Selected-Character algorithm always choose these characters with larger σ_j^2.

(4)For large τ, there is an method to exploit the unchosen characters to enlarge l_js significantly, as shown in next subsection.

(5) The sizes of data chunks generated by FrePartition are unbalanced, because each character's frequency follows a normal distribution approximately. However, our method can guarantee that no data chunks with very small size are generated. On the other aspect, if the data partition is used to guarantee balanced sizes of data chunks, then many similar strings will fall into different data chunks. Contrast to this, our method can find most of similar strings by joining all data chunks with themselves.

4.4 Z-folding Combination of characters

Let $\Sigma_1, \cdots, \Sigma_\theta$ be a partition of the alphabet Σ, s be a string of R. Each Σ_i $(1 \le i \le \theta)$ is called as a combined character associated with the partition. Σ_i's combined frequency in string s, denoted as $f_i^{(\Sigma_i)}(s)$, is the total times of appearances in s of all characters of Σ_i, *i.e.,* $f_i^{(\Sigma_i)}(s) = \sum_{\alpha_j \in \Sigma_i} f_j(s)$. Vector $< f_1^{(\Sigma_1)}(s), \cdots, f_\theta^{(\Sigma_\theta)}(s) >$ is referred to as the combined frequency vector of s associated with the partition and written it as $f^{(c)}(s)$. With the aid of the average frequency μ_j and deviation σ_j^2 of each character α_j, the average frequency μ_{Σ_i} of combined character Σ_i can be computed as $\mu_{\Sigma_i} = \sum_{\alpha_j \in \Sigma_i} \mu_j$ and the deviation $\sigma_{\Sigma_i}^2$ of Σ_i can be computed as $\sigma_{\Sigma_i}^2 = \sum_{\alpha_j \in \Sigma_i} \sigma_j^2$.

Lemma 5: *If the L_1-distance between the combined frequency vector $f^{(c)}(s_1)$ of strings s_1 and the combine frequency vector $f^{(c)}(s_2)$ of sting s_2 is larger than 2τ,* i.e., *$\|f^{(c)}(s_1) - f^{(c)}(s_2)\|_{L_1} > 2\tau$, then $ed(s_1, s_2) > \tau$.* □

Proof: It follows immediately Lemma 1 and the fact that $|f_i^{(\Sigma_i)}(s_1) - f_i^{(\Sigma_i)}(s_2)| \le \sum_{\alpha_j \in \Sigma_i} |f_j(s_1) - f_j(s_2)|$ holds for all $1 \le i \le \theta$. □

Given a partition $\Sigma_1, \cdots, \Sigma_\theta$ of Σ and the statistics of each combined character Σ_i $(1 \le i \le \theta)$. RangeSplit (Algorithm 4) can treat each Σ_i as an usual character and split its frequency range into small intervals. Thus, the interval length, interval number and vth interval of Σ_i is determined similarly. Together with Proposition 4, the following proposition states the utility of combined characters.

Proposition 6: *If Σ_i is a combined character and $\alpha_j \in \Sigma_i \cap \Sigma$, then the ratio between the interval length of Σ_i and that of α_j is $(\sigma_{\Sigma_i}/\sigma_j)^{3/2}$.* □

Proof: Essentially, RangeSplit solves the equation $l_j = 3.49((\kappa - 2)l_j)^{-1/3}\sigma_j$ approximately and iteratively. □

Now, the remaining issue is how to break up alphabet Σ into combined characters $\Sigma_1, \cdots, \Sigma_\theta$ such that the expectation of chunk distances is maximized. This means $\sum_{i=1}^{\theta} l_i$ reaches maximum, according to Proposition 4 and $k_i \approx \kappa$. Moreover, l_i increases with the increase of σ_{Σ_i} and $\sum_{i=1}^{\theta} \sigma_{\Sigma_i}^2 = \sigma_\Sigma^2$ is a constant. This suggests that the partition of Σ should make all $\sigma_{\Sigma_i}^2$ $(1 \le i \le \theta)$ equal approximately. Here comes the following Z-folding algorithm.

Z-Folding-Combined-Character Algorithm. Sort characters of Σ in descending order of their standard deviations as $\alpha_1, \cdots, \alpha_m$. For k from 0 to $\lfloor m/\theta \rfloor$, allot $\alpha_{k+1}, \cdots, \alpha_{k+\theta}$ to $\Sigma_1, \cdots, \Sigma_\theta$ as follows. If k is even, assign α_{k+j} to Σ_j. If k is odd, assign α_{k+j} into $\Sigma_{\theta-j+1}$. Return $\Sigma_1, \cdots, \Sigma_\theta$ as well as their interval lengths, interval numbers and partition point arrays.

Example 5: Let $\theta = 3$, $\kappa = 6$. Consider the string set $\mathcal{R}$ given in Example 1.

(1)After all characters are sorted in descending order of their deviations, the Z-folding-combined character algorithm breaks up the English alphabet into three subsets $\Sigma_1, \Sigma_2, \Sigma_3$, as shown in Example 1.

(2)Then, the combined frequency vector of each string is calculated from the frequency vector. At the same time, the frequency range, as well as the deviation, of each combined character is figured out. For example, the combined frequency vector of s_{10} and s_{18} are $(4, 4, 2)$ and $(5, 5, 1)$, respectively; and the frequency range of $\Sigma_1, \Sigma_2, \Sigma_3$ are $[1, 8]$,$[1, 7]$ and $[0, 8]$ respectively.

(3)RangeSplit splits (a) Σ_1's frequency range $[1, 8]$ into four intervals $[0, 2]$, $[3, 4]$, $[5, 6]$, $[7, 8]$; (b) σ_2's frequency range $[1, 7]$ into four intervals $[1, 2]$, $[3, 4]$, $[5, 6]$, $[7, 7]$; (c) σ_3's frequency range $[0, 7]$ into four intervals $[0, 1]$, $[2, 3]$, $[4, 5]$, $[6, 7]$. Thus, interval number $k_1 = k_2 = k_3 = 4$ and interval length $l_1 = l_2 = l_3 = 2$.

(4) Using the split frequency ranges, FrePartition groups strings in $\mathcal{R}$ into five data chunks C_1, c_2, c_3, C_4, C_5 with $(0, 1, 2)$, $(1, 0, 2)$, $(1, 1, 0)$, $(1, 2, 2)$, $(2, 2, 0)$ as chunk *id* respectively.

(5)According to Definition 8, the chunk distances can be evaluated. For example, $dis(C_2, C_5) = 5$. □

5. Partition-Based Similarity Join Algorithm

In this section, we present the partition-based edit similarity join algorithm FrepJoin. Section 5.1 describes the two-level index. Section 5.2 develops the algorithm. Finally, Section 5.3 discusses the effects of FrepJoin algorithm.

5.1 Two-Level Inverted Indices

After the string set $\mathcal{R}$ is partitioned by FrePartition into data chunks $R_1, \cdots, R_p$, the partition-based similarity join algorithm works in two stages, the *intra-join stage* and *the inter-join stage*. In intra-join stage, it invokes Ed-Join (or any other more efficient algorithm if there is) to join each data chunk R_i with itself and build inverted index $I^{(i)}$ for R_i. In the inter-join stage, it exploits the inverted indices, built in intra-join stage, to join each data chunk R_i with

every other data chunk R_j $(i < j)$ with $dis(R_i, R_j) \leq 2\tau$.

A direct method to accomplish the inter-join stage can work in the same way as Ed-Join, *i.e.,* for each token w in the $q\tau+1$-prefix of each string x in R_i, it first tries to access the index item $I_w^{(j)}$ in the inverted index $I^{(j)}$ of R_j to find out from R_j the strings y which contain token w; then applies filters on each such string pair (x, y).

However, this method will lead to many unnecessary index accesses, since R_j is much smaller than the whole dataset $\mathcal{R}$, and thus the index item $I_w^{(j)}$ is much likely to be *null*. These unnecessary index accesses are time-consuming and counteract the effects of data partition method. To address this issue, we propose to adopt two-level inverted indices.

Local index $I^{(i)}$ for data chunk R_i is just the inverted index of R_i, which can be built by Ed-Join in the intra-join stage as shown in Algorithm 2. Although, the number of local inverted indices equals the number of data chunks, the total size of all local indices roughly equals the size of the inverted index of the whole dataset $\mathcal{R}$.

Global Index I^G for the partition $R_1, \cdots, R_p$ of the string set $\mathcal{R}$ is also an inverted index which maps each q-gram w to a sorted array I_w^G containing integers $i \in [1, p]$ as entries, where each entry i identifies a data chunk R_i that contains at least one string with w appearing in its $q\tau + 1$-prefix.

The global index can also be built by Ed-Join in the intra-join stage, which is shown in Line 15 - Line 16 of Algorithm 2. Specifically, in intra-join stage the algorithm processes each data chunk R_i in the ascending order of its index i. And at the end of processing each token w of a string in data chunk R_i, the algorithm checks whether i is the last entry of I_w^G or not. If not, it is the first time to process w during self-join of R_i and i is added as the last entry of I_w^G. Else, i is already in I_w^G and nothing needs to be done.

The size of the global index is very small, compared with the total size of all local indices, as shown by our experiments. The cost to build the global index can be ignored, compared with the total time of the intra-join stage.

5.2 FrepJoin Algorithm

The FrepJoin algorithm, as shown in Algorithm 6, first calls FrePartition (Algorithm 3) to partition string set $\mathcal{R}$ into data chunks $R_1, \cdots, R_p$ (Line 3). Then, it invokes Ed-Join (*i.e.,* Algorithm 2) for each data chunk R_i to accomplish intra-joins, as well as to build the local index $I^{(i)}$ and the global index I^G (Line 5-Line 6). In inter-join stage (Line 7-Line 28), it joins each data chunk R_i with other data chunks R_j $(j > i)$. To do this, it follows the *filter-and-refine* framework to generate efficiently a candidate set A for each string x in R_i (Line 8-Line 27) and then applies algorithm Verify to verify each candidate pair (x, y) for $y \in A$ (Line 28). The details of the production of the candidate set A will be discussed in next two paragraphs. And procedure Verify() is exactly the one adopted in Ed-Join [33].

The generation of candidate set A benefits a lot from data partition as well as the two-level indices. Both deserve a detailed discussion. Here, we first discuss the benefits of data partition. Before data chunk R_i is inter-joined with other chunks R_j $(j > i)$, ChunkFilter is called (Line 8) to find out an sorted array L of indices j $(j > i)$ of data chunks R_j which R_i has to inter-join with, *i.e.,* $dis(R_i, R_j) \leq 2\tau$. Thus, every other data chunk $R_{j'}$ with $j' \notin L$ are pruned away immediately and its local index will never be accessed, while R_i being processed. ChunkFilter (Algorithm 7) completes this task by checking whether or not the chunk distance between R_i and R_j is larger than 2τ according to Definition 8.

Now, let's consider the benefits of the two-level indices. For each token w of each $x \in R_i$, FrepJoin only needs to access the local index item $I_w^{(j)}$ of data chunk R_j for each $j \in L$ to check whether there is an entry $(y, loc_y) \in I_w^{(j)}$ such that (x, y) is a valid candidate pair. However, $I_w^{(j)}$ is likely to be *null*. The global index I^G can be used to avoid such unnecessary index access. In fact, I_w^G is an sorted array of indices j of data parts R_j such that w appears in at least one string of R_j. Thus, only the local index of data chunk R_j with $j \in L \cap I_w^G$ is needed to be accessed. In view of this, FrepJoin uses a process, which is similar to the merge-stage of MergeSort algorithm, to find out the elements of $L \cap I_w^G$ (Line 14-Line 20). Specifically, it sets a as the starting position of L, and applies binary-search on I_w^G to find the first position b such that $I_w^G[b] \geq i$ (Line 14). If $L[a] \neq I_w^G[b]$, the algorithm increases a or b correspondingly (Line 16-Line 19). In the case of a common element $L[a]$ of $L \cap I_w^G$ being found, the algorithm accesses each entry (y, loc_y) of local index item $I_w^{(L[a])}$ and check whether (x, y) can pass through the filters (Line 21-Line 24).

Algorithm 6: FrepJoin $(\mathcal{R}, \tau)$

Input : String set $\mathcal{R}$, edit distance upper bound τ
Output: Pair set S of strings in $\mathcal{R}$ with $ed(\cdot,\cdot) \leq \tau$.

```
1  S ← ∅;
2  I_j^G ← ∅ (1 ≤ j ≤ U)                          /* global index */
3  I_j^(i) ← ∅(1 ≤ i ≤ p, 1 ≤ j ≤ U)              /* local index */
4  R_1, ··· , R_p ← FrePartition(R);
5  for i = 1 to p do                               /* intra-join */
6  |  S ← S∪Ed-Join (R_i, τ);
7  for i = 1 to p do                               /* inter-join */
8  |  L ← ChunkFilter(i);
9  |  foreach x ∈ R_i do
10 |  |  A ← empty map from id to boolean;
11 |  |  px = x.px;
12 |  |  for j = 1 to px do
13 |  |  |  w ← x[j].token,   loc_x ← x[j].loc;
14 |  |  |  a ← 1,   b ← BinarySearch (I_w^G, i);
15 |  |  |  while a ≤ |L| and b ≤ |I_w^G| do
16 |  |  |  |  if L[a] < I_w^G[b] then
17 |  |  |  |  |  a ← a + 1;
18 |  |  |  |  else if L[a] > I_w^G[b] then
19 |  |  |  |  |  b ← b + 1;
20 |  |  |  |  else
21 |  |  |  |  |  foreach (y, loc_y) ∈ I_w^(L[a]) do
22 |  |  |  |  |  |  if ||x| − |y|| ≤ τ and
23 |  |  |  |  |  |  |loc_x − loc_y| ≤ τ and
24 |  |  |  |  |  |  Frequency Filter(x, y) = true and
25 |  |  |  |  |  |  A[y] not initialized then
26 |  |  |  |  |  |  |  A[y] = true
27 |  |  |  |  |  a ← a + 1,   b ← b + 1;
28 |  |  Verify (x, A);
29 return S;
```

5.3 The effects of Partition-Based Algorithm

FrepJoin leverages data partitioning and frequency filtering to target the two time-consuming factors in edit sim-

Algorithm 7: ChunkFilter (i)

Input : the index i of data chunk R_i
Output: a sorted list of j $(> i)$ s.t. $dis(R_i, R_j) \leq 2\tau$

```
1 L ←empty list;
2 for j = i + 1 to p do    /* p is # of data chunks */
3 |  if dis(R_i, R_j) ≤ 2τ then    /* chunk distance */
4 |  |  Add j to the end of L;
5 return L;
```

ilarity join, *i.e.,* the enumeration of candidate pairs and the computation of edit distances, respectively. This makes FrepJoin an efficient algorithm for edit similarity join.

From bird's eye perspective, data partitioning enables FrepJoin to avoid enumerating string pairs in different data chunks with chunk distances greater than 2τ. And Frequency Filter enables FrepJoin to avoid computing edit distances between dissimilar strings. Besides, FrepJoin integrates the data partitioning and Frequency Filter with the existing filters. Thus, the filtering effectiveness of these existing filters are also fully exploited.

From another perspective, FrepJoin's efficiency comes from (a) accessing only relevant local inverted indices; and (b) exploiting the ability of Ed-Join to skip over irrelevant entries in each relevant local indices; and (c) saving a large amount of computation.

Now, let's see more details of these three aspects.

For (a), before edit similarity join, FrepJoin partitions string set $\mathcal{R}$ into small data chunks $R_1, \cdots, R_p$ and builds inverted index $I^{(i)}$ for each data chunk R_i $(1 \leq i \leq p)$. Note that, $\cup_{i=1}^{p} I_w^{(i)}$ indexes all appearances of token w in $q\tau + 1$-prefixes of strings in $\mathcal{R}$. When processing token w of string s in data chunk R_j, only a small part of entities in $\cup_{i=1}^{p} I_w^{(i)}$ is accessed. In fact, $I_w^{(i)}$ is accessed if and only if $i \in L \cap I_w^G$ (as illustrated in Figure 2), where I_w^G records the data chunks in which w appears and L records the data chunks which possibly contains strings similar to s. Other local indices are skipped over directly.

For (b), since each local index are built in a same way of Ed-Join (*i.e.,* all entries in the index are sorted in ascending order of string length), during accessing local index item $I_w^{(i)}$, the ability of Ed-Join to skip over irrelevant index entries, *i.e.,* (t, loc_t) with $||s| - |t|| > \tau$, are also exploited (as illustrated in Figure 2).

Putting (a) and (b) together, FrepJoin enumerates candidate pairs in a more economic way, compared with Ed-Join.

For (c), FrepJoin applies Frequency Filter before Content Filtering is executed. Many candidate pairs of dissimilar strings are identified and pruned away. As a result, a remarkable proportion of computation executed by the Content Filter, as well as edit distance computation, are saved.

Besides the saved computation, it deserves to point out that Frequency Filter contributes to the efficiency of FrepJoin in other ways. For long strings, the Frequency Filter results in a higher performance, because Frequency Filter's time complexity ($O(|\Sigma|)$ is lower than than that of the Content Filter. While for short string (*i.e.,* $q\tau+1$ exceeds the average string length), Frequency Filter leads to higher performance because it avoids ineffective filtering. In fact, the main reason for low efficiency of Ed-Join on short strings stems from the fact that its Content Filter can hardly catch the dissimilarity of strings by applying Lemma 1 on very small probing windows specified by the mismatching q-grams. However, in FrepJoin, the ineffective filtering computation executed by Content Filter is avoided to a large extent.

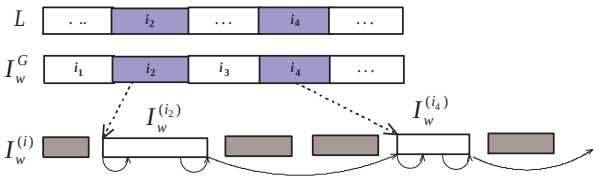


Figure 2: The effect of FrepJoin

6. Experimental Evaluation

Next, we present experimental results and our analysis.

6.1 Experimental setting

We implemented and used the following algorithms in the experiments.

Ed-Join integrated the existing filters and was the most efficient edit similarity join algorithm for long strings [33]. After discussing with the authors, we implement it.

Ed-Join+FF is the Ed-Join algorithm with Frequency Filter added after all exiting filters.

FF+Ed-Join is the Ed-Join algorithm with Frequency Filter added after Prefix Filter and Length Filter.

Trie-Join used a tire-structured index to filter candidate pairs with dissimilar prefixes. It was more efficient than Ed-Join on short strings, as reported. We implemented the trie-path-stack algorithm in [32].

FrepJoin is our proposed algorithm with data partition to avoid enumerating a large part of string pairs and with the Frequency Filter to reduce candidate pairs further.

All algorithms were implemented as in-memory algorithms with C++, with all their inputs loaded into the memory before they were run. Moreover, the data partition and the Frequency Filter only used the English alphabet plus a wild card. Symbols not in the English alphabet are mapped to the wild card there.

In [33], Ed-Join is shown to consistently outperform All-Pairs [3]. And, All-Pairs is shown in [3] to outperform other alternative algorithms such as PartEnum [2], ProbeCount-Sort [24]. Therefore, we did not consider them.

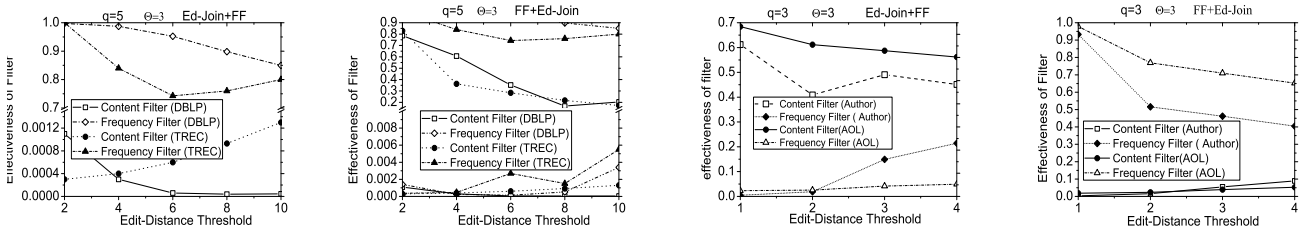
All experiments were performed on an IBM x3650 M3 System with Intel(R) Xeon 2.67GHz 4-core CPU and 8GB RAM. The operating system is scientific Linux. The algorithms were compiled using g++ 4.4.4 with -03 flag.

We used four public available real datasets to evaluate our methods. They were chosen to cover strings of different average length and different content. The detailed statistics information of these datasets are listed in Fig. 6.

DBLP. This dataset is extracted from a snapshot XML document of the bibliography records. [2] It has about 1.0M records; each record is a concatenation of author name(s) and the title of a publication. DBLP dataset is widely used in edit similarity join and near-duplicate detection research [30, 32, 33, 34].

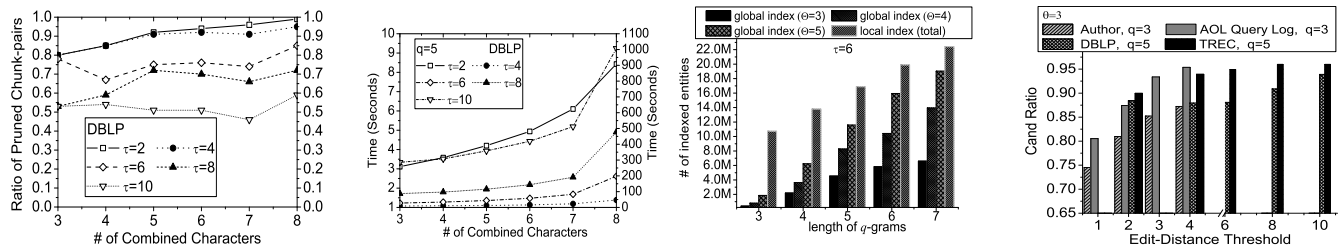
TREC. This dataset is extracted from TREC-9 Filtering Track Collections.[3] It contains about 2.5M references

[2]http://www.informatik.uni-trier.de/ ley/db
[3]http://trec.nist.gov/data/t9_filtering.html



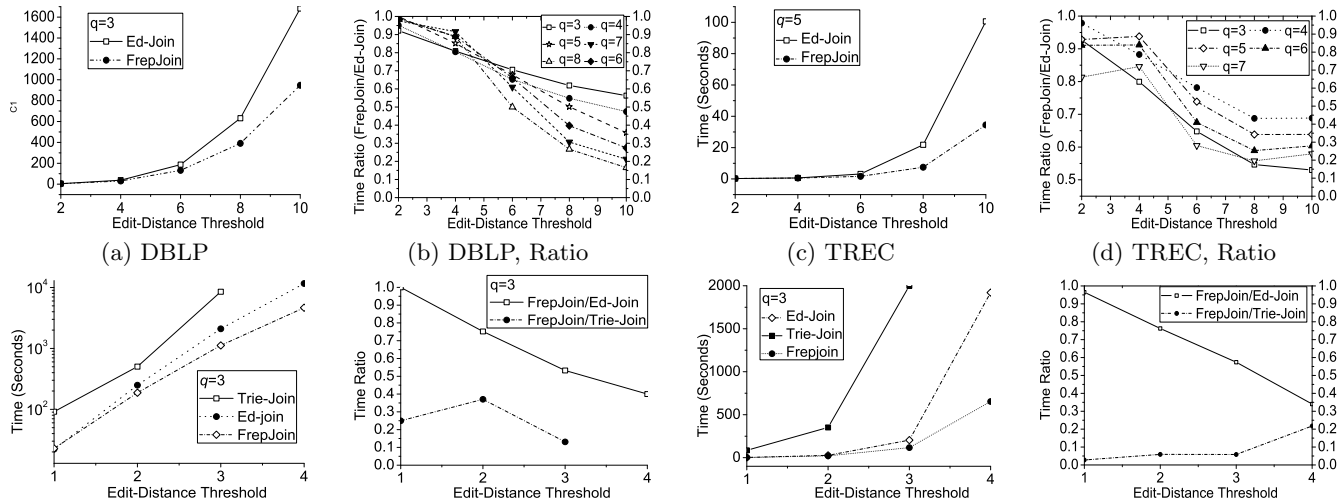
(a) Ed-Join +FF, long strings (b) FF+Ed-Join, long strings (c) Ed-Join +FF, short strings (d) FF+Ed-Join, short strings

Figure 3: The Independence of Frequency Filter



(a) Filtering Effectiveness (b) time VS # of characters (c) # of indexed entities (d) Ratio of Candidate pairs

Figure 4: Performance Evaluation of Data Partitioning



(a) DBLP (b) DBLP, Ratio (c) TREC (d) TREC, Ratio

(e) Author (f) Author, Ratio (g) AOL Query Log (h) AOL Query Log, Ratio

Figure 5: Performance Evaluation of FrepJoin

from the MEDLINE database. We extracted author, title, and abstract fields to form records.

DBLP Author. This dataset consists of about 1M author names extracted from the DBLP dataset with exact duplicates eliminated.

AOL Query Log. This dataset consists of about 0.36M web queries extracted from a snapshot of the AOL query log [4], which contains about 20M web queries collected from 650k users over three months.

These datasets were transformed and cleaned as in [33]. The frequencies of single characters in each strings were counted at the time of tokenizing q-grams.

6.2 Experimental results.

Three sets of experiments were conducted to evaluate:

[4]http://www.gregsadetsky.com/aol-data/

Datasets	sizes	avg_len	$\|\Sigma\|$	Content
Author	810,731	13.90	27	author
AOL Query Log	365,239	21.211	37	web queries
DBLP	999,546	113.630	37	author, title
TREC	249,987	880.075	37	author,title,abstract

Figure 6: Dataset Statistics

(1) the Frequency Filter's independence of the existing filters mentioned in Section 3, by comparing the filtering effectiveness of filters in algorithm Ed-Join +FF and FF+Ed-Join.

(2) the performance of the data partition, by considering the filtering effectiveness of data partitioning, the impacts of parameters on running time, the size of the global index, and the impact of data partitioning on the size of candidate set.

(3) the efficiency of FrepJoin for edit similarity join, compared with (a) Ed-Join, reported as the best algorithm

for long strings [33], and (b) Trie-Join [32], which is more efficient than Ed-Join on short strings.

We next present our findings.

Exp-1: Independence of Frequency Filter. We first evaluated the independence of Frequency Filter by comparing the filtering effectiveness of different filters in algorithms Ed-Join +FF and FF+Ed-Join, obtained by adding Frequency Filter in different positions of Ed-Join. In Ed-Join +FF, the input of Content Filter is the candidate set after length filtering and prefix filtering, and the output of Content Filter is the input of Frequency Filter. In FF+Ed-Join, the roles of the Content Filter and the Frequency Filter are switched.

The filtering effectiveness of a filter is defined as the ratio of the size of the filter's output to the size of its input. The smaller the ratio is, the higher the effectiveness is.

We ran both Ed-Join +FF and FF+Ed-Join on all four datasets with different edit distance thresholds. Fig. 3(a) and Fig. 3(b), where the x-axis are labeled with edit distance threshold and the y-axis are labeled with the filtering effectiveness, show the the filtering effectiveness of different filters on long strings (DBLP and TREC). Fig. 3(c) and Fig. 3(d) show similar results on short strings (Author and AOL Query Log).

From the results, we observe that (1) both Content Filter and Frequency Filter can filter the candidate sets effectively, when they are applied first. For example, more than 99% of candidate pairs can not pass through either Frequency Filter or Content Filter. (2) the output of Content Filter can be further filtered effectively by Frequency Filter, both for long strings (Fig. 3(a)) and short strings (Fig. 3(c)). This is more evident on short strings. For example, less than 70% of the output of Content Filter can pass through Frequency Filter. (3) the output of Frequency Filter can also be further filtered effectively by Content Filter, both for long strings (Fig. 3(b)) and short strings (Fig. 3(d)). (4) Frequency Filter and Content Filter are independent.

Exp-2: Performance of Data partitioning. In this set of experiments, we evaluated the performance of data partitioning on all four datasets, by considering (1) its effectiveness of filtering data-chunk pairs; (2) the effects of combined characters on the running time of the join algorithm; (3) the size of the indices; and (4)the impact of data partitioning on the size of the candidate set.

We ran FrepJoin on all four datasets with θ (the number of combined character) varying from 3 to 8. On Author dataset and AOL Query Log dataset, the length of q-gram is fixed to be 3 and edit distance threshold τ ran from 1 to 4. On DBLP dataset and TREC dataset, q ran from 3 to 8 and τ from 2 to 10.

For (1), the chunk distances between different data chunks, generated for each θ, are computed. And, The ratio of chunk pairs with chunk distance larger than 2τ to all chunk pairs are computed for different τ. Fig. 4(a) shows the result on DBLP dataset. Results on other datasets are similar and not reported.

For (2), the running time of the algorithm are recorded for different parameter setting. Fig. 4(b) shows the results for $q = 5$ on DBLP dataset. Other results are similar and not reported.

For (3), the number of indexed entries in the global index and the total number of index entries in the local indices are counted for different q, θ and τ. Fig. 4(c) show the results for $\tau = 6$ on DBLP dataset. Other results are not reported.

For (4), the total number of candidate pairs generated before Line 28 of FrepJoin (with Frequency Filter removed) are counted for different parameter settings. And the number of candidate pairs generated in Ed-Join, before Content Filter are applied, are also counted. The ratio of the former to the latter (with same parameter setting) are computed. Fig. 4(d) shows the results on all four datasets.

The following observations can be made. Firstly, a notable proportion of data chunks can be pruned away directly. Secondly, partition-based method can enumerate smaller candidate set in a more efficient way. Thirdly, the running time of partition-based method, as well as the size of global index increases rapidly while the number of combined characters increases (especially after $\theta > 5$).

Besides, we also considered the impact of parameter κ on data partitioning. We found that the number of data chunks generated by FrePartition almost unchanged (not shown), when θ was fixed and κ varied reasonably from 5 to 10. Indeed, the actual interval number for each combined character is generated by the algorithm, rather than the expectation value κ itself.

Exp-3: Performance of FrepJoin algorithm. In this set of experiments we evaluated the performance of FrepJoin by comparing its running time with that of Ed-Join and Trie-Join. The running time includes the costs of partitioning data, building indices and joining the datasets. Besides the running time, we also compare the ratios of FrepJoin's running time to other algorithm's running time for different parameter settings. Since combined characters behaves better than single characters in partitioning data (see Section 4), we only consider combined characters here. Moreover, to guarantee better performance of FrepJoin according to the results of Exp-2, we fixed $\theta = 3$, $\kappa = 6$, and allowed other parameters to vary.

To analyze the performance of FrepJoin on long strings, we varied τ from 2 to 10 and q from 3 to 8, and ran FrepJoin, Ed-Join on DBLP dataset and TREC dataset respectively. The results are shown in Fig. 5(a) -Fig. 5(d).

We find that FrepJoin significantly and consistently outperforms Ed-Join. This becomes more evident when τ and q increase, *i.e.,* when the q-gram becomes longer and edit distance threshold becomes larger. The reason for this is, the partition-based method can enumerate smaller candidate set in more efficient way and the Frequency Filter can further reduce the candidate pairs that need edit distance checked.

We also evaluated the performance of FrepJoin for very large τ (not shown). For example, when $q = 5$ and $\tau = 20$, the running time ratio on TREC dataset is $\frac{2142.0s}{3264.6s} \approx 0.66$. When $\tau = 40$, the ratio is $\frac{24909.4s}{29760.8s} \approx 0.84$.

Similarly, to analyze the performance of FrepJoin on short strings, we fixed $q = 3$, varied τ from 1 to 4, and ran FrepJoin, Ed-Join, Trie-Join on Author dataset and AOL Query Log dataset respectively. The results are shown in Fig. 5(e)-Fig. 5(h).

We find that: (1) again, FrepJoin significantly and consistently reduces the time of edit similarity join, compared with both Trie-Join and Ed-Join. This becomes more evident

when τ is increased. (2) Trie-Join needs a great amount of memory when the join results consist of a huge number of string pairs. For example, when $\tau = 4$, the join result on Author dataset contains more than 9M string pairs and the join result on AOL Query Log dataset contains more than 1.7M string pairs. In such cases, Trie-Join always fails because of memory error.

6.3 Summary

From the experimental results we find the following. (1) Our partition-based edit similarity join algorithm significantly and consistently outperform the previous algorithms for both short strings and long strings. Moveover, this becomes more evident when the length of q-grams and the edit distance increase. (2) Partition-based method is more efficient than the all-pairs mechanism to enumerate the candidate set for both short strings and long strings. The number of combined characters adopted in data partitioning should be small. (3) The Frequency Filter is independent of the existing filters. It can be used to further reduce the string pairs that need edit distance checked. (4) The data partitioning techniques and the Frequency Filter are effective in improving the performance of edit similarity join algorithms.

7. Conclusion

We argued that frequencies of single characters, as well as other statistics of strings, can be exploited to prune away efficiently candidate pairs in string similarity join. These information can be used to design an independent filter and to partition dataset into data chunks with chunk distance guaranteed which leads to a fact that a remarkable proportion of candidate pairs can be pruned away without paying the price to enumerate them. Our experimental study has confirmed that our method substantially outperform the existing counterparts.

In the future, we plan to extend our techniques for massive datasets which can not stay in memory, devise parallel versions of the index and the algorithm, and adapt it to a wide variety of similarity measures.

8. References

[1] S. Agrawal, V. R. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD Conference*, pages 359–370, 2004.

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

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

[4] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD Conference*, pages 313–324, 2003.

[5] S. Chaudhuri, V. Ganti, and R. Kaushik. Data debugger: An operator-centric approach for data quality solutions. *IEEE Data Eng. Bull.*, 29(2):60–66, 2006.

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

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

[8] X. L. Dong, A. Y. Halevy, and C. Yu. Data integration with uncertainty. In *VLDB*, pages 687–698, 2007.

[9] T. Ge and Z. Li. Approximate substring matching over uncertain strings. *PVLDB*, 4(11):772–782, 2011.

[10] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

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

[12] M. Hadjieleftheriou and D. Srivastava. Weighted set-based string similarity.

[13] M. R. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR*, pages 284–291, 2006.

[14] J. Jestes, F. Li, Z. Yan, and K. Yi. Probabilistic string similarity joins. In *SIGMOD Conference*, pages 327–338, 2010.

[15] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW*, pages 371–380, 2009.

[16] Y. Kim, K.-G. Woo, H. Park, and K. Shim. Efficient processing of substring match queries with inverted q-gram indexes. In *ICDE*, pages 721–732, 2010.

[17] H. Lee, R. T. Ng, and K. Shim. Similarity join size estimation using locality sensitive hashing. *PVLDB*, 4(11):338–349, 2011.

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

[19] C. Li, B. Wang, and X. Yang. Vgram: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, pages 303–314, 2007.

[20] A. Metwally, D. Agrawal, and A. E. Abbadi. Detectives: detecting coalition hit inflation attacks in advertising networks streams. In *WWW*, pages 241–250, 2007.

[21] G. Navarro and L. Salmela. Indexing variable length substrings for exact and approximate matching. In *SPIRE*, pages 214–221, 2009.

[22] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *SIGMOD Conference*, pages 259–270, 1996.

[23] S. C. Sahinalp, M. Tasan, J. Macker, and Z. M. Özsoyoglu. Distance based indexing for string proximity search. In *ICDE*, pages 125–, 2003.

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

[25] D. W. SCOTT. On optimal and data-based histograms. *Biometrika*, 66:605–610, 1979.

[26] H. Shin, B. Moon, and S. Lee. Partition-based similarity join in high dimensional data spaces. In *DEXA*, pages 741–750, 2002.

[27] Y. N. Silva, W. G. Aref, and M. H. Ali. The similarity join database operator. In *ICDE*, pages 892–903, 2010.

[28] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD Conference*, pages 495–506, 2010.

[29] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21:168–173, January 1974.

[30] J. Wang, G. Li, and J. Feng. Fast-join: An efficient method for fuzzy token matching based string similarity join. In *ICDE*, pages 458–469, 2011.

[31] W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit distance constraints. In *SIGMOD Conference*, pages 759–770, 2009.

[32] S. Wu, D. Jiang, B. C. Ooi, and K.-L. Wu. Efficient b-tree based indexing for cloud data processing. *PVLDB*, 3(1):1207–1218, 2010.

[33] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.

[34] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

[35] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *SIGMOD Conference*, pages 915–926, 2010.