Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

FrepJoin: an efficient partition-based algorithm for edit similarity join

Ji-zhou LUO[‡1,2], Sheng-fei SHI[1], Hong-zhi WANG[1], Jian-zhong LI[1]
([1]*School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China*)
([2]*Guangdong Key Laboratory of Popular High Performance Computers, Key Laboratory of Service Computing and Application, Shenzhen 518000, China*)
E-mail: luojizhou@hit.edu.cn; shengfei@hit.edu.cn; wangzh@hit.edu.cn; lijzh@hit.edu.cn

Received June 17, 2016; Revision accepted Dec. 15, 2016; Crosschecked Nov. 6, 2017

Abstract: String similarity join (SSJ) is essential for many applications where near-duplicate objects need to be found. This paper targets SSJ with edit distance constraints. The existing algorithms usually adopt the filter-and-refine framework. They cannot catch the dissimilarity between string subsets, and do not fully exploit the statistics such as the frequencies of characters. We investigate to develop a partition-based algorithm by using such statistics. The frequency vectors are used to partition datasets into data chunks with dissimilarity between them being caught easily. A novel algorithm is designed to accelerate SSJ via the partitioned data. A new filter is proposed to leverage the statistics to avoid computing edit distances for a noticeable proportion of candidate pairs which survive the existing filters. Our algorithm outperforms alternative methods notably on real datasets.

Key words: String similarity join; Edit distance; Filter and refine; Data partition; Combined frequency vectors
https://dx.doi/10.1631/FITEE.1601347 **CLC number:** TP311.13

1 Introduction

String similarity join (SSJ) is essential for many applications, e.g., coalition detection (Metwally *et al.*, 2007), fuzzy keyword matching (Ji *et al.*, 2009), data integration (Dong *et al.*, 2007), data cleaning (Chaudhuri *et al.*, 2006b), and near-duplicate object detection (Xiao *et al.*, 2008a). It has been applied to solve various practical problems in industrial community; e.g., Google uses it to detect near-duplicate web pages (Henzinger, 2006), and Microsoft adopts it in the Data Debugger Project (Chaudhuri *et al.*, 2006a). Thus, many algorithms (Gravano *et al.*, 2001; Sarawagi and Kirpal, 2004; Arasu *et al.*, 2006; Bayardo *et al.*, 2007; Xiao *et al.*, 2008a; Feng *et al.*, 2012; Qin *et al.*, 2013) have been proposed.

Given two sets of strings, SSJ aims to find all pairs of similar strings from each of the sets according to a threshold τ of the predefined similarity function such as the Jacard distance (Sarawagi and Kirpal, 2004; Xiao *et al.*, 2008b), cosine distance (Bayardo *et al.*, 2007), edit distance (Xiao *et al.*, 2008a; Feng *et al.*, 2012; Qin *et al.*, 2013), and their variants (Chaudhuri *et al.*, 2003; Hadjieleftheriou and Srivastava, 2010; Wang *et al.*, 2011). 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. Edit distance reflects the original order of tokens and allows non-trivial alignment, making it a popular and important similarity function (Xiao *et al.*, 2008a). We focus on the study on SSJ with the edit distance constraint (also referred to as edit similarity join (ESJ)). ESJ is costly,

‡ Corresponding author
ORCID: Ji-zhou LUO, http://orcid.org/0000-0002-3302-3917

stemming from two time-consuming steps, i.e., distance verification and enumeration of all possible pairs for which distance verifications are needed. In fact, a trivial ESJ algorithm bears a prohibitive high complexity of $O(n\tau N^2)$. Thus, the prevalent ESJ algorithms (Gravano *et al.*, 2001; Sarawagi and Kirpal, 2004; Arasu *et al.*, 2006; Bayardo *et al.*, 2007; Xiao *et al.*, 2008a; Feng *et al.*, 2012) adopt mainly a filter-and-refine framework, where they first generate signatures for each string and use filters over the signatures to prune away most pairs of dissimilar strings, and then verify the distance for the remaining pairs and output the final results.

However, there exist some pitfalls in such approaches. First, they generate candidate pairs inherently by enumerating string pairs and cannot catch the dissimilarity between string subsets. Second, the signatures of each string are mainly local structures such as (positional) q-grams and prefixes. It means that they cannot catch the dissimilarity of strings from a global perspective of view. Third, as a result, they usually generate a huge amount of candidate pairs (Xiao *et al.*, 2008a).

To address these issues, in this study, we propose to take the (combined) frequencies of characters as global information of strings to target the two time-consuming steps in ESJ. Specifically, a novel partition-based algorithm is developed to use such information to enumerate a smaller candidate set in a more efficient way by partitioning the dataset into small chunks. A new filter is proposed to use such information to further reduce the size of the candidate set with low complexity. Unlike all 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 example below illustrates our key idea:

Example 1 Consider a self-join over set R (Table 1 with $\tau = 1$). (1) The alphabet is partitioned into $\Sigma_1 = \{a,g,j,l,m,p,v,y,z\}$, $\Sigma_2 = \{b,d,h,i,n,q,s,u,x\}$, and $\Sigma_3 = \{c,e,f,k,o,r,t,w\}$. The combined frequency vector of string s refers to $(f_1^{(\Sigma_1)}(s), f_2^{(\Sigma_2)}(s), f_3^{(\Sigma_3)}(s))$, where $f_i^{(\Sigma_i)}(s)$ counts the total appearances of characters of Σ_i in s; e.g., the vector of s_{10} is $(5,5,1)$. (2) The L_1-distance of the combined frequency vectors larger than 2τ implies the dissimilarity between strings (see Lemma 8 in Section 4.4); e.g., s_{10} is dissimilar with s_{15} because the L_1-distance between $(5,5,1)$ and $(4,4,2)$ is $3>2\tau$). (3) R can be partitioned into five chunks $C_1 - C_5$. Chunk distances are the estimated lower bounds of L_1-distances between strings from different chunks (for details, see Section 4). Table 2 lists the 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 being enumerated, although they may share common prefixes or grams; e.g., C_5 is pruned away for each string in C_3, although s_{10} and s_{15} share the prefix 'David'. Similarly, C_3, C_4, and C_5 can be pruned directly for C_2, etc.

Table 1 Data partitions of the motivation example

String ID	String	Chunk
s_1	Steve Wilson	C1
s_2	Enrico Macii	
s_3	Peter Bunemen	
s_4	Peter Ponelli	C2
s_5	Takeo Kanade	
s_6	Kate Michael	
s_7	Karl Kurbel	
s_8	Bart Preneel	
s_9	Daniel Thalmann	C3
s_{10}	David Sammon	
s_{11}	Marianne Winslett	C4
s_{12}	Ernesto Damiani	
s_{13}	Vladik Kreinovch	
s_{14}	Vipin Kumar	C5
s_{15}	David Maior	
s_{16}	David Manor	
s_{17}	Danny Dolev	
s_{18}	Hanan Samet	

Table 2 Chunk distances of the motivation example

Chunk pair	Distance	Chunk pair	Distance
C_1, C_2	2	C_2, C_4	3
C_1, C_3	7	C_2, C_5	4
C_1, C_4	2	C_3, C_4	4
C_1, C_5	4	C_3, C_5	3
C_2, C_3	7	C_4, C_5	3

2 Preliminaries

The inputs of ESJ algorithms are two string sets R, S, and a threshold τ. The outputs are all pairs of strings from each set such that their edit distance is not larger than τ (i.e., $\{(r,s)|\text{ed}(r,s) \leq \tau, r \in R, s \in$

$S\}$). For ease of exposition, we consider only self-join (i.e., $R = S$). Furthermore, assume that the readers are familiar with the elementary concepts, such as length $|s|$ of a string s, substring $s[i : j]$ of s (contiguous symbols from the ith position to the jth position), prefix $s[1 : i]$ of s, suffix $s[i : |s|]$ of s, (positional) q-grams (token, pos) of s (i.e., token $= s[\text{pos} : \text{pos} + q - 1]$) (Gravano *et al.*, 2001), and the inverted index of a string set R (Bayardo *et al.*, 2007; Xiao *et al.*, 2008a).

Given a string s on an alphabet Σ of size m, $\forall \alpha_i \in \Sigma$, α_i's frequency in s (denoted as $f_i(s)$) is the number of α_i's appearances in s. The frequency vector of s (denoted as $f(s)$) refers to vector $<f_1(s), f_2(s), \cdots, f_m(s)>$. Given a string set R over Σ and $\forall \alpha_j \in \Sigma$, α_j's frequency range associated with R (or simply frequency range) is defined as the integer interval $[m_j, M_j]$, where $m_j = \min_{s\in R} f_j(s)$ and $M_j = \max_{s\in R} f_j(s)$. In this study, we consider how to accelerate ESJ by splitting frequency ranges into small pieces and partitioning R into small chunks. To do this, we consider the distances between vectors. Given real vectors $\boldsymbol{u} = (u_1, u_2, \cdots, u_m)$ and $\boldsymbol{v} = (v_1, v_2, \cdots, v_m)$, the L_1-distance between $\boldsymbol{u}$ and $\boldsymbol{v}$, denoted as ($\|\boldsymbol{u} - \boldsymbol{v}\|_{L_1}$), is defined as $\sum\limits_{i=1}^{m} |u_i - v_i|$.

Definition 1 Let $\boldsymbol{u} = (u_1, u_2, \cdots, u_m)$ and $\boldsymbol{v} = (v_1, v_2, \cdots, v_m)$ be two real vectors. The positive difference between $\boldsymbol{u}$ and $\boldsymbol{v}$ (denoted as $\delta^+(\boldsymbol{u}, \boldsymbol{v})$) is defined as $\sum\limits_{i=1, u_i>v_i}^{m} (u_i - v_i)$. Similarly, the negative difference between $\boldsymbol{u}$ and $\boldsymbol{v}$ (denoted as $\delta^-(\boldsymbol{u}, \boldsymbol{v})$) is defined as $\sum\limits_{i=1, u_i<v_i}^{m} (v_i - u_i)$. The biased-difference distance between $\boldsymbol{u}$ and $\boldsymbol{v}$ (denoted as $\delta(\boldsymbol{u}, \boldsymbol{v})$) is defined as $\max(\delta^+(\boldsymbol{u}, \boldsymbol{v}), \delta^-(\boldsymbol{u}, \boldsymbol{v}))$.

Definition 1 means that: (1) $\delta^+(\boldsymbol{u}, \boldsymbol{v}) \geq 0$, $\delta^-(\boldsymbol{u}, \boldsymbol{v}) \geq 0$; (2) $\delta^+(\boldsymbol{u}, \boldsymbol{v}) = \delta^-(\boldsymbol{v}, \boldsymbol{u})$; (3) $\delta^+(\boldsymbol{u}, \boldsymbol{v}) + \delta^-(\boldsymbol{u}, \boldsymbol{v}) = \|\boldsymbol{u} - \boldsymbol{v}\|_{L_1}$. These facts result in Lemma 1.

Lemma 1 For any vectors $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$, we have: (1) $\delta(\boldsymbol{u}, \boldsymbol{v}) = 0 \iff \boldsymbol{u} = \boldsymbol{v}$; (2) $\delta(\boldsymbol{u}, \boldsymbol{v}) = \delta(\boldsymbol{v}, \boldsymbol{u})$; (3) $\delta(\boldsymbol{u}, \boldsymbol{v}) + \delta(\boldsymbol{v}, \boldsymbol{w}) \geq \delta(\boldsymbol{w}, \boldsymbol{u})$; (4) $\frac{1}{2}\|\boldsymbol{u} - \boldsymbol{v}\|_{L_1} \leq \delta(\boldsymbol{u}, \boldsymbol{v}) \leq \|\boldsymbol{u} - \boldsymbol{v}\|_{L_1}$. Thus, $\delta(\cdot, \cdot)$ is a distance function.

Proof Lemmas 1(1), 1(2), and 1(4) follow Definition 1. We prove Lemma 1(3) as follows: Assume $u_i \neq v_i$, $v_i \neq w_i$, and $w_i \neq u_i$ $(1 \leq i \leq m)$. The proof can be modified easily to deal with other cases. Notice that u_i, v_i, and w_i $(1 \leq i \leq m)$ must be one of the six cases in Table 3, where the constraints will be clarified in the next paragraph. If u_i, v_i, and w_i belong to case j, we put i into subset J_j. Notice that $J_1, J_2, \cdots, J_6$ partition the set $\{1, 2, \cdots, m\}$.

Table 3 Six cases in the proof of Lemma 1

Case	Description	Constraint
1	$u_i > v_i, v_i > w_i, w_i < u_i$	$a_1 + b_1 - c_1 = 0$
2	$u_i > v_i, v_i < w_i, w_i > u_i$	$a_2 - b_2 + c_2 = 0$
3	$u_i > v_i, v_i < w_i, w_i < u_i$	$a_3 - b_3 - c_3 = 0$
4	$u_i < v_i, v_i > w_i, w_i < u_i$	$-a_4 + b_4 - c_4 = 0$
5	$u_i < v_i, v_i < w_i, w_i > u_i$	$-a_5 - b_5 + c_5 = 0$
6	$u_i < v_i, v_i > w_i, w_i > u_i$	$-a_6 + b_6 + c_6 = 0$

Let $a_j = \sum_{i\in J_j} |u_i - v_i|$, $b_j = \sum_{i\in J_j} |v_i - w_i|$, and $c_j = \sum_{i\in J_j} |w_i - u_i|$ $(1 \leq j \leq 6)$. The constraints above can be verified easily. Moreover,

$$\begin{cases} \delta^+(\boldsymbol{u}, \boldsymbol{v}) = a_1 + a_2 + a_3, \\ \delta^-(\boldsymbol{u}, \boldsymbol{v}) = a_4 + a_5 + a_6, \end{cases}$$

$$\begin{cases} \delta^+(\boldsymbol{v}, \boldsymbol{w}) = b_1 + b_4 + b_6, \\ \delta^-(\boldsymbol{v}, \boldsymbol{w}) = b_2 + b_3 + b_5, \end{cases}$$

$$\begin{cases} \delta^+(\boldsymbol{w}, \boldsymbol{u}) = c_2 + c_5 + c_6, \\ \delta^-(\boldsymbol{w}, \boldsymbol{u}) = c_1 + c_3 + c_4. \end{cases}$$

Putting these constraints together, we obtain

$$\begin{aligned} &\delta^+(\boldsymbol{u}, \boldsymbol{v}) - \delta^-(\boldsymbol{u}, \boldsymbol{v}) + \delta^+(\boldsymbol{v}, \boldsymbol{w}) - \delta^-(\boldsymbol{v}, \boldsymbol{w}) \\ &+ \delta^+(\boldsymbol{w}, \boldsymbol{u}) - \delta^-(\boldsymbol{w}, \boldsymbol{u}) = 0. \end{aligned} \tag{1}$$

Let the indicator of $\delta(\boldsymbol{u}, \boldsymbol{v})$ be '+' if $\delta(\boldsymbol{u}, \boldsymbol{v}) = \delta^+(\boldsymbol{u}, \boldsymbol{v})$, or '−' if $\delta(\boldsymbol{u}, \boldsymbol{v}) = \delta^-(\boldsymbol{u}, \boldsymbol{v})$. The concatenation of indicators of $\delta(\boldsymbol{u}, \boldsymbol{v})$, $\delta(\boldsymbol{v}, \boldsymbol{w})$, and $\delta(\boldsymbol{w}, \boldsymbol{u})$ is called the 'schema' of $(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w})$. Eq. (1) asserts that neither '$+ + +$' nor '$- - -$' is a schema unless $\delta^+(\boldsymbol{u}, \boldsymbol{v}) = \delta^-(\boldsymbol{u}, \boldsymbol{v})$, $\delta^+(\boldsymbol{v}, \boldsymbol{w}) = \delta^-(\boldsymbol{v}, \boldsymbol{w})$, and $\delta^+(\boldsymbol{w}, \boldsymbol{u}) = \delta^-(\boldsymbol{w}, \boldsymbol{u})$, where the conclusion holds obviously. Next, for each of $(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w})$'s other six schemas, we show $\delta(\boldsymbol{u}, \boldsymbol{v}) + \delta(\boldsymbol{v}, \boldsymbol{w}) - \delta(\boldsymbol{w}, \boldsymbol{u}) \geq 0$.

For schema '$+ + -$', we have

$$\begin{aligned} &\delta(\boldsymbol{u}, \boldsymbol{v}) + \delta(\boldsymbol{v}, \boldsymbol{w}) - \delta(\boldsymbol{w}, \boldsymbol{u}) \\ &= \delta^+(\boldsymbol{u}, \boldsymbol{v}) + \delta^+(\boldsymbol{v}, \boldsymbol{w}) - \delta^-(\boldsymbol{w}, \boldsymbol{u}) \\ &= (a_1 + a_2 + a_3) + (b_1 + b_4 + b_6) - (c_1 + c_3 + c_4) \\ &= 0 + a_2 + b_3 + b_6 + a_4 \\ &\geq 0. \end{aligned}$$

Schemas '$+ - +$' and '$- + +$' can be verified similarly. Moreover, schemas '$- - +$', '$+ - -$', and '$- + -$' can be transformed into '$+ + -$', '$- + +$', and '$+ - +$', respectively, as $\delta^+(\boldsymbol{u}, \boldsymbol{v}) = \delta^-(\boldsymbol{v}, \boldsymbol{u})$.

3 FreFilter and its independence

Characters' frequencies in strings catch the similarity of strings and can be used to help ESJ, as shown in the following two lemmas. Lemma 2 was proved and used in Xiao *et al.* (2008a). Lemma 3 suggests a more efficient way to use the frequencies of characters.

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

Lemma 3 For two strings x and y, $\delta(f(x), f(y)) > \tau$ implies $\text{ed}(x, y) > \tau$.

Proof Note that applying any edit operation (insertion, deletion, or substitution of a single character) on x or y changes $\delta(f(x), f(y))$ by at most 1. Lemma 1 shows that it needs at least $\delta(f(x), f(y))$ edit operations to transform x to y.

Lemma 3 is stronger than Lemma 2. In fact, Lemma 1(4) asserts that all pairs filtered away by Lemma 2 can also be filtered away by Lemma 3, but not vice versa. For example, let $\tau = 2$, and consider strings s=new and t=event. Note that $\text{ed}(s, t) > \tau$. It is easy to check $\delta(s, t) = 3 > \tau$ but $\|f(s) - f(t)\|_{L_1} = 4 \leq 2\tau$; i.e., Lemma 3 prunes away (s, t) but Lemma 2 does not.

Lemma 4 For strings x and y, if $|x| \geq |y|$ then $\delta(f(x), f(y)) = \delta^+(f(x), f(y))$.

The proof of Lemma 4 follows the fact that $|x| - |y| = \delta^+(f(x), f(y)) - \delta^-(f(x), f(y)) \geq 0$. It simplifies the computation of $\delta(f(x), f(y))$ if $|x| > |y|$ is known. In fact, typical ESJ algorithms such as Ed-Join (Xiao *et al.*, 2008a) and PassJoin (Li *et al.*, 2011) usually sort all strings by lengths, making Lemma 4 be applied directly.

Lemmas 1 and 3, which have been proved here, provide us a new filter named FreFilter. For string pair (x, y), FreFilter first exchanges the roles of x and y, if necessary, to guarantee that $|x| \geq |y|$. Then, it obtains frequency vectors $f(x)$ and $f(y)$, and applies Lemma 4 to compute $\delta(f(x), f(y))$. If $\delta(f(x), f(y)) > \tau$, FreFilter returns 'false' to indicate $\text{ed}(x, y) > \tau$; else, it returns 'true'.

The time complexity of FreFilter is obviously $O(n + |\Sigma|)$. If frequency vectors are pre-computed (by the same way as q-grams being pre-extracted (Xiao *et al.*, 2008a)), its complexity becomes $O(|\Sigma|)$, which can be viewed as a constant.

Independence of FreFilter: Since distance verification is time-consuming in ESJ, several filters have been proposed to identify string pairs (s, t) with $\text{ed}(s, t) \leq \tau$. We show that FreFilter is independent of them via settings of Example 2.

Example 2 Set $\tau = 5$. Consider strings s_0, t_0 below. s_0='Petra Perner Case based reasoning for image interpretation' and t_0='Petra Perner Cbr based ultrasonic image interpretation'. The frequencies of 'a', 'e', 'f', 'o', and 's' in s_0 are higher than their frequencies in t_0. Lemma 4 tells us $\delta(f(s_0), f(t_0)) = 6 > \tau$. Thus, s_0 is dissimilar to t_0 according to FreFilter.

However, string pair (s_0, t_0) survives existing filters, including the PassJoin filter, length filter, count filter, position filter, prefix filter, suffix filter, content-based mismatching filter, and content filter Ed-Join (Xiao *et al.*, 2008a). The details of these filters can be found in Xiao *et al.* (2008a) and Li *et al.* (2011). As an example, we show that (s_0, t_0) survives the PassJoin filter and omit the discussion for other filters. The PassJoin filter mandates that if $\text{ed}(s, t) \leq \tau$, and t is partitioned into $\tau + 1$ consecutive segments of an approximately equal length, then one of these segments must be a consecutive substring of s. For s_0 and t_0, the first segment obtained by partitioning t_0 is 'Petra Per'. It is also a segment of s_0. Thus, (s_0, t_0) survives the PassJoin filter.

Our experiments show that, a noticeable proportion of candidate pairs surviving the existing filters can be further pruned away by FreFilter, and vice visa. Since FreFilter is independent of the existing filters, it is expected that the performance of ESJ algorithms can be improved greatly by integrating FreFilter into the existing algorithms. The limited filtering effectiveness of the existing filters stems from the fact that they use only the local information of strings such as positional q-grams and consecutive substrings, and ignore the global information of strings provided by statistics such as frequency vectors. More statistics are used to avoid enumerating string pairs in Sections 4 and 5.

4 Data partition via frequency vectors

4.1 Overview of data partition

FreFilter captures the dissimilarity between strings by exploiting the fact that the biased-difference distance (bd-distance) between the frequency vectors of similar strings must be small. It also means that different strings with the same frequency vector cannot be filtered by FreFilter, but the bd-distance computation between them should be avoided. With this comes immediately a trivial data partitioning strategy, i.e., to group all strings with the 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 a small alphabet and frequency ranges. As a result, the total computation saved is also small.

To address this issue, we propose to split the frequency ranges of characters into small intervals. Strings with their frequency vectors falling into a same group of intervals are grouped into a data chunk. The number of data chunks grows exponentially with the number of split intervals and the number of 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 of which can be adjusted according to τ and the size of datasets.

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

Data partition: Assume that $\alpha_1, \alpha_2, \cdots, \alpha_\theta$ are partition characters, and each α_j's frequency range $[m_j, M_j]$ is 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 $[m_j, P_j[1]]$ and $[P_j[k_j-1], M_j]$, have an equal length l_j.

To partition string set R, FrePartition (Algorithm 1) processes each string s sequentially (lines 2–15). For each string s, Algorithm 1 first computes id $= (v_1, v_2, \cdots, v_\theta)$ for s, such that frequency $f_j(s)$ falls into α_j's v_jth interval (lines 3–8). id is taken as the identifier of a data chunk. Then, Algorithm 1 checks whether there is an existing data chunk R_k such that R_k.id = id (line 10). If yes, it puts s into R_k (line 11). Otherwise, it creates a new data chunk, sets id as its identifier, and puts s into it (line 13). Finally, all generated data chunks are returned (line 16). FrePartition (Algorithm 1) runs in time of $O(N)$, given necessary information. In fact, all ids of data chunks can be encoded as non-negative integers, and be used as an array index, which makes FrePartition scan the dataset once.

Section 4.2 is aimed to split the frequency range. In Section 4.3 the chunk distance is defined and a greedy strategy is presented for character choosing. In Section 4.4 we develop another strategy to enhance the pruning effectiveness of data partitioning.

Algorithm 1 FrePartition(R)

```
Input: string set $R$; characters $\alpha_1, \alpha_2, \cdots, \alpha_\theta$.
Output: a partition of $R$.
1: $p \leftarrow 0$;
2: for each $s \in R$ do
3:     for $j = 1$ to $\theta$ do
4:         if $f_j(s) \le P_j[1]$ then $v_j = 1$;
5:         else if $f_j(s) > P_j[k_j - 1]$ then $v_j = k_j$;
6:         else $v_j = \lceil (f_j(s) - P_j[1])/l_j \rceil + 1$;
7:         end if
8:     end for
9:     id $\leftarrow (v_1, v_2, \cdots, v_\theta)$;
10:    if $R_k$.id = id ($\exists k \in [1, p]$ then
11:        $R_k \leftarrow R_k \cup \{s\}$;
12:    else
13:        $p \leftarrow p + 1$; $R_p \leftarrow \{s\}$; $R_p$.id $\leftarrow$ id;
14:    end if
15: end for
16: return $R_1, R_2, \cdots, R_p$.
```

4.2 Range split

Let R be the string set, N be the number of strings in R, $\alpha_j \in \Sigma$, and $[m_j, M_j]$ 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 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 R can be calculated as $\mu_j = \sum\limits_{i=m_j}^{M_j} i \cdot (h_j[i]/N)$, and α_j's frequency deviation is $\sigma_j^2 = \sum\limits_{i=m_j}^{M_j} \left[(i-\mu_j)^2 \cdot (h_j[i]/N)\right]$.

To split $[m_j, M_j]$ into small intervals with equal length, one direct method is to view $h_j[m_j : M_j]$ as a usual histogram and use optimal interval length $l_j \approx 3.49(M_j - m_j)^{-1/3}\sigma_j$ given in Scott (1979). However, like many other methods of the same kind, this optimal interval length is designed to produce another histogram such that the expected error between the new histogram and the original one is minimized. Applying this optimal length directly will lead to some very small data chunks, and the number of intervals will also lose 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 has $f_j(s)$ fall into the first interval and the last interval. The try-and-refine method aims 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 l_j and tries again.

RangeSplit (Algorithm 2) splits 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 $m_j \le i \le M_j$ (lines 1–4). Then, it invokes IntervalSplit to split $[m_j, M_j]$ into intervals of initial length $3.49(M_j - m_j)^{-1/3}\sigma_j$ (lines 5–6). Interval number k and an array $P[0 : k]$ of partition points are returned. If $|k-\kappa|$ is too large, then l_j is increased or decreased by one accordingly (lines 7–8). Then, it tries to split with the updated l_j repeatedly until k equals κ approximately (lines 10–13) or the new try makes things worse (line 14). The final l_j, k_j, and $P[0 : k_j]$ are returned.

Algorithm 2 RangeSplit(α_j)

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

IntervalSplit (Algorithm 3) splits range $[m_j, M_j]$ into intervals with a given length l_j and returns interval number k and the array $P[0 : k]$ of partition points. The tailing strategy, is implemented as 'while' conditions in lines 2 and 6.

4.3 Character selection and chunk distance

Given $\alpha_j \in \Sigma$ and 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 2. In what follows, l_j, k_j, and

Algorithm 3 IntervalSplit($H[0 : M_j], l_j$)

```
1: $k \leftarrow 1$; $i \leftarrow m_j$; $P[0] \leftarrow m_j$;
2: while $H_j[i] < \frac{1}{2}(H_j[i + l_j] - H_j[i])$ do
3:     $i \leftarrow i + 1$;
4: end while
5: $P[k] \leftarrow i$; $k \leftarrow k + 1$;
6: while $H_j[i+l_j] - H_j[i] < 2(H_j[M_j] - H_j[i+l_j])$ do
7:     $i \leftarrow i + l_j$; $P[k] \leftarrow i$; $k \leftarrow k + 1$;
8: end while
9: $P[k] \leftarrow M_j$;
10: return $k$ and $P[0 : k]$.
```

interval $[P_j[i-1]+1, P_j[i]]$ will be called "α_j's interval length", "α_j's interval number", and "α_j's ith interval", respectively, without any ambiguity.

Definition 2 The split distance between α_j's i_1th and i_2th intervals ($1 \le i_1, i_2 \le k_j$) is defined as $\delta_j(i_1, i_2) = \max(0, (|i_1 - i_2| - 1) \cdot l_j)$.

Lemma 5 $E(\delta_j(i_1, i_2)) = \dfrac{(k_j-1)(k_j-2)}{3k_j} l_j$, if i_1 and i_2 is chosen from $[1, k_j]$ randomly and uniformly.

Proof

$$\begin{cases} P_r(\delta_j(i_1,i_2) < l_j) = \dfrac{1}{k_j} + \displaystyle\sum_{j=1}^{k_j-1} \dfrac{2}{k_j^2} \\ \qquad\qquad\qquad\quad = \dfrac{k_j + 2(k_j-1)}{k_j^2}, \\ P_r(\delta_j(i_1,i_2) = kl_j) = \displaystyle\sum_{j=1}^{k_j-k} \dfrac{2}{k_j^2} = \dfrac{2(k_j-k)}{k_j^2}, \end{cases}$$
$$1 \le k \le k_j - 2.$$

Therefore,

$$E(\delta_j(i_1,i_2)) = \sum_{k=0}^{k_j-2} kl_j \cdot P_r\left(\delta_j(i_1,i_2) = kl_j\right) = \frac{(k_j-1)(k_j-2)}{3k_j} l_j.$$

Definition 3 The pruning ability of α_j is defined as $\dfrac{(k_j-1)(k_j-2)}{3k_j} l_j$.

Greedy-selected-character algorithm: sort characters of Σ in descending order of their pruning ability as $\alpha_1, \alpha_2, \cdots, \alpha_m$. Return $\alpha_1, \alpha_2, \cdots, \alpha_\theta$ as well as their interval lengths, interval numbers, and division point arrays.

Chunk distance: Now, let $\alpha_1, \alpha_2, \cdots, \alpha_\theta$ be selected as partition characters, and $R_1, R_2, \cdots, R_p$ be data chunks returned by FrePartition. We consider the lower bound of the edit distance between any pair of strings from two data chunks.

Consider data chunks R_i and R_j, with $R_i.\text{id} = (v_{i1}, v_{i2}, \cdots, v_{i\theta})$ and $R_j.\text{id} = (v_{j1}, v_{j2}, \cdots, v_{j\theta})$, respectively. For any $s \in R_i$ and $t \in R_j$, each α_k $(1 \leq k \leq \theta)$ 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 $\alpha_1, \alpha_2, \cdots, \alpha_\theta$ gives a lower bound of $\|f(s) - f(t)\|_{L_1}$.

Definition 4 Chunk distance $\text{dis}(R_i, R_j)$ between data chunks R_i and R_j, with $R_i.\text{id} = (v_{i1}, v_{i2}, \cdots, v_{i\theta})$ and $R_j.\text{id} = (v_{j1}, v_{j2}, \cdots, v_{j\theta})$, is defined as $\sum\limits_{k=1}^{\theta} \max(0, (|v_{ik} - v_{jk}| - 1) \cdot l_k)$.

Lemma 6 For data chunks R_i, R_j and $s \in R_i, t \in R_j$, $\text{dis}(R_i, R_j) > 2\tau$ implies $\text{ed}(s, t) > \tau$.

Lemma 7 If R_i and R_j are data chunks drawn from $R_1, R_2, \cdots, R_p$ randomly and uniformly, then $E(\text{dis}(R_i, R_j)) > \sum\limits_{j=1}^{\theta} \dfrac{(k_j - 1)(k_j - 2)}{3k_j} l_j$.

Discussion:

1. Note that $E(\text{dis}(R_i, R_j)) \gg 2\tau$ means that many chunk-pairs with chunk distances are larger than 2τ and all string pairs from such chunk pairs are dissimilar. $E(\text{dis}(R_i, R_j))$ is determined by θ and κ. We set $\kappa = \lfloor \log_2 \text{avgLen} \rfloor$ and θ to be the solution of $\sum\limits_{j=1}^{\theta} \dfrac{(k_j - 1)(k_j - 2)}{3k_j} l_j = 2\tau$, where avgLen is the average length of strings in R.

For example, if avgLen=100, then $\kappa = 6$. Further, assume $l_j = 2$ and $k_j = \kappa$. Then $\theta = 5$ enables $E(\text{dis}(R_i, R_j)) > 11$, which is large enough for $\tau = 1 - 4$. As another example, assume $l_j = 4$ and $k_j = \kappa = 6$. Then $\theta = 3$ enables $E(\text{dis}(R_i, R_j)) > 13$, which is large enough for $\tau = 1 - 5$. In Section 4.4 we will propose a novel technique to compute a small θ for large τ by amplifying l_j.

2. l_j is determined by σ_j^2 to a large extent. On the one hand, a larger σ_j^2 means a wider frequency range $[m_j, M_j]$. On the other hand, the initial value of l_j is $3.49(M_j - m_j)^{-1/3}\sigma_j$.

3. 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 a very small size are generated. In the other aspect, if the data partition is used to guarantee the balanced sizes of data chunks, then many similar strings will fall into different data chunks. In contrast to this, our method can find most of similar strings by joining all data chunks with themselves.

4. To join two different datasets R and S, a common data partition strategy should be applied on them. Such a strategy can be obtained by redefining $h_j[i]$ (Algorithm 2) to be the total number of $R \cup S$'s strings, in each of which α_j appears exactly i times. Then, R and S can be partitioned into data chunks by Algorithm 1.

4.4 Z-folding combination of characters

Let $\Sigma_1, \Sigma_2, \cdots, \Sigma_\theta$ be a partition of alphabet Σ, and s be a string of R. Each Σ_i is called 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 number 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 $< f_1^{(\Sigma_1)}(s), f_2^{(\Sigma_2)}(s), \cdots, f_\theta^{(\Sigma_\theta)}(s) >$ is referred to as the combined frequency vector of s associated with the partition and written as $f^{(c)}(s)$. With the aid of average frequency μ_j and deviation σ_j^2 of each character α_j, 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 deviation $\sigma_{\Sigma_i}^2$ of Σ_i can be computed as $\sigma_{\Sigma_i}^2 = \sum_{\alpha_j \in \Sigma_i} \sigma_j^2$.

Given a partition $\Sigma_1, \Sigma_2, \cdots, \Sigma_\theta$ of Σ and the statistics of each combined character Σ_i $(1 \leq i \leq \theta)$, RangeSplit (Algorithm 2) can treat each Σ_i as a usual character and split its frequency range into small intervals. Thus, the interval length, interval number, and vth interval of Σ_i are determined similarly. According to $|f_i^{(\Sigma_i)}(s_1) - f_i^{(\Sigma_i)}(s_2)| \leq \sum_{\alpha_j \in \Sigma_i} |f_j(s_1) - f_j(s_2)|$, we can obtain Lemma 8, which states the utility of combined characters:

Lemma 8 For strings s_1 and s_2, if $\|f^{(c)}(s_1) - f^{(c)}(s_2)\|_{L_1} > 2\tau$, then $\text{ed}(s_1, s_2) > \tau$.

Lemma 9 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 $l_j = 3.49((\kappa - 2)l_j)^{-1/3}\sigma_j$ approximately and iteratively.

Now, the remaining issue is how to break up alphabet Σ into combined characters $\Sigma_1, \Sigma_2, \cdots, \Sigma_\theta$ such that the expectation of chunk distances is maximized. This means that $\sum\limits_{i=1}^{\theta} l_i$ reaches the maximum value, according to Lemma 7 and $k_i \approx \kappa$. Moreover, l_i increases with the increase of σ_{Σ_i}, and $\sum\limits_{i=1}^{\theta} \sigma_{\Sigma_i}^2 = \sigma_\Sigma^2$ is a constant. This suggests that the partition of Σ should make all $\sigma_{\Sigma_i}^2$ $(1 \leq i \leq \theta)$ approximately equal. Here comes the following

Z-folding algorithm:

Z-folding-combined-character algorithm: Sort characters of $\varSigma$ in descending order of their standard deviations as $\alpha_1, \alpha_2, \cdots, \alpha_m$. For k from 0 to $\lfloor m/\theta \rfloor$, allot $\alpha_{k\theta+1}, \alpha_{k\theta+2}, \cdots, \alpha_{k\theta+\theta}$ to $\varSigma_1, \varSigma_2, \cdots, \varSigma_\theta$ as follows: if k is even, assign $\alpha_{k\theta+j}$ to $\varSigma_j$; if k is odd, assign $\alpha_{k\theta+j}$ to $\varSigma_{\theta-j+1}$. Return $\varSigma_1, \varSigma_1, \cdots, \varSigma_\theta$ as well as their interval lengths, interval numbers, and partition point arrays.

Example 3 Let $\theta = 3$ and $\kappa = 6$. Consider string set R in Table 1. (1) After characters are sorted in descending order of their deviations, the Z-folding-combined-character algorithm breaks up the alphabet into subsets $\varSigma_1$, $\varSigma_2$, and $\varSigma_3$ (Example 1). (2) Then, the combined frequency vector of each string is calculated from the frequency vector. Meanwhile, the frequency range, as well as the deviation, of each combined character is figured out; e.g., $f^{(c)}(s_{10}) = (5, 5, 1)$ and $f^{(c)}(s_{14}) = (4, 4, 2)$. The frequency ranges of $\varSigma_1$, $\varSigma_2$, and $\varSigma_3$ are $[1, 8]$, $[1, 7]$, and $[0, 8]$, respectively. (3) RangeSplit splits (a) $[1, 8]$ into $[0, 2]$, $[3, 4]$, $[5, 6]$, and $[7, 8]$, (b) $[1, 7]$ into $[1, 2]$, $[3, 4]$, $[5, 6]$, and $[7, 7]$, and (c) $[0, 7]$ into $[0, 1]$, $[2, 3]$, $[4, 5]$, and $[6, 7]$. Thus, interval numbers $k_1 = k_2 = k_3 = 4$ and interval lengths $l_1 = l_2 = l_3 = 2$. (4) Using results of Example 3(3), FrePartition partitions R into five chunks $C_1, C_2, \cdots, C_5$ with $(0, 1, 2)$, $(1, 0, 2)$, $(2, 2, 0)$, $(1, 2, 2)$, and $(1, 1, 1)$ as chunk ids, respectively. (5) The chunk distances can be evaluated; e.g., $\mathrm{dis}(C_2, C_3) = 7$.

5 Partition-based edit similarity join algorithm

5.1 Chunk filtering

Given the partitioning strategy and data chunks $R_1, R_2, \cdots, R_p$ generated by Algorithm 1, the chunk distance between any two chunks can be computed according to Definition 4. Furthermore, Lemma 6 states that the relationships between chunk distances and 2τ's can be used to filter away some candidate pairs. Such relationships can be recorded and taken as a new filter, ChunkFilter. Formally, ChunkFilter is a $p \times p$ matrix $\boldsymbol{M}$, where $M(i, j) = 0$ or 1 $(1 \leq i, j \leq p)$. $M(i, j) = 0$ means that $\mathrm{dis}(R_i, R_j) \leq 2\tau$, and $M(i, j) = 1$ means that $\mathrm{dis}(R_i, R_j) > 2\tau$.

ChunkFilter can be constructed easily. We enumerate all $p(p-1)/2$ chunk pairs (R_i, R_j)'s, check whether $\mathrm{dis}(R_i, R_j) \leq 2\tau$ or not, and set entry $M(i, j)$ accordingly. The algorithm is omitted here for simplicity. Note that the dissimilarity caught by ChunkFilter can also be caught by FreFilter, since Lemma 3 is stronger than Lemma 2. However, using ChunkFilter to find the dissimilarity between two strings (with their chunk ids given) costs only time of $O(1)$, while using FreFilter to do the same thing costs time of $O(|\varSigma|)$. This suggests that ChunkFilter can be used to save some computation caused by FreFilter, if ChunkFilter is applied before FreFilter.

5.2 FrepJoin algorithm

The FrepJoin algorithm integrates FreFilter and ChunkFilter into the filter-and-refine framework, and makes it possible to remarkably improve the performance of any ESJ algorithm $\mathcal{A}$ under this framework with small extra costs. The key idea is to exploit two light-weighted filters to reduce the number of candidate pairs generated by $\mathcal{A}$. The candidate pairs pruned away by the new filters result in remarkable savings on edit distance verification.

FrepJoin (Algorithm 4) runs in three stages: In the first stage (lines 1–8), the data set is constructed and ChunkFilter is partitioned. In the second stage (lines 9–12) an existing algorithm $\mathcal{A}$ is used to generate a candidate set for each string. The third stage (lines 13–18) is to use ChunkFilter and FreFilter to reduce the candidate set further and verify the remaining candidate pairs. It is not hard to understand the algorithm with the concepts built in Section 4. The explanation of each line is omitted.

Analysis: In contrast to the selected ESJ algorithm $\mathcal{A}$, the main extra costs of our algorithm occur in the first stage. This is because the second stage executes the same operations as in algorithm $\mathcal{A}$, and the third stage increases only two filters. For ChunkFilter, each application costs time of $O(1)$. For FreFilter, each application costs time of $O(|\varSigma|)$, when the frequency vectors are recorded. These two costs are small, compared with the saved computation of edit distance verification, which costs time of $O(n\tau)$.

Now, let us focus on the total cost of the first stage. Line 1 costs time of $O(nN)$, since it scans the dataset twice. Line 2 costs time of $O(\theta)$, according to the Z-folding algorithm. Line 3 costs time of $O(|\varSigma|N)$, since it scans the frequency vector of each string twice. Lines 4–5 cost time of $O(n\theta)$ in total, since there are θ combined characters

Algorithm 4 FrepJoin(R, τ)

```
Input: string set $R$; edit distance upper bound $\tau$.
Output: pair set $S$ of strings in $R$ with $\mathrm{ed}(\cdot,\cdot) \leq \tau$.
 1: Scan $R$ to obtain statistics for each character of $\Sigma$;
 2: Generate combined characters $\Sigma_1, \Sigma_2, \cdots, \Sigma_\theta$;
 3: Scan frequency vectors to obtain statistics of the
      combined characters;
 4: for each $\Sigma_j$ do
 5:     RangeSplit($\Sigma_j$);
 6: end for
 7: $R_1, R_2, \cdots, R_p \leftarrow$ FrePartition($R$);
 8: $M_{p\times p} \leftarrow$ ChunkFilter($R_1, R_2, \cdots, R_p$);
 9: $S \leftarrow$ ptyset;
10: $I \leftarrow$ buildIndex($R$);
11: for each $x \in R$ do
12:     Use $I$ and existing filters to obtain a candidate
          set $A$ for $x$;
13:     if $M(x.\mathrm{cid}, y.\mathrm{cid}) = 0$ or FreFilter($x, y$) = 1 then
14:         continue;
15:     end if
16:     if Verify($x, y$) = true then
17:         Add pair $< x, y >$ into $S$;
18:     end if
19: end for
20: return $S$.
```

and the expense of algorithm RangeSplit($\cdot$) is $O(n^2)$. Line 7 costs time of $O(nN)$, since it scans the dataset once. Line 8 costs time of $O(\theta\kappa^{2\theta})$, according to Section 5.1. Note that $n \ll N$ and parameters (κ, θ) can be viewed as small constants. Thus, the total cost of the first stage is $O(nN)$, which is caused mainly by scanning the dataset and the frequency vectors.

6 Experimental evaluation

We compared the following algorithms: (1) Ed-Join, which integrates the existing filters (Xiao *et al.*, 2008a), (2) Ed-Join+FF, which is the Ed-Join algorithm with FreFilter added after all existing filters, (3) FF+Ed-Join, which is the Ed-Join algorithm with FreFilter added after the Prefix Filter and Length Filter, and (4) PassJoin (Li *et al.*, 2011). Frep+$\mathcal{A}$ is Algorithm 4, where $\mathcal{A}$ is PassJoin or Ed-Join. All algorithms were implemented as in-memory algorithms with C++, with all their inputs loaded into the memory before run. Moreover, the alphabet is the English alphabet plus a wild card. All symbols not in the English alphabet were mapped to the wild card.

All experiments were performed on an IBM x3650 M3 system with Intel® Xeon 2.67 GHz 4-core CPU and 8 GB RAM. The operating system is scientific Linux. The algorithms were compiled using g++ 4.4.4 with -O3 flag.

We used three public-available real datasets to evaluate our methods. They are DBLP, TREC, and AOL Query Log. They were chosen to cover strings of different average lengths and different contents. Detailed statistics of these datasets can be found in Xiao *et al.* (2008a) and Li *et al.* (2011). These datasets were transformed and cleaned as in Xiao *et al.* (2008a). The frequencies of single characters in each string were counted at the time of tokenizing q-grams. Thus, $27N \times 10^{-6}$ Mb space is needed for storing the frequency vectors in experiments, where N is the number of strings in the dataset.

q affects the performance of Ed-Join, and thus We have Ed-JoinFF and FFEd-Join. Here we set $q = 5$ to guarantee the performance of Ed-Join on the chosen datasets, according to the experimental results in Section 6.2 of Xiao *et al.* (2008a). κ was set to be $\lfloor \log_2 \mathrm{avgLen} \rfloor$. Thus, different datasets had different κ's. For θ, we find $\theta \leq 3$ holds for all cases of our experiments, according to the method in Section 4.3. For simplicity of statement, we fixed $\theta = 3$ here.

Three sets of experiments were conducted to evaluate: (1) the independence of FreFilter, by comparing the filtering effectiveness between algorithms 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 the candidate set, and (3) the efficiency of FrepJoin for ESJ, compared with those of Ed-Join and PassJoin. The results are reported in Figs. 1–3.

Fig. 1 compares the filtering effectiveness (i.e., the ratio of the number of remaining string pairs to the number of enumerated string pairs) of filters in different settings. We find that, on all three datasets, many pairs surviving the existing filters can be further pruned away by FreFilter, and vice visa. Thus, our FreFilter is independent of the existing filters.

Fig. 2 evaluates the performances of data partitioning on DBLP with algorithm Frep+EdJoin. Fig. 2a compares the effectiveness of filtering data-chunk pairs by setting different values for θ, and the ratios of chunk pairs with chunk distances larger than 2τ to all chunk pairs are reported. Fig. 2b compares the effects of combined characters on the running time of the join algorithms. Fig. 2c compares the numbers of indexed entries in the global index, and the total numbers of index entries in the local indices are counted for different q's, θ's, and τ's.

Fig. 3 compares the running times of algorithms on different datasets. Frep+EdJoin (Frep+PassJoin resp.) consistently outperforms Ed-Join (PassJoin resp.) when $\tau > 3$, and this becomes more evident when τ and q increase. For a small τ, the former is slightly slower than the latter. This is because: (1) For any τ, our partition-based method can enumerate smaller candidate sets in a more efficient way, and the frequency filter can further reduce the candidate pairs whose edit distances need to be checked. (2) However, for a small τ, edit distance verification may be faster than our frequency filter.

7 Related work

Many studies on the ESJ algorithms, e.g., PartEnum (Arasu *et al.*, 2006), AllPairs (Bayardo *et al.*, 2007), SSJoin (Chaudhuri *et al.*,

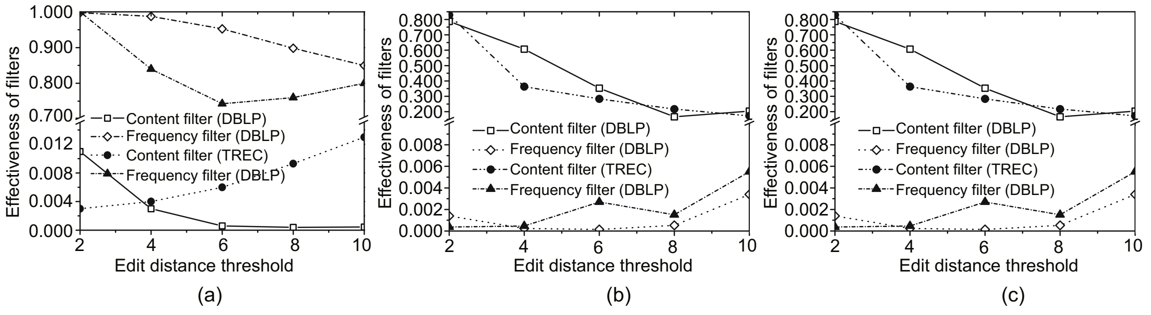

Fig. 1 Independences of frequency filters Ed-Join+FF (a), FF+Ed-Join (b), and Ed-Join(c), with $q = 5$ and $\theta = 3$

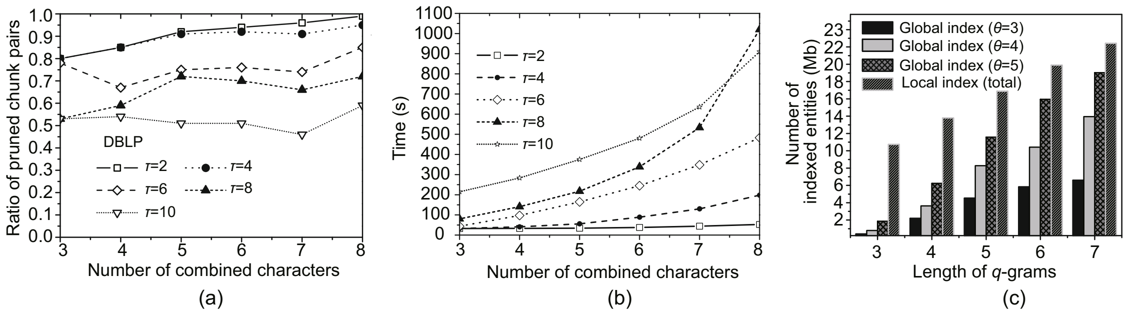

Fig. 2 Performance evaluation of data partitioning: (a) filtering effectiveness in DBLP; (b) time vs. number of characters in DBLP with $q = 5$; (c) number of indexed entities with $\tau = 6$

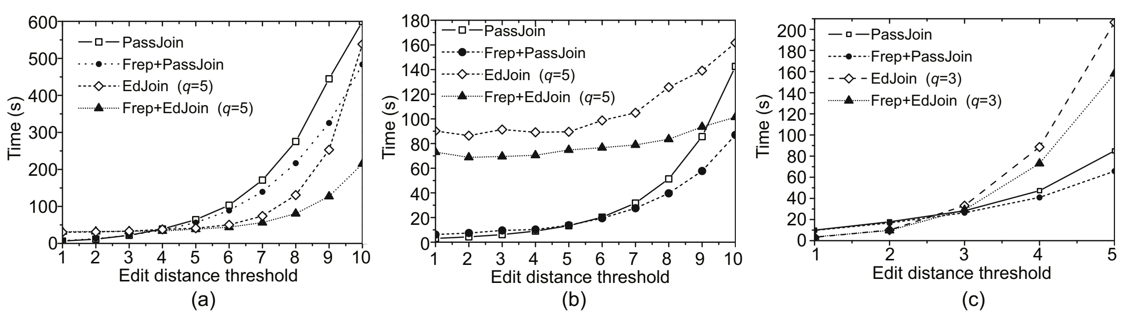

Fig. 3 Performance evaluation of FrepJoin in databases DBLP (a), TREC (b), and AOL Query Log (c) compared with those of Ed-Join and PassJoin

2006b), Ed-Join (Xiao *et al.*, 2008a), TrieJoin (Feng *et al.*, 2012), PassJoin (Li *et al.*, 2011), and VcChunkJoin (Qin *et al.*, 2013), employ the filter-and-refine framework, where algorithms usually use q-grams to filter out most dissimilar pairs and further verify the remaining pairs. Several filters have been developed, e.g., the counting filter (Gravano *et al.*, 2001; Chaudhuri *et al.*, 2006b), length filter (Gravano *et al.*, 2001), positional filter (Gravano *et al.*, 2001; Xiao *et al.*, 2008b), prefix filter (Arasu *et al.*, 2006; Chaudhuri *et al.*, 2006b; Xiao *et al.*, 2008b), suffix filter (Xiao *et al.*, 2008b), non-overlapping segment based filters (Qin *et al.*, 2013), and content-based mismatching filter (Xiao *et al.*, 2008a). Although our frequency filter is a special form of the content-based filter, it is not adopted explicitly before and can prune away many candidate pairs that survive the existing ones.

Partition techniques are not new for ESJ. PartEnum (Arasu *et al.*, 2006) generates signatures for each string by projecting its feature vector to randomly selected two-level partitioned q-gram sets. It needs many candidate pairs to guarantee completeness. The suffix filter (Xiao *et al.*, 2008b) adopts a divide-and-conquer framework to perform set-similarity join. Both PartEnum and the suffix filter are still string-pair based and cannot catch the dissimilarity between subsets. Trie-Join (Feng *et al.*, 2012) takes a trie-structure as an index to group together strings with the same prefixes. Its subtrie pruning rule is able to prune away the whole groups of strings. As reported in Feng *et al.* (2012), it works well for only short strings. In Sarawagi and Kirpal (2004), a string set was partitioned into clusters of partially overlapping strings to reduce the index size, and thus the performance of weighted intersection similarity join is improved. Although it can be adapted to ESJ, its performance is not guaranteed. Vernica *et al.* (2010) used token (groups) to generate data partition for each computing node of MapReduce. Afrati *et al.* (2012) used the ball-hashing method to generate data partition for each computing node of MapReduce. Unlike our method, both methods in Vernica *et al.* (2010) and Afrati *et al.* (2012) may allot a string into several data subsets. This is beneficial only to parallel computing. Moreover, Sarawagi and Kirpal (2004), Vernica *et al.* (2010), and Afrati *et al.* (2012) did not use global information of strings and could not provide lower bounds of edit distances. It is also popular to partition strings into a group of string-segments, and to transform approximate string matching into an exact search (Navarro and Salmela, 2009; Wang *et al.*, 2009; Ge and Li, 2011). Besides, it was proposed in Li *et al.* (2008) to partition long inverted lists into shorter ones and to skip irrelevant ones while processing approximate string matching. Our method intends to partition the string set into small chunks with performance guaranteed, and is orthogonal to those methods mentioned above.

8 Conclusions

Frequency vectors of strings can be exploited to improve the efficiency of edit string similarity join. The statistics can be used to design an independent filter and to partition a dataset into data chunks with guaranteed distances, so that a remarkable proportion of candidate pairs can be pruned away without paying to enumerate them. Experiments confirmed our views.

References

Afrati, F.N., Sarma, A.D., Menestrina, D., *et al.*, 2012. Fuzzy joins using MapReduce. Int. Conf. on Data Engineering, p.498-509. https://doi.org/10.1109/ICDE.2012.66

Arasu, A., Ganti, V., Kaushik, R., 2006. Efficient exact set-similarity joins. Int. Conf. on Very Large Data Bases, p.918-929.

Bayardo, R.J., Ma, Y., Srikant, R., 2007. Scaling up all pairs similarity search. Int. World Wide Web Conf., p.131-140. https://doi.org/10.1145/1242572.1242591

Chaudhuri, S., Ganjam, K., Ganti, V., *et al.*, 2003. Robust and efficient fuzzy match for online data cleaning. Int. SIGMOD Conf. on Management of Data, p.313-324. https://doi.org/10.1145/872757.872796

Chaudhuri, S., Ganti, V., Kaushik, R., 2006a. Data debugger: an operator-centric approach for data quality solutions. *IEEE Data Eng. Bull.*, **29**(2):60-66.

Chaudhuri, S., Ganti, V., Kaushik, R., 2006b. A primitive operator for similarity joins in data cleaning. Int. Conf. on Data Engineering, p.687-698. https://doi.org/10.1109/ICDE.2006.9

Dong, X., Halevy, A.Y., Yu, C., 2007. Data integration with uncertainty. Int. Conf. on Very Large Data Bases, p.687-698.

Feng, J., Wang, J., Li, G., 2012. Trie-join: a trie-based method for efficient string similarity joins. *VLDB J.*, **21**(4):437-461. https://doi.org/10.1007/s00778-011-0252-8

Ge, T., Li, Z., 2011. Approximate substring matching over uncertain strings. *Proc. VLDB Endow.*, **4**(11):772-782.

Gravano, L., Ipeirotis, P.G., Jagadish, H.V., *et al.*, 2001. Approximate string joins in a database (almost) for free. Int. Conf. on Very Large Data Bases, p.491-500.

Hadjieleftheriou, M., Srivastava, D., 2010. Weighted Set-Based String Similarity. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering. AT&T Lab-Research.

Henzinger, M.R., 2006. Finding near-duplicate web pages: a large-scale evaluation of algorithms. Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, p.284-291.
https://doi.org/10.1145/1148170.1148222

Ji, S., Li, G., Li, C., *et al.*, 2009. Efficient interactive fuzzy keyword search. Int. World Wide Web Conf., p.371-380. https://doi.org/10.1145/1526709.1526760

Li, C., Lu, J., Lu, Y., 2008. Efficient merging and filtering algorithms for approximate string searches. Int. Conf. on Data Engineering, p.257-266.
https://doi.org/10.1109/ICDE.2008.4497434

Li, G., Deng, D., Wang, J., *et al.*, 2011. Pass-Join: a partition-based method for similarity joins. *Proc. VLDB Endow.*, **5**(3):253-264.
https://doi.org/10.14778/2078331.2078340

Metwally, A., Agrawal, D., Abbadi, A.E., 2007. Detectives: detecting coalition hit inflation attacks in advertising networks streams. Int. World Wide Web Conf., p.241-250. https://doi.org/10.1145/1242572.1242606

Navarro, G., Salmela, L., 2009. Indexing variable length substrings for exact and approximate matching. Int. Symp. on String Processing and Information Retrieval, p.214-221.
https://doi.org/10.1007/978-3-642-03784-9_21

Qin, J., Wang, W., Xiao, C., *et al.*, 2013. Vchunkjoin: an efficient algorithm for edit similarity joins. *Trans. Knowl. Dat. Eng.*, **25**(8):1916-1929.
https://doi.org/10.1109/TKDE.2012.79

Sarawagi, S., Kirpal, A., 2004. Efficient set joins on similarity predicates. Int. SIGMOD Conf. on Management of Data, p.743-754.
https://doi.org/10.1145/1007568.1007652

Scott, D.W., 1979. On optimal and data-based histograms. *Biometrika*, **66**:605-610.
https://doi.org/10.1093/biomet/66.3.605

Vernica, R., Carey, M.J., Li, C., 2010. Efficient parallel set-similarity joins using MapReduce. Int. SIGMOD Conf. on Management of Data, p.495-506.
https://doi.org/10.1145/1807167.1807222

Wang, J., Li, G., Feng, J., 2011. Fast-join: an efficient method for fuzzy token matching based string similarity join. Int. Conf. on Data Engineering, p.458-469.
https://doi.org/10.1109/ICDE.2011.5767865

Wang, W., Xiao, C., Lin, X., *et al.*, 2009. Efficient approximate entity extraction with edit distance constraints. Int. SIGMOD Conf. on Management of Data, p.759-770. https://doi.org/10.1145/1559845.1559925

Xiao, C., Wang, W., Lin, X., 2008a. Ed-Join: an efficient algorithm for similarity joins with edit distance constraints. *Proc. VLDB Endow.*, **1**(1):933-944.
https://doi.org/10.14778/1453856.1453957

Xiao, C., Wang, W., Lin, X., *et al.*, 2008b. Efficient similarity joins for near duplicate detection. Int. World Wide Web Conf., p.131-140.
https://doi.org/10.1145/2000824.2000825