A Partition-Based Method for String Similarity Joins with Edit-Distance Constraints

GUOLIANG LI, DONG DENG, and JIANHUA FENG, Tsinghua University

As an essential operation in data cleaning, the similarity join has attracted considerable attention from the database community. In this article, we study string similarity joins with edit-distance constraints, which find similar string pairs from two large sets of strings whose edit distance is within a given threshold. Existing algorithms are efficient either for short strings or for long strings, and there is no algorithm that can efficiently and adaptively support both short strings and long strings. To address this problem, we propose a new filter, called *the segment filter*. We partition a string into a set of segments and use the segments as a filter to find similar string pairs. We first create inverted indices for the segments. Then for each string, we select some of its substrings, identify the selected substrings from the inverted indices, and take strings on the inverted lists of the found substrings as candidates of this string. Finally, we verify the candidates to generate the final answer. We devise efficient techniques to select substrings and prove that our method can minimize the number of selected substrings. We develop novel pruning techniques to efficiently verify the candidates. We also extend our techniques to support normalized edit distance. Experimental results show that our algorithms are efficient for both short strings and long strings, and outperform state-of-the-art methods on real-world datasets.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query Processing*

General Terms: Algorithms, Performance, Theory, Design

Additional Key Words and Phrases: String similarity join, edit distance, segment filter

ACM Reference Format:
Li, G., Deng, D., and Feng, J. 2013. A partition-based method for string similarity joins with edit-distance constraints. ACM Trans. Datab. Syst. 38, 2, Article 9 (June 2013), 33 pages.
DOI: http://dx.doi.org/10.1145/2487259.2487261

1. INTRODUCTION

A string similarity join between two sets of strings finds all *similar* string pairs from the two sets. For example, consider two sets of strings {vldb, sigmod, ...} and {*pvldb*, *icde*, ...}. We want to find all similar pairs, for instances, ⟨vldb, *pvldb*⟩. Many similarity functions have been proposed to quantify the similarity between two strings, such as Jaccard similarity, Cosine similarity, and edit distance. In this articles, we study string similarity joins with edit-distance constraints, which, given two large sets of strings, find all *similar* string pairs from the two sets, such that the edit distance between each string pair is within a given threshold. The string similarity join is

This work was partly supported by the National Natural Science Foundation of China under Grant No. 61003004 and 61272090, National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302206, and a project of Tsinghua University under Grant No. 20111081073, and the "NExT Research Center" funded by MDA, Singapore, under Grant No. WBS:R-252-300-001-490.
Author's address: Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNList), Tsinghua University, Beijing 100084, China; email: liguoliang@tsinghua.edu.cn.

DOI: http://dx.doi.org/10.1145/2487259.2487261

an essential operation in many applications, such as data integration and cleaning, near duplicate object detection and elimination, and collaborative filtering [Xiao et al. 2008a].

Existing methods can be broadly classified into two categories. The first one uses a *filter-and-refine* framework, such as Part-Enum [Arasu et al. 2006], All-Pairs-Ed [Bayardo et al. 2007], ED-JOIN [Xiao et al. 2008a]. In the *filter* step, they generate signatures for each string and use the signatures to generate candidate pairs. In the *refine* step, they verify the candidate pairs to generate the final result. However, these approaches are inefficient for the datasets with short strings (e.g., person names and locations) [Wang et al. 2010]. The main reason is that they cannot select high-quality signatures for short strings and will generate large numbers of candidates that need to be further verified. The second one, TRIE-JOIN [Wang et al. 2010], adopts a trie-based framework, which uses a trie structure to share prefixes and utilizes prefix pruning to improve the performance. However TRIE-JOIN is inefficient for long strings (e.g., paper titles and abstracts). There are two main reasons. First it is expensive to traverse the trie with long strings. Second long strings have a small number of shared prefixes and TRIE-JOIN has limited pruning power.

If a system wants to support both short strings and long strings, we have to implement and maintain two separate codes, and tune many parameters to select the best method. To alleviate this problem, it calls for an adaptive method that can efficiently support both short strings and long strings. In this article we propose a new filter, called *the segment filter*, and devise efficient filtering algorithms. We devise a partition scheme to partition a string into a set of segments and prove that if a string s is similar to string r, s must have a substring that matches a segment of r. Based on this observation, we use the segments as a filter and propose a segment-filter based framework. We first partition strings into segments and create inverted indices for the segments. Then for each string s, we select some of its substrings and search for the selected substrings in the inverted indices. If a selected substring appears in the inverted index, each string r on the inverted list of this substring (i.e., r contains the substring) may be similar to s, and we take r and s as a candidate pair. Finally we verify the candidate pairs to generate the final answer. We develop effective techniques to select high-quality substrings and prove that our method can minimize the number of selected substrings. We also devise novel pruning techniques to efficiently verify the candidate pairs. To summarize, we make the following contributions.

—We propose a segment-filter-based framework. We first partition strings into a set of segments. Then given a string, we select some of its substrings and take those strings whose segments match one of the selected substrings as the candidates of this string. We call this pruning technique *the segment filter*. Finally we verify the candidates to generate the final answer.
—To improve the segment-filter step, we discuss how to effectively select substrings and prove that our method can minimize the number of selected substrings.
—To improve the verification step, we propose a length-aware method, an extension-based method, and an iterative-based method to efficiently verify a candidate.
—We extend our techniques to support normalized edit distance and R-S join.
—We have conducted an extensive set of experiments. Experimental results show that our algorithms are very efficient for both short strings and long strings, and outperform state-of-the-art methods on real-world datasets.

The rest of this article is organized as follows. We formalize our problem in Section 2. Section 3 introduces our segment-filter-based framework. We propose to effectively select substrings in Section 4 and develop novel techniques to efficiently verify candidates in Section 5. We discuss how to support normalized edit distance and R-S join in

Table I. A Set of Strings

(a) Strings

Strings
avataresha
caushik chakrabar
kaushik chakrab
kaushuk chadhui
kausic chakduri
vankatesh

(b) Sorted by Length (Ascending)

ID	Strings	Len
s_1	vankatesh	9
s_2	avataresha	10
s_3	kaushik chakrab	15
s_4	kaushuk chadhui	15
s_5	kausic chakduri	15
s_6	caushik chakrabar	17

(c) Sorted by Length (Descending)

ID	Strings	Len
s_6	caushik chakrabar	17
s_5	kausic chakduri	15
s_4	kaushuk chadhui	15
s_3	kaushik chakrab	15
s_2	avataresha	10
s_1	vankatesh	9

Section 6. Experimental results are provided in Section 7. We review related work in Section 8 and make a conclusion in Section 9.

2. PROBLEM FORMULATION

Given two collections of strings, a similarity join finds all similar string pairs from the two collections. In this article, we use edit distance to quantify the similarity between two strings. Formally, the edit distance between two strings r and s, denoted by $\text{ED}(r, s)$, is the minimum number of single-character edit operations (i.e., insertion, deletion, and substitution) needed to transform r to s. For example, $\text{ED}(\text{“}\texttt{kausic chakduri}\text{”}, \text{“}\texttt{kaushuk chadhui}\text{”}) = 6$.

Here two strings are *similar* if their edit distance is not larger than a specified edit-distance threshold τ. We formalize the problem of string similarity joins with edit-distance constraints as follows.

Definition 2.1 (*String Similarity Joins*). Given two sets of strings $\mathcal{R}$ and $\mathcal{S}$ and an edit-distance threshold τ, a similarity join finds all similar string pairs $\langle r, s\rangle \in \mathcal{R} \times \mathcal{S}$ such that $\text{ED}(r, s) \leq \tau$.

In the article we first focus on self join ($\mathcal{R} = \mathcal{S}$). We will discuss how to support R-S join ($\mathcal{R} \neq \mathcal{S}$) in Section 6. For example, consider the strings in Table I(a). Suppose the edit-distance threshold $\tau = 3$. $\langle$“kaushik chakrab”, “caushik chakrabar”$\rangle$ is a similar pair as their edit distance is not larger than τ.

3. THE SEGMENT FILTER BASED FRAMEWORK

We first introduce a partition scheme to partition a string into several disjoint segments (Section 3.1), and then propose a segment filter based framework (Section 3.2).

3.1. Partition Scheme

Given a string s, we partition it into $\tau + 1$ disjoint segments, and the length of each segment is not smaller than one[1]. For example, consider string s_1=“vankatesh”. Suppose $\tau = 3$. We can partition s_1 into $\tau + 1 = 4$ segments, for instance, {“va”,“nk”,“at”, “esh”}.

Consider two strings r and s. If s has no substring that matches a segment of r, then s cannot be similar to r based on the pigeonhole principle as stated in Lemma 3.1.

LEMMA 3.1. *Given a string r with $\tau + 1$ segments and a string s, if s is similar to r within threshold τ, s must contain a substring that matches a segment of r.*

PROOF. We prove it by contradiction. Suppose string s contains no substring that matches a segment of string r. In other words, any segment of r will not match any substring of s. Thus for any transformation $\mathcal{T}$ from r to s, in each segment of r there at least exists one edit operation. That is in any transformation $\mathcal{T}$ there are at least $\tau + 1$

[1]The length of string $s(|s|)$ should be larger than τ, that is, $|s| \geq \tau + 1$.

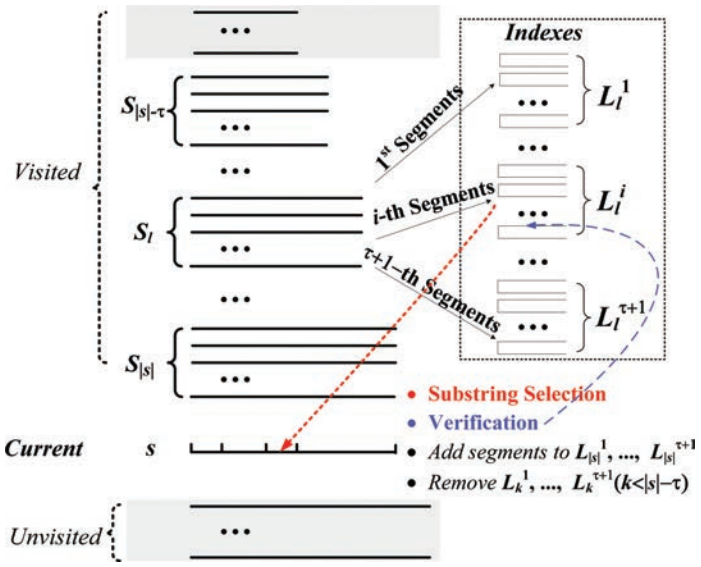


Fig. 1. SEGFILTER framework.

edit operations. This contradicts that s is similar to r. Thus s must contain a substring that matches a segment of r. □

In other words, if s is similar to r, then s must contain a substring matching a segment of r. For example, consider the strings in Table I. Suppose $\tau = 3$. s_1 = "vankatesh" has four segments {"va", "nk", "at", "esh"}. As strings s_3, s_4, s_5, s_6 have no substring that matches segments of s_1, they are not similar to s_1.

Given a string, there could be many strategies to partition the string into $\tau + 1$ segments. A good partition strategy can reduce the number of candidate pairs and improve the performance. Intuitively, the shorter a segment of r is, the higher probability the segment appears in other strings, and the more strings will be taken as r's candidates, thus the pruning power is lower. Based on this observation, we do not want to keep short segments in the partition. In other words, each segment should have nearly the same length. Accordingly we propose an *even-partition* scheme as follows.

Consider a string s with length $|s|$. In even partition scheme, each segment has a length of $\lfloor \frac{|s|}{\tau+1} \rfloor$ or $\lceil \frac{|s|}{\tau+1} \rceil$, thus the maximal length difference between two segments is 1. Let $k = |s| - \lfloor \frac{|s|}{\tau+1} \rfloor * (\tau + 1)$. In even partition, the last k segments have length $\lceil \frac{|s|}{\tau+1} \rceil$, and the first $\tau + 1 - k$ ones have length $\lfloor \frac{|s|}{\tau+1} \rfloor$. For example, consider s_1="vankatesh" and $\tau = 3$. Then length of s_1 ($|s_1|$) is 9. $k = 1$. s_1 has four segments {"va","nk","at", "esh"}.

Although we can devise other partition schemes, it is time consuming to select a good partition strategy. Note that the time for selecting a partition strategy should be included in the similarity-join time. In this article we focus on the even-partition scheme and leave how to select a good partition scheme as a future work.

3.2. The Segment-Filter-Based Framework

We have an observation that if a strings s does not have a substring that matches a segment of r, we can prune the pair $\langle s, r \rangle$. We use this feature to prune large numbers of dissimilar pairs. To this end, we propose a segment-filter-based framework, called SEGFILTER. Figure 1 illustrates our framework.

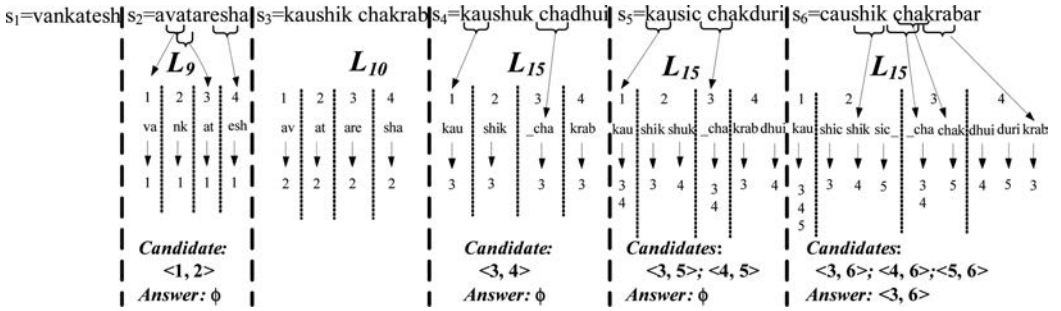


Fig. 2. An example of our segment filter based framework.

For ease of presentation, we first introduce some notations. Let $\mathcal{S}_l$ denote the set of strings with length l and $\mathcal{S}_l^i$ denote the set of the i-th segments of strings in $\mathcal{S}_l$. We build an inverted index for each $\mathcal{S}_l^i$, denoted by $\mathcal{L}_l^i$. Given an i-th segment w, let $\mathcal{L}_l^i(w)$ denote the inverted list of segment w, that is, the set of strings whose i-th segments are w. We use the inverted indices to do similarity joins as follows.

We first sort strings based on their lengths in ascending order. For the strings with the same length, we sort them in alphabetical order. Then we visit strings in order. Consider the current string s with length $|s|$. We find s's similar strings among the visited strings using the inverted indices. To efficiently find such strings, we create indices only for visited strings to avoid enumerating a string pair twice. Based on length filtering [Gravano et al. 2001], we check whether the strings in $\mathcal{L}_l^i$ ($|s| - \tau \le l \le |s|, 1 \le i \le \tau + 1$) are similar to s. Without loss of generality, consider inverted index $\mathcal{L}_l^i$. We find s's similar strings in $\mathcal{L}_l^i$ as follows.

—Substring Selection. If s is similar to a string in $\mathcal{L}_l^i$, then s should contain a substring that matches a segment in $\mathcal{L}_l^i$. A straightforward method enumerates all of s's substrings, and for each substring checks whether it appears in $\mathcal{L}_l^i$. Actually we do not need to consider all substrings of s. Instead we only select some substrings (denoted by $\mathcal{W}(s, \mathcal{L}_l^i)$) and use the selected substrings to find similar pairs. We discuss how to generate $\mathcal{W}(s, \mathcal{L}_l^i)$ in Section 4. For each selected substring $w \in \mathcal{W}(s, \mathcal{L}_l^i)$, we check whether it appears in $\mathcal{L}_l^i$. If so, for each $r \in \mathcal{L}_l^i(w)$, $\langle r, s\rangle$ is a candidate pair.

—Verification. To verify whether a candidate pair $\langle r, s\rangle$ is an answer, a straightforward method computes their real edit distance. However this method is rather expensive. To address this issue, we develop effective techniques to do efficient verification in Section 5.

After finding similar strings for s, we partition s into $\tau + 1$ segments and insert the segments into inverted index $\mathcal{L}_{|s|}^i (1 \le i \le \tau + 1)$. Then we visit strings after s and iteratively we can find all similar pairs. Note that we can remove the inverted index $\mathcal{L}_k^i$ for $k < |s| - \tau$. Thus we maintain at most $(\tau + 1)^2$ inverted indices $\mathcal{L}_l^i$ for $|s| - \tau \le l \le |s|$ and $1 \le i \le \tau + 1$. In this article we focus on the case that the index can be fit in the memory. We leave dealing with a very large dataset as a future work.

For example, consider the strings in Table I. Suppose $\tau = 3$. We find similar pairs as follows (see Figure 2). For the first string $s_1 =$ "vankatesh", we partition it into $\tau + 1$ segments and insert the segments into the inverted indices for strings with length 9, that is, $\mathcal{L}_9^1$, $\mathcal{L}_9^2$, $\mathcal{L}_9^3$, and $\mathcal{L}_9^4$. Next for $s_2 =$ "avataresha", we enumerate its substrings and check whether each substring appears in $\mathcal{L}_{|s_2|-\tau}^i, \ldots, \mathcal{L}_{|s_2|}^i (1 \le i \le \tau + 1)$. Here we find "va" in $\mathcal{L}_9^1$, "at" in $\mathcal{L}_9^3$, and "esh" in $\mathcal{L}_9^4$. For segment "va", as $\mathcal{L}_9^1(\text{va}) = \{s_1\}$. The

ALGORITHM 1: SEGFILTER $(\mathcal{S}, \tau)$

Input: $\mathcal{S}$: A collection of strings
τ: A given edit-distance threshold
Output: $\mathcal{A} = \{(s \in \mathcal{S}, r \in \mathcal{S}) \mid \text{ED}(s, r) \leq \tau\}$

```
1 begin
2     Sort S first by string length and second in alphabetical order;
3     for s ∈ S do
4         for L_l^i (|s| − τ ≤ l ≤ |s|, 1 ≤ i ≤ τ + 1) do
5             W(s, L_l^i) = SUBSTRINGSELECTION(s, L_l^i);
6             for w ∈ W(s, L_l^i) do
7                 if w is in L_l^i then VERIFICATION(s, L_l^i(w), τ);
8         Partition s and add its segments into L_|s|^i;
```

Function SubstringSelection$(s, \mathcal{L}_l^i)$

Input: s: A string; $\mathcal{L}_l^i$: Inverted index
Output: $\mathcal{W}(s, \mathcal{L}_l^i)$: Selected substrings

```
1 begin
2     W(s, L_l^i) = {w | w is a substring of s};
```

Function Verification$(s, \mathcal{L}_l^i(w), \tau)$

Input: s: A string; $\mathcal{L}_l^i(w)$: Inverted list; τ: Threshold
Output: $\mathcal{A} = \{(s, r \in \mathcal{L}_l^i(w)) | \text{ED}(s, r) \leq \tau\}$

```
1 begin
2     for r ∈ L_l^i(w) do
3         if ED(s, r) ≤ τ then A ← ⟨s, r⟩;
```

Fig. 3. SEGFILTER algorithm.

pair $\langle s_2, s_1 \rangle$ is a candidate pair. We verify the pair and it is not an answer as the edit distance is larger than τ. Next we partition s_2 into four segments and insert them into $\mathcal{L}^1_{|s_2|}, \mathcal{L}^2_{|s_2|}, \mathcal{L}^3_{|s_2|}, \mathcal{L}^4_{|s_2|}$. Similarly we repeat these steps and find all similar pairs.

We give the pseudocode of our algorithm in Figure 3. We sort strings first by length and then in alphabetical order (line 2). Then, we visit each string in the sorted order (line 3). For each inverted index $\mathcal{L}_l^i(|s| - \tau \leq l \leq |s|, 1 \leq i \leq \tau + 1)$, we select the substrings of s (line 4-line 4) and check whether each selected substring w is in $\mathcal{L}_l^i$ (line 8-line 7). If yes, for any string r in the inverted list of w in $\mathcal{L}_l^i$, that is, $\mathcal{L}_l^i(w)$, the string pair $\langle r, s \rangle$ is a candidate pair. We verify the pair (line 7). Finally, we partition s into $\tau + 1$ segments, and inserts the segments into the inverted index $\mathcal{L}^i_{|s|}(1 \leq i \leq \tau + 1)$ (line 8). Here function SUBSTRINGSELECTION selects all substrings and function VERIFICATION computes the real edit distance of two strings to verify the candidates using dynamic-programming algorithm. To improve the performance, we propose effective techniques to improve the substring-selection step (the SUBSTRINGSELECTION function) in Section 4 and the verification step (the VERIFICATION function) in Section 5.

Complexity. We first analyze the space complexity. Our indexing structure includes segments and inverted lists of segments. We first give the space complexity of segments. For each string in $\mathcal{S}_l$ we generate $\tau + 1$ segments. Thus the number of segments is at most $(\tau + 1) \times |\mathcal{S}_l|$, where $|\mathcal{S}_l|$ is the number of strings in $\mathcal{S}_l$. As we can use an integer

to encode a segment, the space complexity of segments is

$$\mathcal{O}\left(\max_{l_{min}\le j\le l_{max}}\sum_{l=j-\tau}^{j}(\tau+1)\times|\mathcal{S}_l|\right),$$

where l_{min} and l_{max} respectively denote the minimal and the maximal string length.

Next we give the complexity of inverted lists. For each string in $\mathcal{S}_l$, as the i-th segment of the string corresponds to an element in $\mathcal{L}_l^i$, $|\mathcal{S}_l| = |\mathcal{L}_l^i|$. The space complexity of inverted lists (i.e., the sum of the lengths of inverted lists) is

$$\mathcal{O}\left(\max_{l_{min}\le j\le l_{max}}\sum_{l=j-\tau}^{j}\sum_{i=1}^{\tau+1}|\mathcal{L}_l^i| = \max_{l_{min}\le j\le l_{max}}\sum_{l=j-\tau}^{j}(\tau+1)\times|\mathcal{S}_l|\right).$$

Then we give the time complexity. To sort the strings, we can first group the strings based on lengths and then sort strings in each group. Thus the sort complexity is $\mathcal{O}(\sum_{l_{min}\le l\le l_{max}}|\mathcal{S}_l|log(|\mathcal{S}_l|))$. For each string s, we select its substring set $\mathcal{W}(s,\mathcal{L}_l^i)$ for $|s|-\tau\le l\le|s|$, $1\le i\le\tau+1$. The selection complexity is $\mathcal{O}(\sum_{s\in\mathcal{S}}\sum_{l=|s|-\tau}^{|s|}\sum_{i=1}^{\tau+1}\mathcal{X}(s,\mathcal{L}_l^i))$, where $\mathcal{X}(s,\mathcal{L}_l^i)$ is the selection time complexity for $\mathcal{W}(s,\mathcal{L}_l^i)$, which is $\mathcal{O}(\tau)$ (see Section 4). The selection complexity is $\mathcal{O}(\tau^3|\mathcal{S}|)$. For each substring $w\in\mathcal{W}(s,\mathcal{L}_l^i)$, we verify whether strings in $\mathcal{L}_l^i(w)$ are similar to s. The verification complexity is

$$\mathcal{O}\left(\sum_{s\in\mathcal{S}}\sum_{l=|s|-\tau}^{|s|}\sum_{i=1}^{\tau+1}\sum_{w\in\mathcal{W}(s,\mathcal{L}_l^i)}\sum_{r\in\mathcal{L}_l^i(w)}\mathcal{V}(s,r)\right),$$

where $\mathcal{V}(s,r)$ is the complexity for verifying $\langle s,r\rangle$, which is $\mathcal{O}(\tau*\min(|s|,|r|))$ (see Section 5). In the article we propose to reduce the size of $\mathcal{W}(s,\mathcal{L}_l^i)$ and improve the verification cost $\mathcal{V}(s,r)$.

4. IMPROVING THE FILTER STEP BY SELECTING EFFECTIVE SUBSTRINGS

For any string $s\in\mathcal{S}$ and a length l ($|s|-\tau\le l\le|s|$), we select a substring set $\mathcal{W}(s,l)=\cup_{i=1}^{\tau+1}\mathcal{W}(s,\mathcal{L}_l^i)$ of s and use substrings in $\mathcal{W}(s,l)$ to find the candidates of s. We need to guarantee completeness of the method using $\mathcal{W}(s,l)$ to find candidate pairs. That is any similar pair must be found as a candidate pair. Next we give the formal definition.

Definition 4.1 (*Completeness*). A substring selection method satisfies completeness, if for any string s and a length $l(|s|-\tau\le l\le|s|)$, $\forall\, r$ with length l that is similar to s and visited before s, r must have an i-th segment r_m that matches a substring $s_m\in\mathcal{W}(s,\mathcal{L}_l^i)$ where $1\le i\le\tau+1$.

A straightforward method is to add all substrings of s into $\mathcal{W}(s,l)$. As s has $|s|-i+1$ substrings with length i, the total number of s's substrings is $\sum_{i=1}^{|s|}(|s|-i+1)=\frac{|s|*(|s|+1)}{2}$. For long strings, there are large numbers of substrings and it is rather expensive to enumerate all substrings.

Intuitively, the smaller size of $\mathcal{W}(s,l)$, the higher performance. Thus we want to find substring sets with smaller sizes. In this section, we propose several methods to select the substring set $\mathcal{W}(s,l)$. As $\mathcal{W}(s,l)=\cup_{i=1}^{\tau+1}\mathcal{W}(s,\mathcal{L}_l^i)$ and we want to use inverted index $\mathcal{L}_l^i$ to do efficient filtering, next we focus on how to generate $\mathcal{W}(s,\mathcal{L}_l^i)$ for $\mathcal{L}_l^i$. Table II shows the notations used in this article.

Table II. Notations

Notation	Description
τ	edit distance threshold
$\mathcal{W}_\ell(s,l)$	substring set selected by length-based selection method
$\mathcal{W}_f(s,l)$	substring set selected by shift-based selection method
$\mathcal{W}_p(s,l)$	substring set selected by position-aware selection method
$\mathcal{W}_m(s,l)$	substring set selected by multimatch-aware selection method
p_{min}	minimal start position of position-aware substring selection
p_{max}	maximal start position of position-aware substring selection
$\perp_i^l$	minimal start position of multimatch-aware from left-side perspective
$\perp_i^r$	minimal start position of multimatch-aware from right-side perspective
$\perp_i$	minimal start position of multimatch-aware substring selection

Length-based Method. As segments in $\mathcal{L}_l^i$ have the same length, denoted by l_i, the length-based method selects all substrings of s with length l_i, denoted by $\mathcal{W}_\ell(s, \mathcal{L}_l^i)$. Let $\mathcal{W}_\ell(s,l) = \cup_{i=1}^{\tau+1}\mathcal{W}_\ell(s, \mathcal{L}_l^i)$. The length-based method satisfies completeness, as it selects all substrings with length l_i. The size of $\mathcal{W}_\ell(s, \mathcal{L}_l^i)$ is $|\mathcal{W}_\ell(s, \mathcal{L}_l^i)| = |s| - l_i + 1$, and the number of selected substrings is $|\mathcal{W}_\ell(s,l)| = (\tau+1)(|s|+1) - l$.

Shift-based Method. However the length-based method does not consider the positions of segments. To address this problem, Wang et al. [2009] proposed a shift-based method to address the entity identification problem. We can extend their method to support our problem as follows. As segments in $\mathcal{L}_l^i$ have the same length, they have the same start position, denoted by p_i, where $p_1 = 1$ and $p_i = p_1 + \sum_{k=1}^{i-1} l_k$ for $i > 1$. The shift-based method selects s's substrings with start positions in $[p_i - \tau, p_i + \tau]$ and with length l_i, denoted by $\mathcal{W}_f(s, \mathcal{L}_l^i)$. Let $\mathcal{W}_f(s,l) = \cup_{i=1}^{\tau+1}\mathcal{W}_f(s, \mathcal{L}_l^i)$. The size of $\mathcal{W}_f(s, \mathcal{L}_l^i)$ is $|\mathcal{W}_f(s, \mathcal{L}_l^i)| = 2\tau+1$. The number of selected substrings is $|\mathcal{W}_f(s,l)| = (\tau+1)(2\tau+1)$.

The basic idea behind the method is as follows. Suppose a substring s_m of s with start position smaller than $p_i - \tau$ or larger than $p_i + \tau$ matches a segment in $\mathcal{L}_l^i$. Consider a string $r \in \mathcal{L}_l^i(s_m)$. We can partition $s(r)$ into three parts: the matching part $s_m(r_m)$, the left part before the matching part $s_l(r_l)$, and the right part after the matching part $s_r(r_r)$. As the start position of r_m is p_i and the start position of s_m is smaller than $p_i - \tau$ or larger than $p_i + \tau$, the length difference between s_l and r_l must be larger than τ. If we align the two strings by matching s_m and r_m (i.e., transforming r_l to s_l, matching r_m with s_m, and transforming r_r to s_r), they will not be similar, thus we can prune substring s_m. Hence the shift-based method satisfies completeness.

However, the shift-based method still involves many unnecessary substrings. For example, consider two strings s_1 = "vankatesh" and s_2 = "avataresha". Suppose $\tau = 3$ and "vankatesh" is partitioned into four segments {va, nk, at, esh}. s_2 = "avataresha" contains a substring "at" that matches the third segment in "vankatesh", the shift-based method will select it as a substring. However we can prune it and the reason is as follows. Suppose we partition the two strings into three parts based on the matching segment. For instance, we partition "vankatesh" into {"vank" , "at", "esh"}, and "avataresha" into {"av", "at", "aresha"}. Obviously the minimal edit distance (length difference) between the left parts ("vank" and "av") is 2 and the minimal edit distance (length difference) between the right parts ("esh" and "aresha") is 3. Thus if we align the two strings using the matching segment "at", they will not be similar. In this way, we can prune the substring "at".

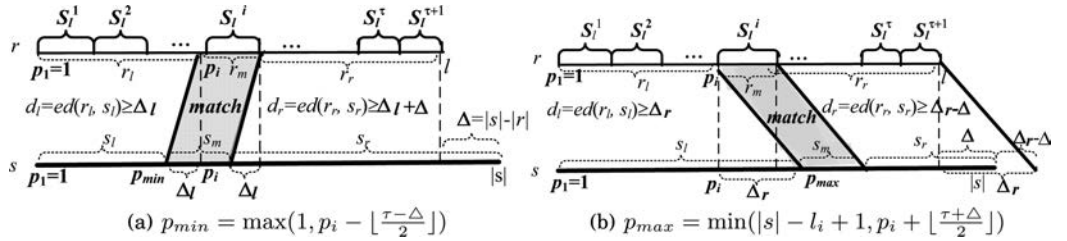


Fig. 4. Position-aware substring selection.

4.1. Position-Aware Substring Selection

Notice that all the segments in $\mathcal{L}_l^i$ have the same length l_i and the same start position p_i. Without loss of generality, we consider a segment $r_m \in \mathcal{L}_l^i$. Moreover, all the strings in inverted list $\mathcal{L}_l^i(r_m)$ have the same length l ($l \leq |s|$), and we consider a string r that contains segment r_m. Suppose s has a substring s_m that matches r_m. Next we give the possible start positions of s_m. We still partition $s(r)$ into three parts: the matching part $s_m(r_m)$, the left part $s_l(r_l)$, and the right part $s_r(r_r)$. If we align r and s by matching $r_m = s_m$, that is we transform r to s by first transforming r_l to s_l with $d_l = \text{ED}(r_l, s_l)$ edit operations, then matching r_m with s_m, and finally transforming r_r to s_r with $d_r = \text{ED}(r_r, s_r)$ edit operations, the total transformation distance is $d_l + d_r$. If s is similar to r, $d_l + d_r \leq \tau$. Based on this observation, we give s_m's minimal start position (p_{min}) and the maximal start position (p_{max}) as illustrated in Figure 4.

Minimal Start Position. Suppose the start position of s_m, denoted by p, is not larger than p_i. Let $\triangle = |s| - |r|$ and $\triangle_l = p_i - p$. We have $d_l = \text{ED}(r_l, s_l) \geq \triangle_l$ and $d_r = \text{ED}(r_r, s_r) \geq \triangle_l + \triangle$, as illustrated in Figure 4(a). If s is similar to r (or any string in $\mathcal{L}_l^i(r_m)$), we have $\triangle_l + (\triangle_l + \triangle) \leq d_l + d_r \leq \tau$. That is $\triangle_l \leq \lfloor \frac{\tau-\triangle}{2} \rfloor$ and $p = p_i - \triangle_l \geq p_i - \lfloor \frac{\tau-\triangle}{2} \rfloor$. Thus $p_{min} \geq p_i - \lfloor \frac{\tau-\triangle}{2} \rfloor$. As $p_{min} \geq 1$, $p_{min} = \max(1, p_i - \lfloor \frac{\tau-\triangle}{2} \rfloor)$.

Maximal Start Position. Suppose the start position of s_m, p, is larger than p_i. Let $\triangle = |s| - |r|$ and $\triangle_r = p - p_i$. We have $d_l = \text{ED}(r_l, s_l) \geq \triangle_r$ and $d_r = \text{ED}(r_r, s_r) \geq |\triangle_r - \triangle|$ as illustrated in Figure 4(b). If $\triangle_r \leq \triangle$, $d_r \geq \triangle - \triangle_r$. Thus $\triangle = \triangle_r + (\triangle - \triangle_r) \leq d_l + d_r \leq \tau$, and in this case, the maximal value of $\triangle_r$ is $\triangle$; otherwise if $\triangle_r > \triangle$, $d_r \geq \triangle_r - \triangle$. If s is similar to r (or ***any*** string in $\mathcal{L}_l^i(r_m)$), we have

$$\triangle_r + (\triangle_r - \triangle) \leq d_l + d_r \leq \tau.$$

That is $\triangle_r \leq \lfloor \frac{\tau+\triangle}{2} \rfloor$, and $p = p_i + \triangle_r \leq p_i + \lfloor \frac{\tau+\triangle}{2} \rfloor$. Thus $p_{max} \leq p_i + \lfloor \frac{\tau+\triangle}{2} \rfloor$. As the segment length is l_i, based on the boundary, we have $p_{max} \leq |s| - l_i + 1$. Thus $p_{max} = \min(|s| - l_i + 1, p_i + \lfloor \frac{\tau+\triangle}{2} \rfloor)$.

For example, consider string $r =$ "vankatesh". Suppose $\tau = 3$ and "vankatesh" is partitioned into four segments, {va, nk, at, esh}. For string $s =$ "avataresha", we have $\triangle = |s| - |r| = 1$. $\lfloor \frac{\tau-\triangle}{2} \rfloor = 1$ and $\lfloor \frac{\tau+\triangle}{2} \rfloor = 2$. For the first segment "va", $p_1 = 1$. $p_{min} = \max(1, p_1 - \lfloor \frac{\tau-\triangle}{2} \rfloor) = 1$ and $p_{max} = 1 + \lfloor \frac{\tau+\triangle}{2} \rfloor = 3$. Thus we only need to enumerate the following substrings "av", "va", "at" for the first segment. Similarly, we need to enumerate substrings "va", "at", "ta", "ar" for the second segment, "ta", "ar", "re", "es" for the third segment, and "res", "esh", "sha" for the fourth segment. We see that the position-aware method can reduce many substrings over the shift-based method (reducing the number from 28 to 14).

For $\mathcal{L}_l^i$, the position-aware method selects substrings with start positions in $[p_{min}, p_{max}]$ and length l_i, denoted by $\mathcal{W}_p(s, \mathcal{L}_l^i)$. Let $\mathcal{W}_p(s, l) = \cup_{i=1}^{\tau+1} \mathcal{W}_p(s, \mathcal{L}_l^i)$. The

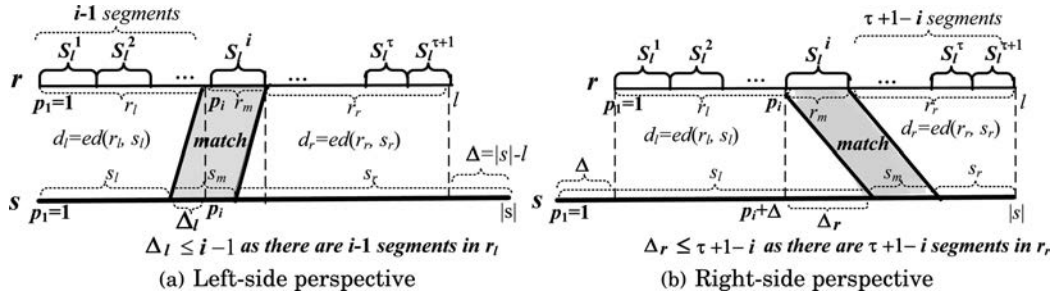


Fig. 5. Multimatch-aware substring selection.

size of $\mathcal{W}_p(s, \mathcal{L}_l^i)$ is $|\mathcal{W}_p(s, \mathcal{L}_l^i)| = \tau + 1$ and the number of selected substrings is $|\mathcal{W}_p(s, l)| = (\tau + 1)^2$. The position-aware method satisfies completeness as formalized in Theorem 4.2.

THEOREM 4.2. *The position-aware substring selection method satisfies the completeness.*

PROOF. See Section A in Appendix. □

4.2. MultiMatch-Aware Substring Selection

We have an observation that string s may have multiple substrings that match some segments of string r. In this case we can discard some of these substrings. For example, consider r = "vankatesh" with four segments, {va, nk, at, esh}. s = "avataresha" has three substrings va, at, esh matching the segments of r. We can discard some of these substrings to reduce the verification cost. To this end, we propose a multimatch-aware substring selection method.

Consider $\mathcal{L}_l^i$. Suppose string s has a substring s_m that matches a segment in $\mathcal{L}_l^i$. If we know that s must have a substring after s_m that will match a segment in $\mathcal{L}_l^j (j > i)$, we can discard substring s_m. For example, s = "avataresha" has a substring "va" matching a segment in r = "vankatesh". Consider the three parts $r_m = s_m =$ "va", $r_l = \phi$ and $s_l =$ "a", and $r_r =$ "nkatesh" and $s_r =$ "taresha". As $d_l \geq 1$, if s and r are similar, $d_r \leq \tau - d_l \leq \tau - 1 = 2$. As there are still 3 segments in r_r, thus s_r must have a substring matching a segment in r_r based on the pigeonhole principle. Thus we can discard the substring "va" and use the next matching substring to find similar pairs. Next we generalize our idea.

Suppose s has a substring s_m with start position p matching a segment $r_m \in \mathcal{L}_l^i$. We still consider the three parts of the two strings: s_l, s_m, s_r and r_l, r_m, r_r as illustrated in Figure 5. Let $\triangle_l = |p_i - p|$. $d_l = \text{ED}(r_l, s_l) \geq \triangle_l$. As there are $i - 1$ segments in s_l, if each segment only has less than 1 edit operation when transforming r_l to s_l, we have $\triangle_l \leq i - 1$. If $\triangle_l \geq i$, $d_l = \text{ED}(r_l, s_l) \geq \triangle_l \geq i$, $d_r = \text{ED}(r_r, s_r) \leq \tau - d_l \leq \tau - i$ (if s is similar to r). As r_r contains $\tau + 1 - i$ segments, s_r must contain a substring matching a segment in r_r based on the pigeonhole principle, which can be proved similar to Lemma 3.1. In this way, we can discard s_m, since for any string $r \in \mathcal{L}_l^i(r_m)$, s must have a substring that matches a segment in the right part r_r, and thus we can identify strings similar to s using the next matching segment. In summary, if $\triangle_l = |p - p_i| \leq i - 1$, we keep the substring with start position p for $\mathcal{L}_l^i$. That is the minimal start position is $\perp_i^l = \max(1, p_i - (i-1))$ and the maximal start position is $\top_i^l = \min(|s| - l_i + 1, p_i + (i-1))$.

For example, suppose $\tau = 3$. Consider r ="vankatesh" with four segments, {va, nk, at, esh}, and s = "avataresha". For the first segment, we have $\perp_i^l = 1 - 0 = 1$ and

$\top_i^l = 1 + 0 = 1$. Thus the selected substring is only "av" for the first segment. For the second segment, we have $\perp_i^l = 3-1 = 2$ and $\top_i^l = 3+1 = 4$. Thus the selected substrings are "va", "at", and "ta" for the second segment. Similarly for the third segment, we have $\perp_i^l = 5 - 2 = 3$ and $\top_i^l = 5 + 2 = 7$, and for the fourth segment, we have $\perp_i^l = 7 - 3 = 4$ and $\top_i^l = \min(8, 7 + 3) = 8$.

Right-side Perspective. The previous observation is made from the left-side perspective. Similarly, we can use the same idea from the right-side perspective. As there are $\tau + 1 - i$ segments on the right part r_r, there are at most $\tau + 1 - i$ edit operations on r_r. If we transform r to s from the right-side perspective, position p_i on r should be aligned with position $p_i + \triangle$ on s as shown in Figure 5(b). Suppose the position p on s matching position p_i on r. Let $\triangle_r = |p - (p_i + \triangle)|$. We have $d_r = \text{ED}(s_r, r_r) \geq \triangle_r$. As there are $\tau + 1 - i$ segments on the right part r_r, we have $\triangle_r \leq \tau + 1 - i$. Thus the minimal start position for $\mathcal{L}_l^i$ is $\perp_i^r = \max(1, p_i + \triangle - (\tau + 1 - i))$ and the maximal start position is $\top_i^r = \min(|s| - l_i + 1, p_i + \triangle + (\tau + 1 - i))$.

Consider this example. We have $\triangle = 1$. For the fourth segment, we have $\perp_i^r = 7 + 1 - (3 + 1 - 4) = 8$ and $\top_i^r = 7 + 1 + (3 + 1 - 4) = 8$. The selected substring is only "sha" for the fourth segment. Similarly for the third segment, we have $\perp_i^r = 5$ and $\top_i^r = 7$. The selected substrings are "ar", "re", and "es" for the third segment.

Combine Left-side Perspective and Right-side Perspective. More interestingly, we can use the two techniques simultaneously. That is for $\mathcal{L}_l^i$, we only select the substrings with start positions between $\perp_i = \max(\perp_i^l, \perp_i^r)$ and $\top_i = \min(\top_i^l, \top_i^r)$ and with length l_i, denoted by $\mathcal{W}_m(s, \mathcal{L}_l^i)$. Let $\mathcal{W}_m(s, l) = \cup_{i=1}^{\tau+1} \mathcal{W}_m(s, \mathcal{L}_l^i)$. The number of selected substrings is $|\mathcal{W}_m(s, l)| = \lfloor \frac{\tau^2 - \triangle^2}{2} \rfloor + \tau + 1$ as stated in Lemma 4.3.

LEMMA 4.3. $|\mathcal{W}_m(s, l)| = \lfloor \frac{\tau^2 - \triangle^2}{2} \rfloor + \tau + 1$.

PROOF. See Section B in Appendix. □

The multimatch-aware method satisfies completeness as stated in Theorem 4.4.

THEOREM 4.4. *The multimatch-aware substring selection method satisfies the completeness.*

PROOF. See Section C in Appendix. □

Consider this example. For the first segment, we have $\perp_i = 1-0 = 1$ and $\top_i = 1+0 = 1$. We select "av" for the first segment. For the second segment, we have $\perp_i = 3 - 1 = 2$ and $\top_i = 3+1 = 4$. We select substrings "va", "at", and "ta" for the second segment. For the third segment, we have $\perp_i = 5+1-(3+1-3) = 5$ and $\top_i = 5+1+(3+1-3) = 7$. We select substrings "ar", "re", and "es" for the third segment. For the fourth segment, we have $\perp_i = 7 + 1 - (3 + 1 - 4) = 8$ and $\top_i = 7 + 1 + (3 + 1 - 4) = 8$. Thus we select the substring "sha" for the fourth segment. The multimatch-aware method only selects 8 substrings.

4.3. Comparison of Selection Methods

We compare the selected substring sets of different methods. Let $\mathcal{W}_\ell(s, l)$, $\mathcal{W}_f(s, l)$, $\mathcal{W}_p(s, l)$, and $\mathcal{W}_m(s, l)$, respectively, denote the sets of selected substrings that use the length-based selection method, the shift-based selection method, the position-aware selection method, and the multimatch-aware selection method. Based on the size analysis of each set, we have $|\mathcal{W}_m(s, l)| \leq |\mathcal{W}_p(s, l)| \leq |\mathcal{W}_f(s, l)| \leq |\mathcal{W}_\ell(s, l)|$. Next we prove $\mathcal{W}_m(s, l) \subseteq \mathcal{W}_p(s, l) \subseteq \mathcal{W}_f(s, l) \subseteq \mathcal{W}_\ell(s, l)$ as formalized in Lemma 4.5.

ALGORITHM 2: SUBSTRINGSELECTION$(s, \mathcal{L}_l^i)$

Input: s: A string; $\mathcal{L}_l^i$: Inverted index
Output: $\mathcal{W}(s, \mathcal{L}_l^i)$: Selected substrings

```
1 begin
2     for p ∈ [⊥_i, ⊤_i] do
3         Add the substring of s with start position p and with length l_i (s[p, l_i]) into W(s, L_l^i);
```

Fig. 6. SUBSTRINGSELECTION algorithm.

LEMMA 4.5. *For any string s and a length l, we have*

$$\mathcal{W}_m(s, l) \subseteq \mathcal{W}_p(s, l) \subseteq \mathcal{W}_f(s, l) \subseteq \mathcal{W}_\ell(s, l).$$

PROOF. See Section D in Appendix. □

Moreover, we can prove that $\mathcal{W}_m(s, l)$ has the minimum size among all substring sets generated by the methods that satisfy completeness as formalized in Theorem 4.6.

THEOREM 4.6. *The substring set $\mathcal{W}_m(s, l)$ generated by the multimatch-aware selection method has the minimum size among all the substring sets generated by the substring selection methods that satisfy completeness.*

PROOF. See Section E in Appendix. □

Theorem 4.6 proves that the substring set $\mathcal{W}_m(s, l)$ has the minimum size. Next we introduce another concept to show the superiority of our multimatch-aware method.

Definition 4.7 (*Minimality*). A substring set $\mathcal{W}(s, l)$ generated by a method with the completeness property satisfies minimality, if for any substring set $\mathcal{W}'(s, l)$ generated by a method with the completeness property, $\mathcal{W}(s, l) \subseteq \mathcal{W}'(s, l)$.

Next we prove that if $l \geq 2(\tau + 1)$ and $|s| \geq l$, the substring set $\mathcal{W}_m(s, l)$ generated by our multimatch-aware selection method satisfies minimality as stated in Theorem 4.8. The condition $l \geq 2(\tau + 1)$ makes sense where each segment is needed to have at least two characters. For example, if $10 \leq l < 12$, we can tolerate $\tau = 4$ edit operations. If $12 \leq l < 14$, we can tolerate $\tau = 5$ edit operations.

THEOREM 4.8. *If $l \geq 2(\tau + 1)$ and $|s| \geq l$, $\mathcal{W}_m(s, l)$ satisfies minimality.*

PROOF. See Section F in Appendix. □

4.4. Substring-Selection Algorithm

Based on previous discussions, we improve SUBSTRINGSELECTION algorithm by removing unnecessary substrings. For $\mathcal{L}_l^i$, we use the multimatch-aware selection method to select substrings, and the selection complexity is $\mathcal{O}(\tau)$. Figure 6 gives the pseudocode of the substring selection algorithm.

For example, consider the strings in Table I. We create inverted indices as illustrated in Figure 2. Consider string $s_1 =$ "vankatesh" with four segments, we build four inverted lists for its segments {va, nk, at, esh}. Then for $s_2 =$ "avataresha". We use multimatch-aware selection method to select its substrings. Here we only select 8 substrings for s_2 and use the 8 substrings to find similar strings of s_2 from the inverted indices. Similarly, we can select substrings for other strings.

5. IMPROVING THE VERIFICATION STEP

In our framework, for string s and inverted index $\mathcal{L}_l^i$, we generate a set of its substrings $\mathcal{W}(s, \mathcal{L}_l^i)$. For each substring $w \in \mathcal{W}(s, \mathcal{L}_l^i)$, we need to check whether it appears in $\mathcal{L}_l^i$.

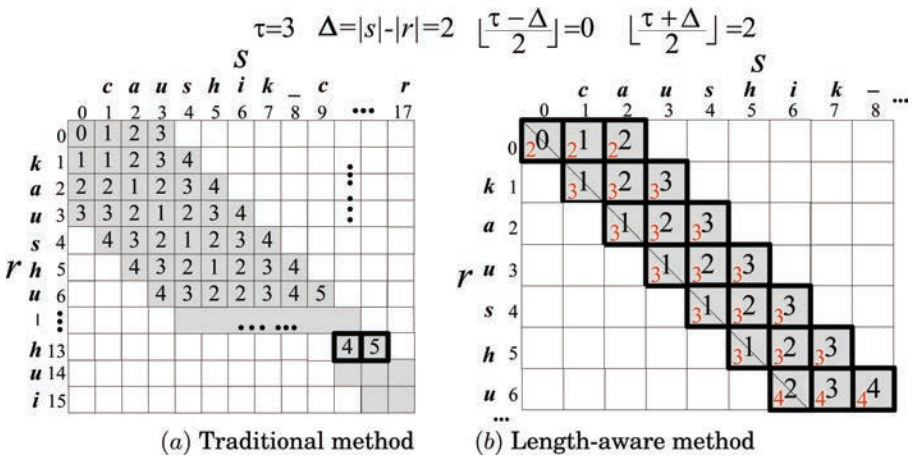


Fig. 7. An example for verification.

If $w \in \mathcal{L}_l^i$, for each string $r \in \mathcal{L}_l^i(w)$, $\langle r, s\rangle$ is a candidate pair and we need to verify the candidate pair to check whether they are similar. In this section we propose effective techniques to do efficient verification.

Traditional Method. Given a candidate pair $\langle r, s\rangle$, a straightforward method to verify the pair is to use a dynamic-programming algorithm to compute their real edit distance. If the edit distance is not larger than τ, the pair is an answer. We can use a matrix M with $|r|+1$ rows and $|s|+1$ columns to compute their edit distance, in which $M(0, j) = j$ for $0 \le j \le |s|$, and $M(i, 0) = i$ for $1 \le i \le |r|$,

$$M(i, j) = \min(M(i-1, j) + 1, M(i, j-1) + 1, M(i-1, j-1) + \delta)$$

where $\delta = 0$ if the i-th character of r is the same as the j-th character of s; otherwise $\delta = 1$. The time complexity of the dynamic-programming algorithm is $\mathcal{O}(|r| * |s|)$.

Actually, we do not need to compute their real edit distance and only need to check whether their edit distance is not larger than τ. An improvement based on length pruning [Ukkonen 1985] is proposed that computes only the values $M(i, j)$ for $|i-j| \le \tau$, as shown in the shaded cells of Figure 7(a). The basic idea is that if $|i-j| > \tau$, $M(i, j) > \tau$, and we do not need to compute such values. This method improves the time complexity $\mathcal{V}(s, r)$ to $\mathcal{O}((2 * \tau + 1) * \min(|r|, |s|))$. Next, we propose a technique to further improve the performance by considering the length difference between r and s.

5.1. Length-Aware Verification

In this section, we propose a length-aware verification method. We first use an example to illustrate our idea. Consider string $r =$ "`kaushuk chadhui`" and string $s =$ "`caushik chakrabar`". Suppose $\tau = 3$. Existing methods need to compute all the shaded values in Figure 7(a). We have an observation that we do not need to compute $M(2, 1)$, which is the edit distance between "`ka`" and "`c`". This is because if there is a transformation from r to s by first transforming "`ka`" to "`c`" with at least 1 edit operation (length difference) and then transforming "`ushuk chadhui`" to "`aushik chakrabar`" with at least 3 edit operations (length difference), the transformation distance is at least 4, which is larger than $\tau = 3$. In other words, even if we do not compute $M(2, 1)$, we know that there is no transformation including $M(2, 1)$ (the transformation from "`ka`" to "`c`") whose distance is not larger than τ. Actually we only need to compute the highlighted values as illustrated in Figure 7(b). Next we formally introduce our length-aware method.

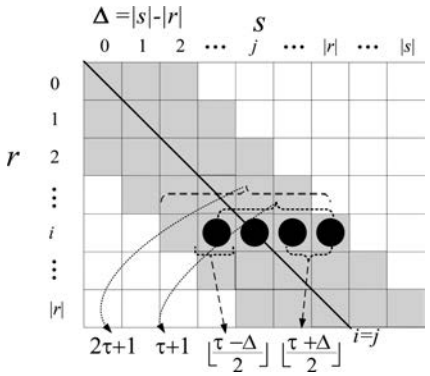


Fig. 8. Length-aware verification.

Length-aware Method. Without loss of generality, let $|s| \geq |r|$ and $\triangle = |s| - |r| \leq \tau$ (otherwise their edit distance must be larger than τ). We call a transformation from r to s including $M(i, j)$, if the transformation first transforms the first i characters of r to the first j characters of s with d_1 edit operations and then transforming the other characters in r to the other characters in s with d_2 edit operations. Based on length difference, we have $d_1 \geq |i - j|$ and $d_2 \geq |(|s| - j) - (|r| - i)| = |\triangle + (i - j)|$. If $d_1 + d_2 > \tau$, we do not need to compute $M(i, j)$, since the distance of any transformation including $M(i, j)$ is larger than τ. To check whether $d_1 + d_2 > \tau$, we consider the following cases.

—If $i \geq j$, $d_1 + d_2 \geq i - j + \triangle + i - j$. If $i - j + \triangle + i - j > \tau$, that is $j < i - \frac{\tau - \triangle}{2}$, we do not compute $M(i, j)$. In other words, we only need to compute $M(i, j)$ with $j \geq i - \frac{\tau - \triangle}{2}$.
—If $i < j$, $d_1 = j - i$. If $j - i \leq \triangle$, $d_1 + d_2 \geq j - i + \triangle - (j - i) = \triangle$. As $\triangle \leq \tau$, there is no position constraint. We need to compute $M(i, j)$; otherwise if $j - i > \triangle$, we have $d_1 + d_2 \geq j - i + j - i - \triangle$. If $j - i + j - i - \triangle > \tau$, that is $j > i + \frac{\tau + \triangle}{2}$, we do not need to compute $M(i, j)$. In other words, we only need to compute $M(i, j)$ with $j \leq i + \frac{\tau + \triangle}{2}$.

Based on this observation, for each row $M(i, *)$, we only compute $M(i, j)$ for $i - \lfloor \frac{\tau - \triangle}{2} \rfloor \leq j \leq i + \lfloor \frac{\tau + \triangle}{2} \rfloor$. For example, in Figure 8, we only need to compute the values in black circles. Thus we can improve the time complexity $\mathcal{V}(s, r)$ from $\mathcal{O}((2\tau + 1) * \min(|r|, |s|))$ to $\mathcal{O}((\tau + 1) * \min(|r|, |s|))$.

Early Termination. We can further improve the performance by doing an early termination. Consider the values in row $M(i, *)$. A straightforward early-termination method is to check each value in $M(i, *)$, and if all the values are larger than τ, we can do an early termination. This is because the values in the following rows $M(k > i, *)$ must be larger than τ based on the dynamic-programming algorithm. This pruning technique is called *prefix pruning*. For example in Figure 7(a), if $\tau = 3$, after we have computed $M(13, *)$, we can do an early termination as all the values in $M(13, *)$ are larger than τ. But in our method, after we have computed the values in $M(6, *)$, we can conclude that the edit distance between the two strings is at least 4 (larger than $\tau = 3$). Thus we do not need to compute $M(i > 6, *)$ and can terminate the computation as shown in Figure 7(b). To this end, we propose a novel early-termination method.

For ease of presentation, we first introduce several notations. Given a string s, let $s[i]$ denote the i-th character and $s[i : j]$ denote the substring of s from the i-th character to the j-th character. Notice that $M(i, j)$ denotes the edit distance between $r[1 : i]$ and $s[1 : j]$. We can estimate the lower bound of the edit distance between $r[i : |r|]$ and $s[j : |s|]$ using their length difference $|(|s| - j) - (|r| - i)|$. We use $E(i, j) = M(i, j) + |(|s| - j) - (|r| - i)|$

ALGORITHM 3: LENGTHAWAREVERIFICATION (r, s, τ)

Input: r: A string; s: Another string; τ: Threshold;
Output: $d = \min(\tau + 1, \text{ED}(s, r))$

```
1 begin
2     △ = |s| − |r| ;
3     for i = 1 to |r| do
4         st = i − ⌊(τ−△)/2⌋; en = i + ⌊(τ+△)/2⌋ ;
5         for j = st to en do
6             M(i, j) = min(M(i − 1, j) + 1, M(i, j − 1) + 1, M(i − 1, j − 1) + δ) ;
7             E(i, j) = M(i, j) + |(|s| − j) − (|r| − i)| ;
8         if E(i, j) > τ for st ≤ j ≤ en then return τ + 1 ;
9     return M[|r|][|s|] ;
```

Fig. 9. Length-aware verification algorithm.

to estimate the edit distance between s and r, which is called *expected edit distance* of s and r with respect to $M(i, j)$. If each expected edit distance for $M(i, j)$ in $M(i, *)$ is larger than τ, the edit distance between r and s must be larger than τ, thus we can do an early termination. To achieve our goal, for each value $M(i, j)$, we maintain the expected edit distance $E(i, j)$. If each value in $E(i, *)$ is larger than τ, we can do an early termination as formalized in Lemma 5.1.

LEMMA 5.1. *Given strings s and r, if each value in $E(i, *)$ is larger than τ, the edit distance of r and s is larger than τ.*

PROOF. We prove that any transformation from r to s will involve more than τ edit operations if each value in $E(i, *)$ is larger than τ. For any transformation $\mathcal{T}$ from r to s, $\mathcal{T}$ must include one of $M(i, *)$. Without loss of generality, suppose $\mathcal{T}$ includes $M(i, j)$. Then we have $d_1 = M(i, j)$ and $d_2 \geq |(|s| - j) - (|r| - i)|$. Thus $|\mathcal{T}| = d_1 + d_2 \geq M(i, j) + |(|s| - j) - (|r| - i)| = E(i, j) > \tau$. Thus transformation $\mathcal{T}$ will involve more than τ edit operations. Therefor the edit distance of r and s is larger that τ. □

Figure 9 shows the pseudocode of the length-aware algorithm. Different from traditional methods, for each row $M[i][*]$, we only compute the columns between $i - \lfloor \frac{\tau - \triangle}{2} \rfloor$ and $i + \lfloor \frac{\tau + \triangle}{2} \rfloor$ (lines 4–6). We also use the expected matrix to do early termination (lines 7-8). Next we use an example to walk through our algorithm. In Figure 7(b), the expected edit distances are shown in the left-bottom corner of each cell. When we have computed $M(6, *)$ and $E(6, *)$, all values in $E(6, *)$ are larger than 3, thus we can do an early termination and avoid many unnecessary computations.

We use the length-aware verification algorithm to improve the Verification function in Figure 3 (by replacing line 3). Our technique can be applied to any other algorithms that need to verify a candidate in terms of edit distance (e.g., ED-JOIN and NGPP).

5.2. Extension-Based Verification

Consider a selected substring w of string s. If w appears in the inverted index $\mathcal{L}_l^i$, for each string r in the inverted list $\mathcal{L}_l^i(w)$, we need to verify the pair $\langle s, r \rangle$. As s and r share a common segment w, we can use the shared segment to efficiently verify the pair. To achieve our goal, we propose an extension-based verification algorithm.

As r and s share a common segment w, we partition them into three parts based on the common segment. We partition r into three parts, the left part r_l, the matching part $r_m = w$, and the right part r_r. Similarly, we get three parts for string s: s_l, $s_m = w$, and s_r. Here we align s and r based on the matching substring r_m and s_m, and we only

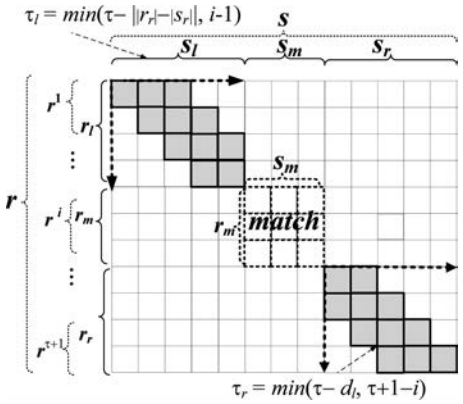


Fig. 10. Extension-based verification.

need to verify whether r and s are similar in this alignment. Thus we first compute the edit distance $d_l = \text{ED}(r_l, s_l)$ between r_l and s_l using the aforementioned method. If d_l is larger than τ, we terminate the computation; otherwise, we compute the edit distance $d_r = \text{ED}(s_r, r_r)$ between s_r and r_r. If $d_l + d_r$ is larger than τ, we discard the pair; otherwise we take it as an answer.

Note that this method can correctly verify a candidate pair. Here we present the basic idea and will formally prove it in Theorem 5.3. Recall Lemma 3.1. If s and r are similar, s must have a substring that matches a segment of r. In addition, based on dynamic-programming algorithm, there must exist a transformation by aligning r_m with s_m and $\text{ED}(s, r) = d_l + d_r$. As our method selects all possible substrings and considers all such common segments, our method will not miss any results. On the other hand, the results found in our algorithm satisfy $d_l + d_r \leq \tau$. Since $\text{ED}(s, r) \leq d_l + d_r \leq \tau$, the results found in our algorithm must be true answers.

Improve the Verification Algorithm Using Tighter Bounds. Actually, we can further improve the verification algorithm. For the left parts, we can give a tighter threshold $\tau_l \leq \tau$. The basic idea is as follows. As the minimal edit distance between the right parts r_r and s_r is $||r_r| - |s_r||$. Thus we can set $\tau_l = \tau - ||r_r| - |s_r||$. If the edit distance between r_l and s_l is larger than threshold τ_l, we can terminate the verification; otherwise we continue to compute $d_r = \text{ED}(r_r, s_r)$. Similarly for the right parts, we can also give a tighter threshold $\tau_r \leq \tau$. As d_l has been computed, we can use $\tau_r = \tau - d_l$ as a threshold to verify whether r_r and s_r are similar. If d_r is larger than threshold τ_r, we can terminate the verification.

For example, suppose $\tau = 3$ and we want to verify $s_5 =$ "`kausic chakduri`" and $s_6 =$ "`caushik chakrabar`". s_5 and s_6 share a segment "`chak`". We have $s_{5_l} =$ "`kausic_`" and $s_{6_l} =$ "`caushik_`", and $s_{5_r} =$ "`duri`" and $s_{6_r} =$ "`rabar`". As $||s_{5_r}| - |s_{6_r}|| = 1$, $\tau_l = \tau - 1 = 2$. We only need to verify whether the edit distance between s_{5_l} and s_{6_l} is not larger than $\tau_l = 2$. After we have computed $M(6, *)$, we can do an early termination as each value in $E(6, *)$ is larger than 2.

Actually we can deduce two much tighter thresholds for τ_l and τ_r respectively. Consider the i-th segment, we can terminate the verification based on the multi-match-aware method. Thus we have $d_l \leq \tau_l = i - 1$. Combining with the given pruning condition, we have $\tau_l = \min(\tau - ||r_r| - |s_r||, i - 1)$. As $||r_r| - |s_r|| = |(|r| - p_i - l_i) - (|s| - p - l_i)| = |p - p_i - \triangle| \leq \tau + 1 - i$ (based on the multimatch-aware method), $\tau - ||r_r| - |s_r|| \geq i - 1$. we set $\tau_l = i - 1$.

ALGORITHM 4: ExtensionBasedVerification$(s, \mathcal{L}_l^i(w), \tau)$

Input: s: A string; $\mathcal{L}_l^i(w)$: Inverted list; τ: Threshold
Output: $\mathcal{A} = \{(s, r \in \mathcal{L}_l^i(w)) | \text{ED}(s, r) \leq \tau\}$

```
1 begin
2     τ_l = i − 1, τ_r = τ + 1 − i ;
3     for r ∈ L_l^i(w) do
4         if ⟨r, s⟩ is in A then continue;
5         d_l = LengthAwareVerification(r_l, s_l, τ_l) ;
6         if d_l ≤ τ_l then
7             d_r = LengthAwareVerification(r_r, s_r, τ_r) ;
8             if d_r ≤ τ_r then A ← ⟨r, s⟩;
```

Fig. 11. Extension-based verification algorithm.

We can get similar conclusion from the right-side perspective. If $d_r \geq \tau + 1 - i$, we can terminate the verification based on the multimatch-aware method from the right-side perspective. Thus we have $\tau_r = \min(\tau - d_l, \tau + 1 - i)$. As $d_l \leq \tau_l \leq i - 1$, $\tau - d_l \geq \tau - (i - 1)$. Thus we set $\tau_r = \tau + 1 - i$.

Also we can use these two tighter thresholds simultaneously. That is for any substring $s_m \in \mathcal{W}_m(s, l)$ of s that matches the i-th segment r_m of r, we only need to check whether $\text{ED}(r_l, s_l) \leq i - 1$ and $\text{ED}(r_r, s_r) \leq \tau + 1 - i$ using the length-aware method. If so, we can say that r and s are similar and output $\langle r, s \rangle$ as an answer.

Based on our proposed techniques, we improve the Verification function. Figure 11 illustrates the pseudocode. Consider a string s, a selected substring w, and an inverted list $\mathcal{L}_l^i(w)$. For each $r \in \mathcal{L}_l^i(w)$, we use the extension-based method to verify the candidate pair $\langle s, r \rangle$ as follows. We first compute $\tau_l = i - 1$ and $\tau_r = \tau + 1 - i$ (line 2). Then for each $r \in \mathcal{L}_l^i(w)$, we compute the edit distance (d_l) between r_l and s_l with the tighter bound τ_l using the length-aware verification method (line 5). If $d_l > \tau_l$, we terminate the verification; otherwise we verify whether s_r and r_r are similar with threshold τ_r using the length-aware verification method (line 7).

To guarantee correctness of our extension-based method, we first give a formal definition of correctness.

Definition 5.2 (*Correctness*). Given a candidate pair $\langle s, r \rangle$, a verification algorithm is correct, if it satisfies (1) If $\langle s, r \rangle$ passes the algorithm, $\langle s, r \rangle$ must be a similar pair; and (2) If $\langle s, r \rangle$ is a similar pair, it must pass the algorithm.

Our extension-based method satisfies correctness as stated in Theorem 5.3.

Theorem 5.3. *Our extension-based verification method satisfies correctness.*

Proof. See Section G in Appendix. □

5.3. Iterative-Based Verification

In this section, we introduce an iterative-based verification method to further improve the verification step. Instead of verifying a candidate pair with a matching segment/substring using the extension-based verification method, we can iteratively apply our multimatch-aware technique on the left and right part of the matching segment/substring to filter this candidate pair. We first present the basic idea, then give the pseudocode, and finally discuss the technical details.

Basic Idea. Consider two strings r and s where s has a selected substring that matches r's i-th segment w. We still partition r/s into three parts, the left part r_l/s_l, the matching part $r_m/s_m = w$ and the right part r_r/s_r. Instead of checking whether

ALGORITHM 5: IterativeVerification (r, s, w, τ)

Input: r: A string; s: Another string; w: Common segment; τ: Threshold;
Output: $d = \min((\tau + 1), \text{ED}(s, r))$

```
1  begin
2      Compute r_l/s_l and r_r/s_r based on w ; τ_l = i − 1, τ_r = τ + i − 1;
3      if IterativeVerify(r_l, s_l, τ_l, i, left)==pass then
4          d_l = LengthAwareVerification (r_l, s_l, τ_l) ;
5          if IterativeVerify(r_r, s_r, τ_r, i, right)==pass then
6              d_r = LengthAwareVerification (r_r, s_r, τ_r);
7              return d_l + d_r ;
8      return τ + 1 ;
```

Function IterativeVerify(r', s', τ', i, f)

Input: r': A string; s': Another string; τ': A threshold; i: An integer; f: left or right;
Output: pass or fail

```
1  begin
2      if the i-th segment is the first matching segment then
3          Partition r' into τ' + 1 segments (c_1c_2 ··· c_x, c_{x+1} ··· c_y and the last τ − i segments of r'
           if f is right; the first i − 2 segments of r', c_1c_2 ··· c_{x−1}, and c_x ··· c_y if f is left) ;
4          j = 2 if f is right; j = i − 2 if f is left ;
5          if the j-th segment of r' is not empty then
6              Select substrings of s' on the j-th segment of r' ;
7              if s' has no substring matching the j-th segment of r' then return fail ;
8              else
9                  Suppose s' has a substring w' matching the j-th segment of r'. Partition r'/s'
                   based on w' and suppose the left parts are r'_l/s'_l and the right parts are r'_r/s'_r;
10                 if f is left then return IterativeVerify(r'_l, s'_l, τ' − 1, i − 1, left) ;
11                 else return IterativeVerify(r'_r, s'_r, τ' − 1, i + 1, right) ;
12     return pass;
```

Fig. 12. Iterative-based verification algorithm.

$\text{ED}(r_l, s_l) \leq \tau_l = i - 1$ and $\text{ED}(r_r, s_r) \leq \tau_r = \tau + i - 1$ using the length-aware verification technique, we iteratively use the multimatch-aware technique to check whether $r_l(r_r)$ and $s_l(s_r)$ are similar. Without loss of generality, consider the left parts r_l and s_l. We partition r_l into $\tau_l + 1$ segments. If s_l has no selected substring that matches a segment of r_l, r and s cannot be similar and we can prune the pair.

For example consider a string r = "kausic_chakduri" with four segments "kau", "sic_", "chak", and "duri" and another string s = "caushik_chakrabar". String s has a substring "chak" matching with the third segment of string r. Thus r_l = "kausic_" and s_l = "caushik_". The extension-based verification will compute their edit distance using the tighter bound $\tau_l = i - 1 = 2$. Actually we need not compute their real edit distance using the dynamic-programming method. Instead, we partition r_l into $\tau_l + 1 = 3$ segments "kau", "si", and "c_". Based on the multimatch-aware substring selection method, we only select four substrings of s_l, "cau", "sh", "hi" and "k_". As none of the four substrings matches any segment of r_l, we deduce that the edit distance between s_l and r_l is larger than $\tau_l = 2$. Thus we can prune the pair of s and r.

Pseudocode. Figure 12 shows the pseudocode of our iterative-based method. It first verifies the left parts by calling subroutine IterativeVerify (line 3). If the left-part verification passes, it verifies the right parts by calling subroutine IterativeVerify (line 5) again. If the verifications on the both parts passes, it returns the real edit

distance (line 8). IterativeVerify first partitions the input strings into $\tau' + 1$ segments (line 3). It employs different partition strategies for the leaf part and the right part, which will be discussed later. Then it selects substrings based on the left part or the right part (line 6). If there is no selected substring matching the segment, it returns `fail` (line 7); otherwise it iteratively calls itself to verify the candidate pair (line 11).

To use the iterative-based method in the verification step, we only need to replace lines 5-8 in Figure 9 with the IterativeVerification algorithm.

Technical Details of The Iterative-based Method. We formally introduce how to use the iterative-based method to verify r and s, that is, how to implement the IterativeVerify function. Suppose s has a selected substring s_m that matches the i-th segment (r_m) of string r, and the left parts are r_l/s_l and right parts are r_r/s_r. We respectively discuss how to iteratively verify the left parts (r_l/s_l) and right parts (r_r/s_r).

Left Parts. We consider r_l/s_l and check whether $\text{ED}(r_l, s_l) \leq \tau_l = i - 1$. If s_m is not the first selected substring of s (with the minimum start position) that matches a segment of r, we still use the length-aware method; otherwise we use our iterative-based method. (Notice that the matching substring s_m has a very large probability (larger than 90% in our experiments) to be the first substring). The iterative-based method first partitions r_l into $\tau_l + 1 = i$ segments. Then it uses the multimatch-aware method to select substrings from s_l. If s_l has a selected substring matching a segment of r_l, we iteratively call the iterative-based method; otherwise we prune $\langle r, s \rangle$.

Next we discuss how to partition r_l into $\tau_l + 1 = i$ segments. As r_m is the i-th segment of r, r_l contains the first $i - 1$ segments of r. Since s_m is the first selected substring that matches the i-th segment of r, s has no selected substring that matches the first $i - 1$ segments. More interestingly we find that s_l also has no selected substring that matches the first $i - 1$ segments (which will be proved in Theorem 5.4).[2] Thus we keep the first $i - 2$ segments of r as the first $i - 2$ segments of r_l. In this way, we know that s_l has no substring that matches the first $i - 2$ segments of r_l. Then we partition the $(i - 1)$-th segment of r into two segments and take them as the last 2 segments of r_l as follows. Let $c_1c_2 \cdots c_x \cdots c_y$ denote the $(i - 1)$-th segment of r and $s_l = c'_1c'_2 \cdots c'_{x'} \cdots c'_{y'}$. We compute the longest common suffix of $c_1c_2 \cdots c_x \cdots c_y$ and s_l. Suppose $c_{x+1} \cdots c_y = c'_{x'+1} \cdots c'_{y'}$ is the longest common suffix. If $x > 1$, we partition the $(i - 1)$-th segment into two segments $c_1c_2 \cdots c_{x-1}$ and $c_x \cdots c_y$. Thus we can partition r_l into i segments. Based on the multimatch-aware method, s_l has no selected substring that matches the i-th segments of r_l as $c_x \neq c'_{x'}$. Thus we only need to select substrings from s_l for the $(i - 1)$-th segment of r_l (e.g., $c_1 \cdots c_{x-1}$). We check whether the selected substrings match the $(i - 1)$-th segment of r_l. If yes, we iteratively call the iterative-based method on the left parts of the matching segments/substrings; otherwise we prune the pair of r and s. Notice that if $x = 1$, we cannot partition the $(i - 1)$-th segment into two segments and we still use the length-aware verification method to verify r_l and s_l.

Right Parts. If $\text{ED}(r_l, s_l) \leq \tau_l = i - 1$, we continue to verify the right parts r_r/s_r similarly. The differences are as follows. First we partition r_r into $\tau_r + 1 = \tau + 2 - i$ segments. Second the partition strategy is different. Next we discuss how to partition r_r into $\tau + 2 - i$ segments. Notice that if s has another substring that matches a segment among the last $\tau - i + 1$ segments of r, we can discard r_m and s_m and verify r and s using the next matching pair based on the multimatch-aware technique. Thus we keep the last $\tau - i$ segments of r as the last $\tau - i$ segments of r_r and we do not need to select substrings from s_r to match such segments. Then we repartition the $(i + 1)$-th

[2]As $|s_l| - |r_l|$ may be unequal to $|s| - |r|$, the selected substrings of s_l for segments of r_l may be different from those selected substrings of s_l (as a part of s) for segments of r. Thus we need to prove the statement.

segment of r into the first two segments of r_r as follows. We find the longest common prefix of the $(i+1)$-th segment of r and s_r and then partition $(i+1)$-th segment of r into two segments similarly. Let $c_1c_2\cdots c_x\cdots c_y$ denote the $(i+1)$-th segment of r and $s_r = c'_1c'_2\cdots c'_{x'}\cdots c'_{y'}$. We compute the longest common prefix of $c_1c_2\cdots c_x\cdots c_y$ and s_r. Suppose $c_1\cdots c_{x-1} = c'_1\cdots c'_{x'-1}$ is the longest common prefix. If $x < y$, we partition the $(i+1)$-th segment into two segments $c_1c_2\cdots c_x$ and $c_{x+1}\cdots c_y$ and take them as the first two segments of r_r. Thus we can partition r_r into $\tau+2-i$ segments. Based on the multimatch-aware method, we only need to select substrings from s_r for the second segment of r_r (e.g., $c_{x+1}\cdots c_y$) and check whether the selected substrings match the second segment. If yes, we iteratively call the iterative-based method on the right parts of the matching segments/substrings; otherwise we prune the pair of r and s. Notice that if $x = y$, we cannot partition the $(i+1)$-th segment into two segments and we still use the length-aware verification method to verify r_r and s_r.

As verifying the left parts is similar to verifying right parts, we combine them and use the IterativeVerify function to verify them. In the function, r'/s' refer to r_l/s_l or r_r/r_s as show in Figure 12. We use a flag to distinguish the left parts or the right parts. For the left parts we keep the first $i-2$ segments and split the $i-1$-th segment into two new segments and for the right parts we split the first segment into two new segment and keep the last $\tau-i$ segments. We select the substrings based on the left parts or right parts. Then we can use the segments and selected substrings to do pruning.

The iterative-based verification method satisfies the correctness as stated in Theorem 5.4.

Theorem 5.4. *Our iterative-based verification method satisfies the correctness.*

Proof. See Section H in Appendix. □

5.4. Correctness and Completeness

We prove correctness and completeness of our algorithm as formalized in Theorem 5.5.

Theorem 5.5. *Our algorithm satisfies the (1) completeness: Given any similar pair $\langle s, r\rangle$, our algorithm must find it as an answer; and (2) correctness: A pair $\langle s, r\rangle$ found in our algorithm must be a similar pair.*

Proof. See Section I in Appendix. □

6. DISCUSSIONS

In this section we first discuss how to support normalized edit distance (Section 6.1) and then extend our techniques to support R-S join (Section 6.2).

6.1. Supporting Normalized Edit Distance

Normalized edit distance, aka edit similarity, is also a widely used similarity function to quantify the similarity of two strings. The normalized edit distance of two strings r and s is defined as $\text{NED}(r,s) = 1 - \frac{\text{ED}(r,s)}{max(|r|,|s|)}$. For example, NED("kausic chakduri", "*kaushuk chadhui*") $= \frac{11}{17}$. Given a normalized edit distance threshold δ, we say two strings are similar if their normalized edit distance is not smaller than δ. Then we formalize the problem of string similarity join with normalized edit distance constraint as follows.

Definition 6.1 (*String Similarity Joins With Normalized Edit Distance Constraint*). Given two sets of strings $\mathcal{R}$ and $\mathcal{S}$ and an normalized edit-distance threshold δ, it finds all similar string pairs $\langle r, s\rangle \in \mathcal{R}\times\mathcal{S}$ such that $\text{NED}(r,s) \geq \delta$.

Next we discuss how to support normalize edit distance. For two strings r and s, as $\text{NED}(r,s) = 1 - \frac{\text{ED}(r,s)}{max(|r|,|s|)}$, $\text{ED}(r,s) = max(|r|,|s|)\cdot(1-\text{NED}(r,s))$. If $\text{NED}(r,s) \geq \delta$, $\text{ED}(r,s) =$

ALGORITHM 6: SEGFILTER-NED $(\mathcal{S}, \delta)$

Input: $\mathcal{S}$: A collection of strings
δ: A given normalized edit-distance threshold
Output: $\mathcal{A} = \{(s \in \mathcal{S}, r \in \mathcal{S}) \mid \text{NED}(s, r) \geq \delta\}$

```
1  begin
2      Sort S by string length in descending order;
3      for s ∈ S do
4          for |s| ≤ l ≤ ⌊|s|/δ⌋ do
5              τ = ⌊(1 − δ) · l⌋ ;
6              for L_l^i (1 ≤ i ≤ τ + 1) do
7                  W(s, L_l^i) = SUBSTRINGSELECTION(s, L_l^i);
8                  for w ∈ W(s, L_l^i) do
9                      if w is in L_l^i then VERIFICATION(s, L_l^i(w), τ);
10     Partition s into ⌊(1 − δ) · |s|⌋ + 1 segments and add them into L_|s|^i;
```

Fig. 13. SEGFILTER-NED algorithm.

$max(|r|, |s|) \cdot (1 - \text{NED}(r, s)) \leq max(|r|, |s|) \cdot (1 - \delta)$. Notice that in the index phase we need to partition string r into $\tau + 1$ segments. If $|s| > |r|$, we cannot determine the number of segments. To address this issue, we first index the long strings (r) and then visit the short strings (s). That is we index segments of the long strings and select substrings from the short strings. In this case, we always have $|s| \leq |r|$. Thus $\text{ED}(r, s) \leq |r| \cdot (1 - \delta)$. Let $\tau = |r| \cdot (1 - \delta)$, we can partition r to $\tau + 1 = \lfloor |r| \cdot (1 - \delta) \rfloor + 1$ segments using the even partition scheme. In addition, as $|r| - |s| \leq \text{ED}(r, s) \leq |r| \cdot (1 - \delta)$, we have $|r| \leq \lfloor \frac{|s|}{\delta} \rfloor$. Thus for string s, we only need to find candidates for strings with length between $|s|$ and $\lfloor \frac{|s|}{\delta} \rfloor$. The substring selection phase and verification phase are still the same as the original method.

Figure 13 gives the pseudocode SEGFILTER-NED to support normalized edit distance. We first sort strings in $\mathcal{S}$ by string length in descending order (line 2) and then visit each string s in sorted order (line 3). For each possible length ($[|s|, \lfloor \frac{|s|}{\delta} \rfloor]$) of strings that may be similar to s (line 4), we transform the normalized edit distance threshold δ to edit distance threshold τ (line 5). Then for each inverted index $\mathcal{L}_l^i (1 \leq i \leq \tau + 1)$ (line 6), we select the substrings of s (line 7) and check whether each selected substring w is in $\mathcal{L}_l^i$ (line 8). If yes, for any string r in the inverted list $\mathcal{L}_l^i(w)$, the string pair $\langle r, s \rangle$ is a candidate pair. We verify the pair (line 9). Finally, we partition s into $\lfloor (1 - \delta) \cdot |s| \rfloor + 1$ segments, and insert the segments into the inverted index $\mathcal{L}_{|s|}^i (1 \leq i \leq \lfloor (1 - \delta) \cdot |s| \rfloor + 1)$ (line 10). Here algorithms SUBSTRINGSELECTION and VERIFICATION are the same as the algorithms in Figure 3.

Next we give a running example of our SEGFILTER-NED algorithm (Figure 14). Consider the string set in Table I and suppose the normalized edit distance threshold $\delta = 0.82$. We sort the strings in descending order as show in Table I(c). For the first string s_6, we partition it to $\lfloor (1 - \delta) \cdot |s_6| \rfloor + 1 = 4$ segments and insert the segments into $\mathcal{L}_{|s_6|}$. Next for s_5 we select substrings for $\mathcal{L}_{|s_6|}$ using the multimatch-aware method and check if there is any selected substring matching with its corresponding segment. Here we find "chak" and the pair $\langle s_6, s_5 \rangle$ is a candidate. Then we verify this pair using the iterative-based method based on the matching part "chak" and it is not a result. Next we partition s_5 into $\lfloor (1 - \delta) \cdot |s_5| \rfloor + 1 = 3$ segments. Similarly we repeat these steps and we find another two candidate pairs $\langle s_3, s_4 \rangle$ and $\langle s_3, s_6 \rangle$. We verify them using the iterative-based method and get a final result $\langle s_3, s_6 \rangle$.

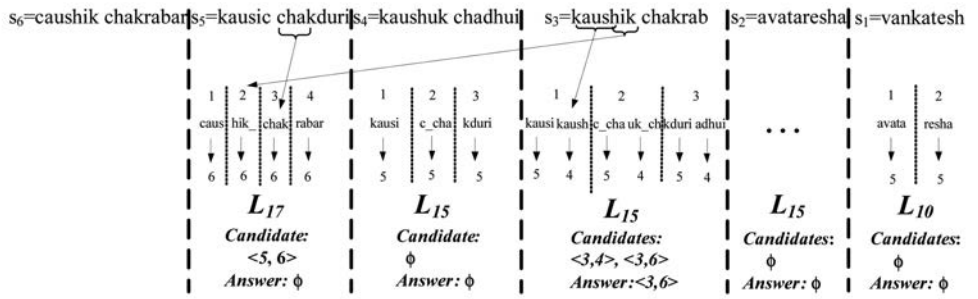


Fig. 14. An example of SEGFILTER-NED algorithm.

ALGORITHM 7: SEGFILTER-RSJOIN $(\mathcal{R}, \mathcal{S}, \tau)$

Input: $\mathcal{R}$: A collection of strings
$\mathcal{S}$: Another collection of strings
τ: A given edit-distance threshold
Output: $\mathcal{A} = \{(r \in \mathcal{R}, s \in \mathcal{S}) \mid \text{ED}(r, s) \leq \tau\}$

```
1 begin
2     Sort R and S by string length in ascending order;
3     for r ∈ R do
4         Partition r and add its segments into L^i_{|r|};
5     for s ∈ S do
6         for L^i_l (|s| − τ ≤ l ≤ |s| + τ, 1 ≤ i ≤ τ + 1) do
7             W(s, L^i_l) = SubstringSelection(s, L^i_l);
8             for w ∈ W(s, L^i_l) do
9                 if w is in L^i_l then Verification(s, L^i_l(w), τ);
```

Fig. 15. SEGFILTER-RSJOIN algorithm.

6.2. Supporting R-S Join

To support R-S join on two sets $\mathcal{R}$ and $\mathcal{S}$, we first sort the strings in the two sets respectively. Then we index the segments of strings in a set, for instance, $\mathcal{R}$. Next we visit strings of $\mathcal{S}$ in order. For each string $s \in \mathcal{S}$ with length $|s|$, we use the inverted indices of strings in $\mathcal{R}$ with lengths between $[|s| - \tau, |s| + \tau]$ to find similar pairs. We can remove the indices for strings with lengths smaller than $|s| - \tau$. Finally we verify the candidates. Notice that in Section 4, for two strings r and s, we only consider the case that $|r| \geq |s|$ where we partition r to segments and select substrings from s. Actually, Theorem 4.4 still holds for $|r| < |s|$.

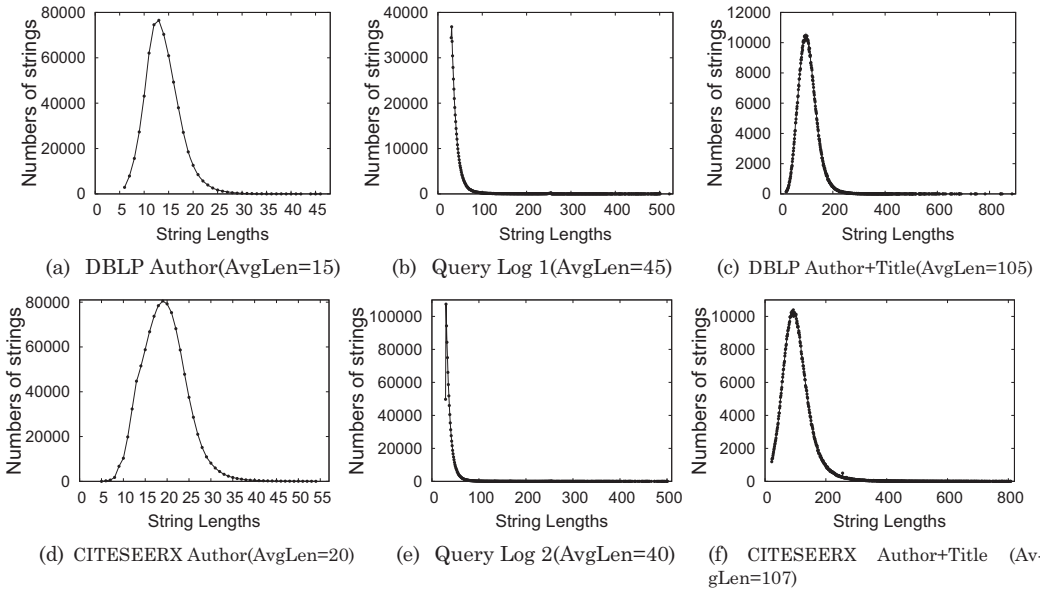
The pseudocode of SEGFILTER-RSJOIN algorithm is illustrated in Figure 15. It first sorts the strings in the two sets (line 2), and then builds indices for strings in $\mathcal{R}$ (lines 3–4). Next it visits strings in $\mathcal{S}$ in sorted order. For each string s, it selects substrings of s by calling algorithm SUBSTRINGSELECTION (line 7) and finds candidates using the indices. Finally it verifies the candidates by calling algorithm VERIFICATION (line 9). Here algorithms SUBSTRINGSELECTION and VERIFICATION are the same as the algorithms in Figure 3.

7. EXPERIMENTAL STUDY

We have implemented our method and conducted an extensive set of experimental studies. We used six real-world datasets. To evaluate self-join, we used three datasets,

Table III. Datasets

Datasets	Cardinality	Avg Len	Max Len	Min Len
DBLP Author	612,781	14.83	46	6
Query Log 1	464,189	44.75	522	30
DBLP Author+Title	863,073	105.82	886	21
Citeseer Author	1,000,000	20.35	54	5
Query Log 2	1,000,000	39.76	501	29
Citeseer Author+Title	1,000,000	107.45	808	22



(a) DBLP Author(AvgLen=15) (b) Query Log 1(AvgLen=45) (c) DBLP Author+Title(AvgLen=105)

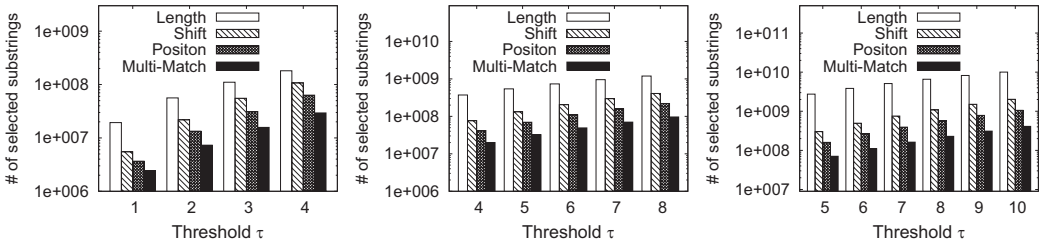
(d) CITESEERX Author(AvgLen=20) (e) Query Log 2(AvgLen=40) (f) CITESEERX Author+Title (AvgLen=107)

Fig. 16. String length distribution.

DBLP Author,[3] DBLP Author+Title, and AOL Query Log 1.[4] DBLP Author is a dataset with short strings, DBLP Author+Title is a dataset with long strings, and the Query Log 1 is a set of query logs. Note that the DBLP Author+Title dataset is the same as that used in ED-JOIN and the DBLP Author dataset is the same as that used in TRIE-JOIN. To evaluate R-S join, we used other three datasets: CITESEERX Author,[5] CITESEERX Author+Title, and AOL Query Log 2. AOL Query Log 2 is another set of query logs that is different from AOL Query Log 1. We joined DBLP Author and CITESEERX author, DBLP Author+Title and CITESEERX Author+Title, and AOL Query Log 1 and AOL Query Log 2. Table III shows the detailed information of the datasets and Figure 16 shows the string length distributions of different datasets.

We compared our algorithms with state-of-the-art methods, ED-JOIN [Xiao et al. 2008a], QCHUNK-JOIN [Qin et al. 2011] and TRIE-JOIN [Wang et al. 2010]. As ED-JOIN, QCHUNK-JOIN and TRIE-JOIN outperform other methods, for instance, Part-Enum [Arasu et al. 2006] and All-Pairs-Ed [Bayardo et al. 2007] (also experimentally shown in [Xiao et al. 2008a; Wang et al. 2010; Qin et al. 2011]), in the article we only compared our

[3]http://www.informatik.uni-trier.de/~ley/db.
[4]http://www.gregsadetsky.com/aol-data/.
[5]http://asterix.ics.uci.edu/fuzzyjoin/.



(a) DBLP Author(AvgLen=15) (b) Query Log 1(AvgLen=45) (c) DBLP Author+Title(AvgLen=105)

Fig. 17. Numbers of selected substrings.

method with them. We downloaded their binary codes from their homepages, ED-JOIN,[6] QCHUNK-JOIN[7] and TRIE-JOIN.[8]

All the algorithms were implemented in C++ and compiled using GCC 4.2.4 with -O3 flag. All the experiments were run on a Ubuntu machine with an Intel Core 2 Quad X5450 3.00GHz processor and 4 GB memory.

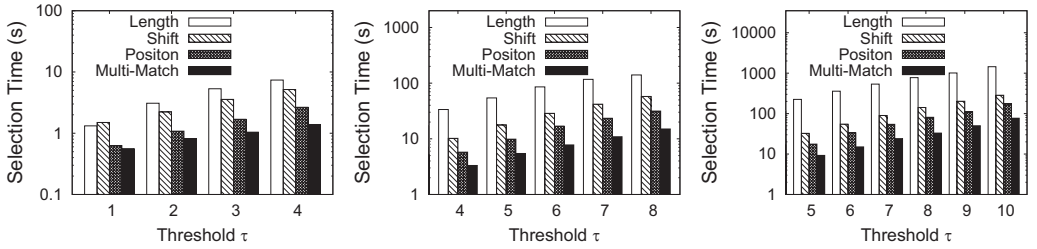
7.1. Evaluating Substring Selection

In this section, we evaluate substring selection techniques. We implemented the following four methods. (1) The length-based selection method, denoted by Length, which selects the substrings with the same lengths as the segments. (2) The shift-based method, denoted by Shift, which selects the substring by shifting $[-\tau, \tau]$ positions as discussed in Section 4. (3) Our position-aware selection method, denoted by Position. (4) Our multimatch-aware selection method, denoted by Multi-match. We first compared the total number of selected substrings. Figure 17 shows the results.

We can see that the Length-based method selected large numbers of substrings. The number of selected substring of the Position-based method was about a tenth to a fourth of that of the Length-based method and a half of the Shift-based method. The Multi-match-based method further reduced the number of selected substrings to about a half of that of the Position-based method. For example, on DBLP Author dataset, for $\tau = 1$, the Length-based method selected 19 million substrings, the Shift-based method selected 5.5 million substrings, the Position-based method reduced the number to 3.7 million, and the Multimatch-based method further deceased the number to 2.4 million. Based on our analysis in Section 4, for strings with l, the length-based method selected $(\tau+1)(|s|+1)-l$ substrings, the shift-based method selected $(\tau+1)(2\tau+1)$ substrings, the position-based method selected $(\tau+1)^2$ substrings, and the multimatch-aware method selected $\lfloor \frac{\tau^2-\triangle^2}{2} \rfloor + \tau + 1$ substrings. If $|s| = l = 15$ and $\tau = 1$, the number of selected substrings of the four methods are respectively 17, 6, 4, and 2. Obviously the experimental results consisted with our theoretical analysis.

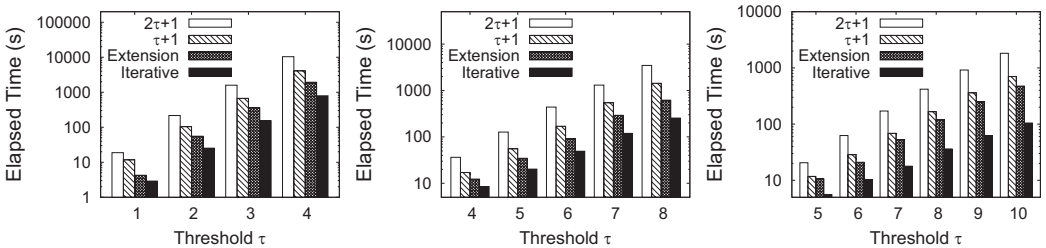
We also compared the elapsed time to generate substrings. Figure 18 shows the results. We see that the Multimatch-based method outperformed the Position-based method, which in turn was better than the Shift-based method and the Length-based method. This is because the elapsed time depended on the number of selected substrings and the Multimatch-based selected the smallest number of substrings.

[6]http://www.cse.unsw.edu.au/~weiw/project/simjoin.html.
[7]http://www.cse.unsw.edu.au/~jqin/.
[8]http://dbgroup.cs.tsinghua.edu.cn/wangjn/.



(a) DBLP Author(AvgLen=15) (b) Query Log 1(AvgLen=45) (c) DBLP Author+Title(AvgLen=105)

Fig. 18. Elapsed time for generating substrings.



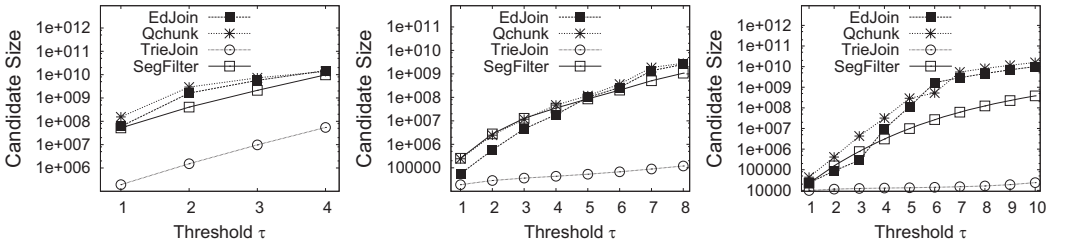
(a) DBLP Author(AvgLen=15) (b) Query Log 1(AvgLen=45) (c) DBLP Author+Title(AvgLen=105)

Fig. 19. Elapsed time for verification.

7.2. Evaluating Verification

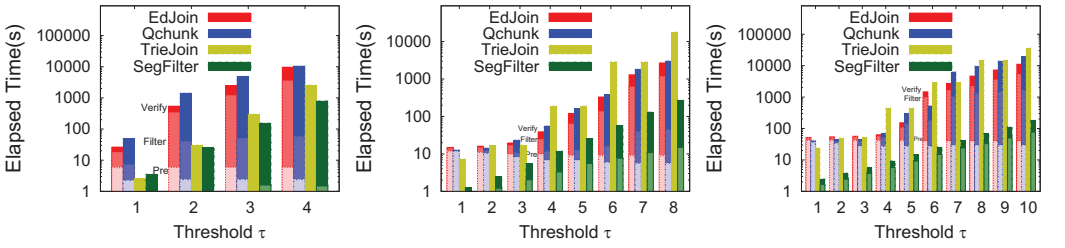
In this section, we evaluate our verification techniques. We implemented four methods. (1) The naive method, denoted by $2\tau + 1$, which computed $2\tau + 1$ values in each row and used the naive early-termination technique (if all values in a row are larger than τ, we terminate). (2) Our length-aware method, denoted by $\tau + 1$, which computed $\tau + 1$ values in each row and used the expected edit distance to do early termination. (3) Our extension-based method, denoted by Extension, which used the extension-based framework. It also computed $\tau+1$ rows and used the expected edit distance with tighter threshold to do early termination. (4) Our iterative-based method, denoted by Iterative, which used the iterative-based verification algorithm. Figure 19 shows the results.

We see that the naive method had the worst performance, as it needed to compute many unnecessary values in the matrix. Our length-aware method was 2–5 times faster than the naive method. This is because our length-aware method can decrease the complexity from $2\tau+1$ to $\tau+1$ and used expected edit distances to do early termination. The extension-based method achieved higher performance and was 2–4 times faster than the length-aware method. The reason is that the extension-based method can avoid the duplicated computations on the common segments and it also used a tighter bound to verify the left parts and the right parts. The Iterative method achieved the best performance, as it can prune dissimilar candidate pairs quickly and avoid many unnecessary computations. For example, on the Query Log 1 dataset, for $\tau = 8$ the naive method took 3, 500 seconds, the length-aware method decreased the time to 1500 seconds, the extension-based method reduced it to 600 seconds, and the Iterative method further improved the time to about 250 seconds. On the DBLP Author+Title dataset, for $\tau = 10$, the elapsed time of the four methods were respectively 1800 seconds, 700 seconds, 475 seconds, and 100 seconds.



(a) DBLP Author(AvgLen=15) (b) Query Log 1(AvgLen=45) (c) DBLP Author+Title(AvgLen=105)

Fig. 20. Comparison of candidate sizes with state-of-the-art methods.



(a) DBLP Author(AvgLen=15) (b) Query Log 1(AvgLen=45) (c) DBLP Author+Title(AvgLen=105)

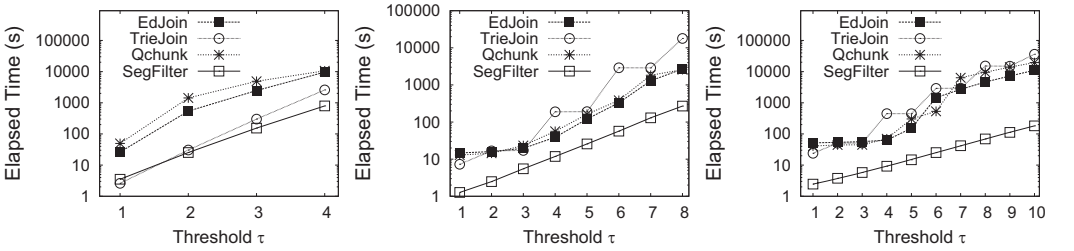
Fig. 21. Comparison of running time of preprocessing, filtering, verification with state-of-the-art methods.

7.3. Comparison with Existing Methods

In this section, we compare our method with state-of-the-art methods ED-JOIN [Xiao et al. 2008a], QCHUNK-JOIN [Qin et al. 2011] and TRIE-JOIN [Wang et al. 2010]. As TRIE-JOIN had multiple algorithms, we reported the best results. For ED-JOIN and QCHUNK-JOIN, we tuned its parameter q and reported the best results. Notice that to avoid involving false negatives, it requires to select a small q for a large edit-distance threshold. As TRIE-JOIN was efficient for short strings, we downloaded the same dataset from TRIE-JOIN homepage (i.e., Author with short strings) and used it to compare with TRIE-JOIN. As ED-JOIN was efficient for long strings, we downloaded the same dataset from ED-JOIN homepage (i.e., Author+Title with long strings) and used it to compare with ED-JOIN.

Candidate Sizes. We first compare the candidate sizes of various methods. Figure 20 shows the results. Notice that TRIE-JOIN directly computed the answers and thus it involved the smallest number of candidates. SEGFILTER generated smaller numbers of candidates than ED-JOIN and QCHUNK-JOIN. This is attributed to our effective substring selection techniques that can prune large numbers of dissimilar pairs. ED-JOIN and QCHUNK-JOIN pruned dissimilar pairs based on the gram-based count filter. SEGFILTER utilized the shared segments to prune dissimilar pairs. Since we can minimize the number of selected substrings and achieve high pruning power, SEGFILTER generates smaller numbers of candidates. For example, on the DBLP Author+Title dataset, SEGFILTER had 1 billion candidates while ED-JOIN and QCHUNK-JOIN had about 10 billion candidates.

Running Time of Different Steps. ED-JOIN and QCHUNK-JOIN includes three steps: preprocessing step, filter step and verification step. The preprocessing step includes tokenizing records into q-grams, generating binary data, and sorting the binary data. SEGFILTER contains two steps: filter step and verification step. TRIE-JOIN directly computes the answers. We compared the running time of each step and Figure 21 shows



(a) DBLP Author(AvgLen=15) (b) Query Log 1(AvgLen=45) (c) DBLP Author+Title(AvgLen=105)

Fig. 22. Comparison of the overall time with state-of-the-art methods.

the results (In the figure, we use different colors to distinguish different steps). For different thresholds, the preprocessing time in ED-JOIN and QCHUNK-JOIN was stable since it only depended on the dataset size. With the increase of the thresholds, the filtering time and the verification time also increased since large thresholds will lead to more results. SEGFILTER involved less filtering time than ED-JOIN and QCHUNK-JOIN, because we only needed to consider smaller numbers of segments and selected substrings while they required to enumerate larger numbers of grams/chunks. SEGFILTER also involved less verification time since it has smaller numbers of candidates and used effective extension-based and iterative-based techniques. Notice that our extension-based and iterative-based verification methods are designed for SEGFILTER and are not applicable for ED-JOIN and QCHUNK-JOIN.

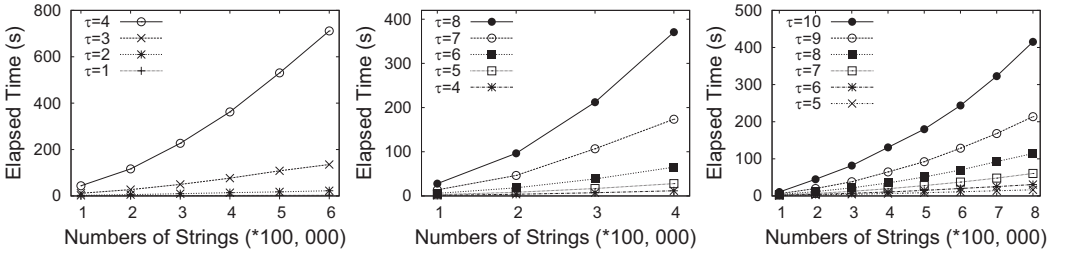
Overall Join Time. We compare the overall time, including preprocessing time, filtering time and verification time. Figure 22 shows the results. On the DBLP Author dataset with short strings, TRIE-JOIN outperformed ED-JOIN and QCHUNK-JOIN, and our method was much better than them, especially for $\tau \geq 2$. The main reason is as follows. ED-JOIN and QCHUNK-JOIN must use a smaller q for a larger threshold. In this way ED-JOIN and QCHUNK-JOIN will involve large numbers of candidate pairs, since a smaller q has rather lower pruning power [Xiao et al. 2008a]. TRIE-JOIN used the prefix filtering to find similar pairs using a trie structure. If a small number of strings shared prefixes, TRIE-JOIN had low pruning power and was expensive to traverse the trie structure. Instead our framework utilized segments to prune large numbers of dissimilar pairs. The segments were selected across the strings and not restricted to prefix filtering. For instance, for $\tau = 4$, TRIE-JOIN took 2500 seconds. SEGFILTER improved it to 700 seconds. ED-JOIN and QCHUNK-JOIN were rather slow and even larger than 10,000 seconds.

On the DBLP Author+Title dataset with long strings, our method significantly outperformed ED-JOIN, QCHUNK-JOIN and TRIE-JOIN, even in 2–3 orders of magnitude. This is because TRIE-JOIN was rather expensive to traverse the trie structures with long strings, especially for large thresholds. ED-JOIN needed to use a mismatch technique and QCHUNK-JOIN needed to use a error estimation-based filtering in verification phase, which were inefficient while our verification method was more efficient than existing ones. For instance, for $\tau = 8$, TRIE-JOIN needed 15,000 seconds, QCHUNK-JOIN took 9500 seconds, ED-JOIN decreased it to 5000 seconds, and SEGFILTER improved the time to 70 seconds.

Index Size. We compared index sizes on three datasets, as shown in Table IV. We can observe that existing methods involve larger indices than our method. For example, on the DBLP Author+Title dataset, ED-JOIN had 335 MB index, TRIE-JOIN used 90 MB, and our method only took 2.1 MB. There are two main reasons. Firstly for each string with length l, ED-JOIN generated $l - q + 1$ grams where q is the gram length, and

Table IV. Index Sizes (MB)

Data Sets	Data Sizes	ED-JOIN $q=4$	TRIE-JOIN	QCHUNK-JOIN $q=4$	SEGFILTER $\tau=2$	SEGFILTER $\tau=4$	SEGFILTER $\tau=6$	SEGFILTER $\tau=8$
DBLP Author	8.7	25.34	16.32	8.06	1.15	1.92	3.49	4.58
Query Log 1	20	72.17	69.65	18.69	2.98	4.96	6.94	8.93
DBLP Author+Title	88	335.24	90.17	23.21	1.26	2.1	2.94	3.78



(a) DBLP Author(AvgLen=15) (b) Query Log 1(AvgLen=45) (c) DBLP Author+Title(AvgLen=105)

Fig. 23. Scalability (Edit Distance).

our method only generated $\tau + 1$ segments. Secondly for a string with length l, we only maintained the indices for strings with lengths between $l - \tau$ and l, and ED-JOIN kept indices for all strings. TRIE-JOIN needed to use a trie structure to maintain strings, which had overhead to store the strings (e.g., pointers to children and indices for searching children with a given character).

7.4. Scalability

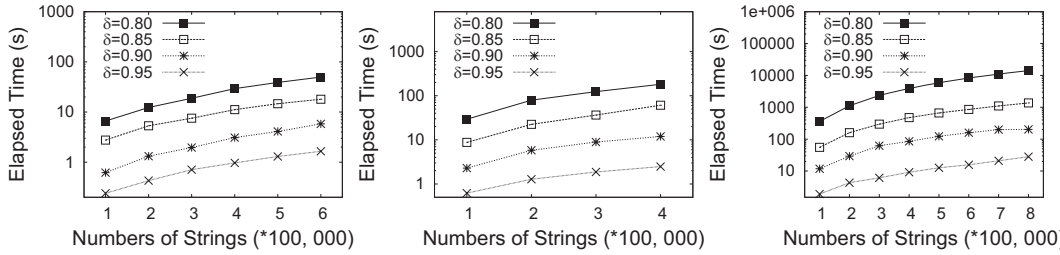
In this section, we evaluate the scalability of our method. We varied the number of strings in the dataset and tested the elapsed time.

7.4.1. Evaluating Edit Distance. Figure 23 shows the results using edit-distance function. We can see that our method achieved nearly linear scalability as the number of strings increases. For example, for $\tau = 4$, on the DBLP Author dataset, the elapsed time for 400,000 strings, 500,000 strings, and 600,000 strings were respectively 360 seconds, 530 seconds, and 700 seconds. This is attributed to our effective segment filter.

7.4.2. Evaluating Normalized Edit Distance. To support normalized edit distance, TRIE-JOIN and ED-JOIN needed to use the maximal length of strings to deduce the edit-distance thresholds. If the length difference between strings is large, these two methods are rather expensive and lead to low performance.[9] We also compared with these two state-of-the-art methods. However they are rather inefficient and cannot report the results in 24 hours. Thus we do not show the results in our experiments. Figure 24 shows the results of our SEGFILTER-NED algorithm. We can see that our method scales very well for the normalized edit distance and it achieves as high efficiency as on the edit-distance function.

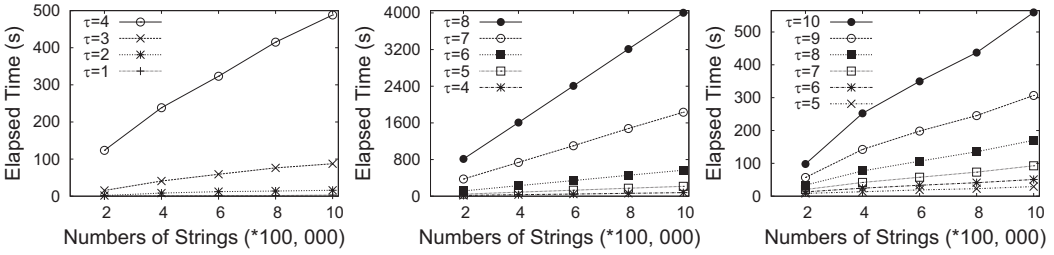
7.4.3. Evaluating R-S Join. We evaluate our similarity join algorithm to support R–S join. (See Figure 25). We compared with TRIE-JOIN. As ED-JOIN and QCHUNK-JOIN focused on self-join and the authors did not implement the R-S join algorithms, we did not show their results. We increased the number of strings in CITESEERX Author, Query Log

[9]Notice that we cannot extend ED-JOIN to support normalized edit distance efficiently. This is because they did not group the strings based on lengths. They used prefix filter and thus we cannot use our techniques to deduce tighter bounds.



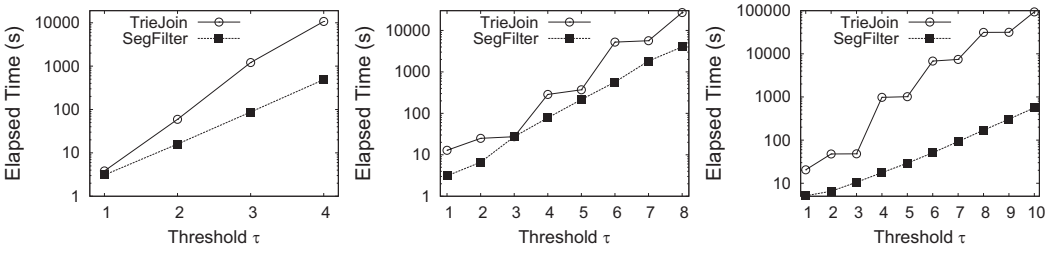
(a) DBLP Author(AvgLen=15) (b) Query Log 1(AvgLen=45) (c) DBLP Author+Title(AvgLen=105)

Fig. 24. Scalability (Normalized Edit Distance).



(a) CITESEERX Author(AvgLen=20) (b) Query Log 2 (AvgLen=40) (c) CITESEERX Author+Title(AvgLen=107)

Fig. 25. R-S Join.



(a) CITESEERX Author(AvgLen=20) (b) Query Log 2 (AvgLen=40) (c) CITESEERX Author+Title(AvgLen=107)

Fig. 26. Comparison of state-of-the-art R-S Join algorithms.

2, and CITESEERX Author+Title by 200,000 each time and respectively joined them with DBLP Author, Query Log 1, and DBLP Author+Title. We evaluated the elapsed time. Figure 26 shows the results. We can see that our method still scales well for R-S join and outperformed TRIE-JOIN. For example, on the CITESEERX Author+Title dataset. For $\tau = 8$, the elapsed time for 0.2 million strings was about 33 seconds, while for 1 million strings, the time was about 170 seconds. This is because our filtering algorithms and verification algorithms can improve the performance.

8. RELATED WORK

String Similarity Join. There have been many studies on string similarity joins [Gravano et al. 2001; Arasu et al. 2006; Bayardo et al. 2007; Chaudhuri et al. 2006; Sarawagi and Kirpal 2004; Xiao et al. 2008a; Xiao et al. 2009; Qin et al. 2011; Vernica et al. 2010]. The approaches most related to ours are TRIE-JOIN [Wang et al. 2010],

All-Pairs-Ed [Bayardo et al. 2007], ED-JOIN [Xiao et al. 2008a], QCHUNK-JOIN [Qin et al. 2011] and Part-Enum [Arasu et al. 2006]. All-Pairs-Ed is a q-gram-based method. It first generates q-grams for each string and then selects the first $q\tau + 1$ grams as a gram prefix based on a pre-defined order. It prunes the string pairs with no common grams and verifies the survived string pairs. ED-JOIN improves All-Pairs-Ed by using location-based and content-based mismatch filters. It has been shown that ED-JOIN outperforms All-Pairs-Ed [Bayardo et al. 2007]. QCHUNK-JOIN is a variant of All-Pairs-Ed that utilizes an asymmetric signature scheme to index the q-gram and search the q-chunks, and adopts an error estimation-based filtering. Although our techniques utilize length difference to do pruning, they are different from the error estimation-based filtering as follows. First, our position-aware substring selection technique is in the filtering step that can prune large numbers of dissimilar string pairs. However the error estimation-based filtering method is in the verification step, which only prunes the candidate pairs one by one. Second, our length-aware verification technique can improve the verification time for both similar string pairs and dissimilar strings pairs while the error estimation-based filtering can only prune dissimilar pairs. Third, our early termination technique can get much better estimation on edit distance than the error estimation-based method. For example, consider a matrix entry $M(i, j)$ for string r and string s. They use two estimated values $|i - j|$ and $|(|r| - i|) - (|s| - j)|$ to estimate the edit distance while we can get the accurate value of $M(i, j)$ and only use $|(|r| - i|) - (|s| - j)|$ to estimate the edit distance. Fourth, our extension-based verification technique only considers a matching segment while the error estimation-based method requires to consider all matching grams. Thus the error estimation-based method considers many more candidate pairs than our method. Also our extension-based verification technique uses much tighter bounds to accelerate the verification step. TRIE-JOIN uses a trie structure to do similarity joins based on prefix filtering. Part-Enum proposed an effective signature scheme called Part-Enum to do similar joins for hamming distance. It has been proved that All-Pairs-Ed and Part-Enum are worse than ED-JOIN, QCHUNK-JOIN and TRIE-JOIN [Wang et al. 2010; Feng et al. 2012]. Thus we only compared with state-of-the-art methods ED-JOIN and TRIE-JOIN.

Gravano et al. [2001] proposed gram-based methods and used SQL statements for similarity joins inside relational databases. Sarawagi and Kirpal [2004] proposed inverted index-based algorithms to solve similarity-join problem. Chaudhuri et al. [2006] proposed a primitive operator for effective similarity joins. Arasu et al. [2006] developed a signature scheme that can be used as a filter for effective similarity joins. Xiao et al. [2008b] proposed ppjoin to improve all-pair algorithm by introducing positional filtering and suffix filtering. Xiao et al. [2009] studied top-k similarity joins, which can directly find the top-k similar string pairs without a given threshold.

In addition, Jacox and Samet [2008] studied the metric-space similarity join. As this method is not as efficient as ED-JOIN and TRIE-JOIN [Wang et al. 2010], we did not compare with it in the article. Chaudhuri et al. [2006] proposed the prefix-filtering signature scheme for effective similarity join. Recently, Wang et al. [2011] devised a new similarity function by tolerating token errors in token-based similarity and developed effective algorithms to support similarity join on such functions. Jestes et al. [2010] studied the problem of efficient string joins in probabilistic string databases, by using lower bound filters based on probabilistic q-grams to effectively prune string pairs. Silva et al. [2010] focused on similarity joins as first-class database operators. They proposed several similarity join operators to support similarity joins in databases. Recently Vernica et al. [2010] studied how to support similarity joins in map-reduce environments.

Difference from Our Conference Version [Li et al. 2011b]. The significant additions in this extended manuscript are summarized as follows.

—We proposed new optimization techniques to improve our verification method. Section 5.3 was newly added. We also conducted a new experiment to evaluate our new optimization techniques and show their superiority on real datasets. Figures 19–22 were newly added based on our new method.
—We discussed how to support normalized edit distance and how to support R-S join. Section 6 was newly added. We also conducted experiments to evaluate our new techniques and Sections 7.4.2 and 7.4.3 were newly added.
—We formally proved all the theorem and lemmas and the appendix was newly added. We refined the article to make it easy to follow and added some new references.

Approximate String Search. The other related studies are approximate string searching [Chaudhuri et al. 2003; Li et al. 2008; Hadjieleftheriou et al. 2008a; Li et al. 2011c; Hadjieleftheriou et al. 2009; Zhang et al. 2010; Behm et al. 2011; Behm et al. 2009; Yang et al. 2008; Wang et al. 2012; Li et al. 2013; Deng et al. 2013], which given a query string and a set of strings, finds all similar strings of the query string in the string set. Hadjieleftheriou and Li [2009] gave a tutorial to the approximate string searching problem. Existing methods usually adopted a gram based indexing structure to do efficient filtering. They first generated grams of each string and built gram based inverted lists. Then they merged the inverted lists to find answers. Navarro [2001] studied the approximate string matching problem, which given a query string and a text string, finds all substrings of the text string that are similar to the query string. Notice that these two problems are different from our similarity-join problem, which given two sets of strings, finds all similar string pairs.

Approximate Entity Extraction. There are some studied on approximate entity extraction [Agrawal et al. 2008; Chakrabarti et al. 2008; Wang et al. 2009; Li et al. 2011a; Sun and Naughton 2011; Deng et al. 2012], which, given a dictionary of entities and a document, finds all substrings of the document that are similar to some entities. Existing methods adopted inverted indices and used different filters (e.g., length filter, count filter, position filter, and token order filter) to facilitate the extraction.

Estimation. There are some studies on selectivity estimation for approximate string queries and similarity joins [Hadjieleftheriou et al. 2008; Lee et al. 2007, 2009, 2011; Jin et al. 2008].

9. CONCLUSION

In this article, we have studied the problem of string similarity joins with edit-distance constraints. We proposed a new filter, the segment filter, to facilitate the similarity join. We devised a partition scheme to partition a string into several segments. We sorted and visited strings in order. We built inverted indices on top of the segments of the visited strings. For the current string, we selected some of its substrings and utilized the selected substrings to find similar string pairs using the inverted indices and then inserted segments of the current string into the inverted indices. We developed a position-aware method and a multimatch-aware method to select substrings. We proved that the multimatch-aware selection method can minimize the number of selected substrings. We also developed efficient techniques to verify candidate pairs. We proposed a length-aware method, an extension-based method, and an iterative-based method to further improve the verification performance. We extended our techniques to support normalized edit distance and R-S join. Experiments show that our method outperforms state-of-the-art studies on both short strings and long strings.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

REFERENCES

Agrawal, S., Chakrabarti, K., Chaudhuri, S., and Ganti, V. 2008. Scalable ad-hoc entity extraction from text collections. *Proc. VLDB Endow. 1*, 1, 945–957.

Arasu, A., Ganti, V., and Kaushik, R. 2006. Efficient exact set-similarity joins. In *Proceedings of the International Conference on Very Large Databases*. 918–929.

Bayardo, R. J., Ma, Y., and Srikant, R. 2007. Scaling up all pairs similarity search. In *Proceedings of the International World Wide Web Conference*. 131–140.

Behm, A., Ji, S., Li, C., and Lu, J. 2009. Space-constrained gram-based indexing for efficient approximate string search. In *Proceedings of the International Conference on Data Engineering*. 604–615.

Behm, A., Li, C., and Carey, M. J. 2011. Answering approximate string queries on large data sets using external memory. In *Proceedings of the International Conference on Data Engineering*. 888–899.

Chakrabarti, K., Chaudhuri, S., Ganti, V., and Xin, D. 2008. An efficient filter for approximate membership checking. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 805–818.

Chaudhuri, S., Ganjam, K., Ganti, V., and Motwani, R. 2003. Robust and efficient fuzzy match for online data cleaning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 313–324.

Chaudhuri, S., Ganti, V., and Kaushik, R. 2006. A primitive operator for similarity joins in data cleaning. In *Proceedings of the International Conference on Data Engineering*. 5–16.

Deng, D., Li, G., and Feng, J. 2012. An efficient trie-based method for approximate entity extraction with edit-distance constraints. In *Proceedings of the International Conference on Data Engineering*. 141–152.

Deng, D., Li, G., Feng, J., and Li, W.-S. 2013. Top-k string similarity search with edit-distance constraints. In *Proceedings of the International Conference on Data Engineering*.

Feng, J., Wang, J., and Li, G. 2012. Trie-join: a trie-based method for efficient string similarity joins. *VLDB J. 21*, 4, 437–461.

Gravano, L., Ipeirotis, P. G., Jagadish, H. V., Koudas, N., Muthukrishnan, S., and Srivastava, D. 2001. Approximate string joins in a database (almost) for free. In *Proceedings of the International Conference on Very Large Databases*. 491–500.

Hadjieleftheriou, M., Chandel, A., Koudas, N., and Srivastava, D. 2008a. Fast indexes and algorithms for set similarity selection queries. In *Proceedings of the International Conference on Data Engineering*. 267–276.

Hadjieleftheriou, M., Koudas, N., and Srivastava, D. 2009. Incremental maintenance of length normalized indexes for approximate string matching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 429–440.

Hadjieleftheriou, M. and Li, C. 2009. Efficient approximate search on string collections. *Proc. VLDB Endow. 2*, 2, 1660–1661.

Hadjieleftheriou, M., Yu, X., Koudas, N., and Srivastava, D. 2008b. Hashed samples: selectivity estimators for set similarity selection queries. *Proc. VLDB Endow. 1*, 1, 201–212.

Jacox, E. H. and Samet, H. 2008. Metric space similarity joins. *ACM Trans. Datab. Syst. 33*, 2.

Jestes, J., Li, F., Yan, Z., and Yi, K. 2010. Probabilistic string similarity joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 327–338.

Jin, L., Li, C., and Vernica, R. 2008. Sepia: estimating selectivities of approximate string predicates in large databases. *VLDB J. 17*, 5, 1213–1229.

Lee, H., Ng, R. T., and Shim, K. 2007. Extending q-grams to estimate selectivity of string matching with low edit distance. In *Proceedings of the International Conference on Very Large Databases*. 195–206.

Lee, H., Ng, R. T., and Shim, K. 2009. Power-law based estimation of set similarity join size. *Proc. VLDB Endow. 2*, 1, 658–669.

Lee, H., Ng, R. T., and Shim, K. 2011. Similarity join size estimation using locality sensitive hashing. *Proc. VLDB Endow. 4*, 6, 338–349.

Li, C., Lu, J., and Lu, Y. 2008. Efficient merging and filtering algorithms for approximate string searches. In *Proceedings of the International Conference on Data Engineering*. 257–266.

Li, G., Deng, D., and Feng, J. 2011a. Faerie: efficient filtering algorithms for approximate dictionary-based entity extraction. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 529–540.

LI, G., DENG, D., WANG, J., AND FENG, J. 2011b. Pass-join: A partition-based method for similarity joins. *Proc. VLDB Endow. 5*, 3, 253–264.

LI, G., FENG, J., AND LI, C. 2013. Supporting search-as-you-type using sql in databases. *IEEE Trans. Knowl. Data Eng. 25*, 2, 461–475.

LI, G., JI, S., LI, C., AND FENG, J. 2011c. Efficient fuzzy full-text type-ahead search. *VLDB J. 20*, 4, 617–640.

NAVARRO, G. 2001. A guided tour to approximate string matching. *ACM Comput. Surv. 33*, 1, 31–88.

QIN, J., WANG, W., LU, Y., XIAO, C., AND LIN, X. 2011. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1033–1044.

SARAWAGI, S. AND KIRPAL, A. 2004. Efficient set joins on similarity predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 743–754.

SILVA, Y. N., AREF, W. G., AND ALI, M. H. 2010. The similarity join database operator. In *Proceedings of the International Conference on Data Engineering*. 892–903.

SUN, C. AND NAUGHTON, J. F. 2011. The token distribution filter for approximate string membership. In *Proceedings of the International Workshop on Web and Databases*.

UKKONEN, E. 1985. Algorithms for approximate string matching. *Inf. Control 64*, 1-3, 100–118.

VERNICA, R., CAREY, M. J., AND LI, C. 2010. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 495–506.

WANG, J., LI, G., AND FENG, J. 2010. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *Proc. VLDB Endow. 3*, 1, 1219–1230.

WANG, J., LI, G., AND FENG, J. 2011. Fast-join: An efficient method for fuzzy token matching based string similarity join. In *Proceedings of the International Conference on Data Engineering*. 458–469.

WANG, J., LI, G., AND FENG, J. 2012. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 85–96.

WANG, W., XIAO, C., LIN, X., AND ZHANG, C. 2009. Efficient approximate entity extraction with edit distance constraints. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 759–770.

XIAO, C., WANG, W., AND LIN, X. 2008a. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *Proc. VLDB Endow. 1*, 1, 933–944.

XIAO, C., WANG, W., LIN, X., AND SHANG, H. 2009. Top-k set similarity joins. In *Proceedings of the International Conference on Data Engineering*. 916–927.

XIAO, C., WANG, W., LIN, X., AND YU, J. X. 2008b. Efficient similarity joins for near duplicate detection. In *Proceedings of the International World Wide Web Conference*. 131–140.

YANG, X., WANG, B., AND LI, C. 2008. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 353–364.

ZHANG, Z., HADJIELEFTHERIOU, M., OOI, B. C., AND SRIVASTAVA, D. 2010. Bed-tree: An all-purpose index structure for string similarity search based on edit distance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 915–926.

Received June 2012; revised November 2012; accepted February 2013