Fast-Join: An Efficient Method for Fuzzy Token Matching based String Similarity Join

Jiannan Wang Guoliang Li Jianhua Feng
Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
wjn08@mails.tsinghua.edu.cn; liguoliang@tsinghua.edu.cn; fengjh@tsinghua.edu.cn

***Abstract*—String similarity join that finds similar string pairs between two string sets is an essential operation in many applications, and has attracted significant attention recently in the database community. A significant challenge in similarity join is to implement an effective fuzzy match operation to find all *similar* string pairs which may not match exactly. In this paper, we propose a new similarity metrics, called "fuzzy token matching based similarity", which extends token-based similarity functions (e.g., Jaccard similarity and Cosine similarity) by allowing fuzzy match between two tokens. We study the problem of similarity join using this new similarity metrics and present a signature-based method to address this problem. We propose new signature schemes and develop effective pruning techniques to improve the performance. Experimental results show that our approach achieves high efficiency and result quality, and significantly outperforms state-of-the-art methods.**

I. Introduction

Similarity join has become a fundamental operation in many applications, such as data integration and cleaning, near duplicate object detection and elimination, and collaborative filtering [20]. In this paper we study string similarity join, which, given two sets of strings, finds all *similar* string pairs from each set. Existing studies [15], [6], [2], [3], [20], [21], [18] mainly use the following functions to quantify similarity of two strings.

Token-based similarity functions: They first tokenize strings as token sets ("bag of words"), and then quantify the similarity based on the token sets, such as Jaccard similarity and Cosine similarity. Usually if two strings are similar, their token sets should have a large overlap. Token-based similarity functions have a limitation that they only consider exact match of two tokens, and neglect fuzzy match of two tokens. Note that many data sets contain typos and inconsistences in their tokens and may have many mismatched token pairs that refer to the same token. For example, consider two strings "`nba mcgrady`" and "`macgrady nba`". Their token sets are respectively {"`nba`", "`mcgrady`"} and {"`macgrady`", "`nba`"}. The two token sets contain a mismatched token pair ("`mcgrady`", "`macgrady`"). As an example, the Jaccard similarity between the two strings is $1/3$ (the ratio of the number of tokens in their intersection to that in their union). Although the two strings are very similar, their Jaccard similarity is very low.

Character-based similarity functions : They use characters in the two strings to quantify the similarity, such as edit distance which is the minimum number of single-character edit operations (i.e., insertion, deletion, and substitution) needed to transform one to another. In comparison with token-based similarity, edit distance is sensitive to the positions of the tokens in a string. For example, recall the two strings "`nba mcgrady`" and "`macgrady nba`". Their edit distance is 9. Although the two strings are very similar, their edit-distance-based similarity is very low.

The above two classes of similarity metrics have limitations in evaluating the similarity of two strings. These problems seem trivial but are very serious for many datasets, such as Web query log and person names. To address this problem, we propose a new similarity metrics, fuzzy token matching based similarity (hereinafter referred to as fuzzy-token similarity), by combining token-based similarity and character-based similarity. Different from token-based similarity that only considers exact match between two tokens, we also incorporate character-based similarity of mismatched token pairs into the fuzzy-token similarity. For example, recall the two strings "`nba mcgrady`" and "`macgrady nba`". They contain one exactly matched token "`nba`" and one approximately matched token pair ("`mcgrady`", "`macgrady`"). We consider both of the two cases in the fuzzy-token similarity. We give the formal definition of the fuzzy-token similarity and prove that many well-known similarity functions (e.g., Jaccard similarity) are special cases of fuzzy-token similarity (Section II).

There are several challenges to address the similarity-join problem using fuzzy-token similarity. Firstly, fuzzy-token similarity is more complicated than token-based similarity and character-based similarity, and it is even rather expensive to compute fuzzy-token similarity of two strings (Section II-B). Secondly, for exact match of token pairs, we can sort the tokens and use prefix filtering to prune large numbers of dissimilar string pairs [20]. However as we consider fuzzy match of two tokens, it is nontrivial to sort the tokens and use prefix filtering. Thus it calls for new effective techniques and efficient algorithms. In this paper we propose *f*uzzy token m*a*tching based *s*tring similari*t*y *join* (called Fast-Join) to address these problems. To summarize, we make the following contributions in this paper.

- We propose a new similarity metric, fuzzy-token similarity, and prove that many existing token-based similarity

functions and character-based similarity functions are special cases of fuzzy-token similarity.
- We formulate the similarity-join problem using fuzzy-token similarity. We propose a signature-based framework to address this problem.
- We propose a new signature scheme for token sets and prove it is superior to the state-of-the-art method. We present a new signature scheme for tokens and develop effective punning techniques to improve the performance.
- We have implemented our method in real data sets. The experimental results show that our method achieves high performance and result quality, and outperforms state-of-the-art methods.

The rest of this paper is organized as follows. Section II proposes the fuzzy-token similarity. Section III formalizes the similarity-join problem using fuzzy-token similarity and presents a signature-based method. To increase performance, we propose new signature schemes for token sets and tokens respectively in Section IV and Section V. Experimental results are provided in Section VI. We review related works in Section VII and make a conclusion in Section VIII.

II. Fuzzy-Token Similarity

We first review existing similarity metrics, and then formalize the fuzzy-token similarity. Finally we prove that existing similarities are special cases of fuzzy-token similarity.

A. Existing Similarity Metrics

String similarity functions are used to quantify the similarity between two strings, which can be roughly divided into three groups: token-based similarity, character-based similarity, and hybrid similarity.

Token-based similarity: It tokenizes strings into token sets (e.g., using white space) and quantifies the similarity based on the token sets. For example, given a string "`nba mcgrady`", its token set is {"`nba`", "`mcgrady`"}. We give some representative token-based similarity: Dice similarity, Cosine similarity, and Jaccard similarity, defined as follows. Given two strings s_1 and s_2 with token sets T_1 and T_2:

Dice similarity: $\text{DICE}(s_1, s_2) = \frac{2\cdot|T_1 \cap T_2|}{|T_1|+|T_2|}$.

Cosine similarity: $\text{COSINE}(s_1, s_2) = \frac{|T_1 \cap T_2|}{\sqrt{|T_1|\cdot|T_2|}}$.

Jaccard similarity: $\text{JACCARD}(s_1, s_2) = \frac{|T_1 \cap T_2|}{|T_1|+|T_2|-|T_1 \cap T_2|}$.

Note that the token-based similarity functions use the overlap of two token sets to quantify the similarity. They only consider exactly matched token pairs to compute the overlap, and neglect the approximately matched pairs which refer to the same token. For example, consider two strings s_1 = "`nba trace mcgrady`" and s_2 = "`trac macgrady nba`". Their token sets have a overlap "`nba`", and their Jaccard similarity is $1/5$. Consider another string s_3 = "`nba trace video`". For s_1 and s_3, their token sets have a larger overlap {"`nba`", "`trace`"}, and their Jaccard similarity is $2/4$. Although $\text{JACCARD}(s_1, s_2) < \text{JACCARD}(s_1, s_3)$, actually s_2 should be much more similar to s_1 than s_3, since all of the three tokens in s_2 are similar to those in s_1.

Character-based similarity: It considers characters in strings to quantify the similarity. As an example, edit distance is the minimum number of single-character edit operations (i.e. insertion, deletion, substitution) to transform one into another. For example, the edit distance between "`macgrady`" and "`mcgrady`" is 1. We normalize the edit distance to interval [0,1] and use *edit similarity* to quantify the similarity of two strings, where edit similarity between two strings s_1 and s_2 is $\text{NED}(s_1, s_2) = 1 - \frac{\text{ED}(s_1,s_2)}{\max(|s_1|,|s_2|)}$ in which $|s_1|$ ($|s_2|$) denotes the length of s_1 (s_2).

Note that edit similarity is sensitive to the position information of each token. For example, the edit similarity between strings "`nba trace mcgrady`" and "`trace mcgrady nba`" is very small although they are actually very similar.

Hybrid Similarity: Chaudhuri et al. [5] proposed generalized edit similarity (GES), which extends the character-level edit operator to the token-level edit operator. For example, consider two strings "`nba mvp mcgrady`" and "`mvp macgrady`". We can use two token-level edit operators to transform the first one to the second one (e.g. deleting the token "`nba`" and substituting "`mcgrady`" for "`macgrady`"). Note that we can consider the token weight in the transformation. For example, "`nba`" is less important than "`macgrady`" and we can assign a lower weight for "`nba`". However the generalized edit similarity is sensitive to token positions.

Chaudhuri et al. [5] also derived an approximation of generalized edit similarity (AGES). This similarity ignores the positions of tokens and requires each token in one string to match the "closest" token (the most similar one) in another string. For example, consider two strings s_1 = "`wnba nba`" and s_2 = "`nba`". For the tokens in s_1 "`wnba`" and "`nba`", their "closest" tokens in s_2 are both "`nba`". We respectively compute the similarity between "`wnba`" and "`nba`" and that between "`nba`" and "`nba`". The AGES between s_1 and s_2 is the average value of these two similarity values, i.e. AGES(s_1,s_2) = $\frac{0.75+1}{2} = 0.875$. However AGES does not follow the symmetry property. Consider the above two strings s_1 and s_2. We can also compute their similarity from the viewpoint of s_2. For the token in s_2 "`nba`", we find its "closest" token in s_1 "`nba`". We only need to compute the similarity between "`nba`" and "`nba`". The AGES between s_1 and s_2 turns to this similarity value, i.e. AGES(s_2,s_1) = 1. The asymmetry property will make AGES have limitations for many practical problems. For example, if using AGES to quantity the similarity for self-join problem, AGES will lead to inconsistent results.

As existing similarity functions have limitations, to address these problems, we propose a new similarity metrics.

B. Fuzzy-Token Similarity

We propose a powerful similarity metrics, fuzzy-token similarity, by combining token-based similarity and character-based similarity. Different from token-based similarity which computes the exact overlap of two token sets (i.e., the number of exactly matched token pairs), we compute *fuzzy overlap* in considering fuzzy match between tokens as follows.

Given two token sets, we use character-based similarity to quantify the similarity of token pairs from the two sets. As an example, in this paper we focus on edit similarity. We first compute the edit similarity of each token pair from the two sets, then use *maximum weight matching in bipartite graphs (bigraphs)* for computing *fuzzy overlap* as follows.

We construct a weighted bigraph $G = \big((X, Y), E\big)$ for token sets T_1 and T_2 as follows, where X and Y are two disjoin sets of vertexes, and E is a set of weighted edges that connect a vertex from X to a vertex in Y. In our problem, as illustrated in Figure 1, the vertexes in X and Y are respectively tokens in T_1 and T_2, and an edge from a token $t_i \in T_1$ to a token $t'_j \in T_2$ is their edit similarity. For example, in the figure, the edge with the weight $w_{1,1}$ means that the edit similarity between t_1 and t'_1 is $w_{1,1}$. We can only keep the edges with weight larger than a given edit-similarity threshold δ. The *maximum weight matching* of G is a set of edges $M \subseteq E$ satisfying the following conditions: (1) Matching: Any two edges in M have no a common vertex; (2) Maximum: The sum of weights of edges in M is maximal. We use G's *maximum weight matching* as the *fuzzy* overlap of T_1 and T_2, denoted by $T_1 \widetilde{\cap}_\delta T_2$. Note that the time complexity for finding maximum weight matching is $\mathcal{O}(|V|^2 * |E|)$ [4], where $|V|$ is the number of vertexes and $|E|$ is the number of edges in bigraph G. We give an example to show how to compute the *fuzzy overlap*.

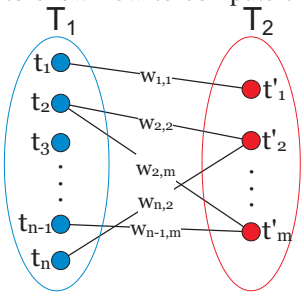


Fig. 1. Weighted bigraph

Example 1: Consider two strings s_1 = "nba mcgrady" and s_2 = "macgrady nba". We first compute the edit similarity of each token pair: NED("nba", "macgrady") $= \frac{1}{8}$, NED("nba", "nba") $= 1$, NED("mcgrady", "macgrady") $= \frac{7}{8}$, NED("mcgrady", "nba")= $\frac{1}{7}$. For an edit-similarity threshold $\delta = 0.8$, we construct a weighted bigraph with two weighted edges: one edge e_1 with weight 1 for token pair ("nba", "nba") and the other edge e_2 with weight $\frac{7}{8}$ for token pair ("mcgrady", "macgrady"). The *maximum weight matching* of this bigraph is the edge set $\{e_1, e_2\}$ which meets two conditions: matching and maximum. Thus the *fuzzy overlap* $T_1 \widetilde{\cap}_{0.8} T_2$ is $\{e_1, e_2\}$ and its weight is $|T_1 \widetilde{\cap}_{0.8} T_2| = \frac{15}{8}$.

Using *fuzzy overlap*, we define fuzzy-token similarity.

Definition 1 (Fuzzy-Token Similarity): Given two strings s_1 and s_2 and an edit-similarity threshold δ, let T_1 and T_2 be the token sets of s_1 and s_2 respectively,

Fuzzy-Dice similarity: $\text{FDICE}_\delta(s_1, s_2) = \frac{2 \cdot |T_1 \widetilde{\cap}_\delta T_2|}{|T_1| + |T_2|}$.

Fuzzy-Cosine similarity: $\text{FCOSINE}_\delta(s_1, s_2) = \frac{|T_1 \widetilde{\cap}_\delta T_2|}{\sqrt{|T_1| \cdot |T_2|}}$.

Fuzzy-Jaccard similarity: $\text{FJACCARD}_\delta(s_1, s_2) = \frac{|T_1 \widetilde{\cap}_\delta T_2|}{|T_1| + |T_2| - |T_1 \widetilde{\cap}_\delta T_2|}$.

For example, consider s_1 and s_2 in Example 1. Their Fuzzy-Jaccard similarity is $\text{FJACCARD}_\delta(s_1, s_2) = \frac{1+7/8}{4-1-7/8} = 15/17$.

C. Comparison with Existing Similarities

In this section, we compare fuzzy-token similarity with existing similarities. Existing token-based similarity such as Jaccard similarity obeys the triangle inequality, however fuzzy-token similarity does not obey the triangle inequality. We give an example to prove this property. Consider three strings with only one token, s_1="abc", s_2="abcd" and s_3="bcd". We have $\text{NED}(s_1, s_2) = \text{NED}(s_2, s_3) = 0.75$, and $\text{NED}(s_1, s_3) = \frac{1}{3}$. Let edit-similarity threshold $\delta = 0.5$. We have $|s_1 \widetilde{\cap}_{0.5} s_2| = |s_2 \widetilde{\cap}_{0.5} s_3| = 0.75$ and $|s_1 \widetilde{\cap}_{0.5} s_3| = 0$ (as $\frac{1}{3} < 0.5$). Thus $\text{FJACCARD}_\delta(s_1, s_2) = \text{FJACCARD}_\delta(s_2, s_3) = \frac{0.75}{2-0.75} = 0.6$ and $\text{FJACCARD}_\delta(s_1, s_3) = 0$. Usually, one minus the similarity denotes the corresponding distance. We have $(1 - 0.6) + (1 - 0.6) < (1 - 0)$. Thus Fuzzy-Jaccard similarity does not obey the triangle inequality. Similarly, the example can also show Fuzzy-Dice similarity and Fuzzy-Cosin similarity do not obey the triangle inequality. Thus our similarities are not metric-space similarities and cannot use existing studies [10] to support our similarities.

Compared with AGSE (Section II-A), fuzzy-token similarity has the symmetry property. This is because we construct a bigraph for the token sets of two strings, and the *maximum weight matching* of this bigraph is symmetric, thus $|T_1 \widetilde{\cap}_\delta T_2| = |T_2 \widetilde{\cap}_\delta T_1|$.

Next we investigate the relationship between fuzzy-token similarity and existing similarities. We first compare it with token-based similarity. If $\delta = 1$ for fuzzy-token similarity, then a fuzzy overlap will be equal to the overlap (Lemma 1), and the corresponding fuzzy-token similarity will turn to token-based similarity. Thus token-based similarity is only a special case of the fuzzy-token similarity when $\delta = 1$.

Lemma 1: For token sets T_1 and T_2, $|T_1 \widetilde{\cap}_1 T_2| = |T_1 \cap T_2|$.

For the general case ($\delta \in [0, 1]$), a fuzzy overlap never have a smaller value than the corresponding overlap (Lemma 2).

Lemma 2: For token sets T_1 and T_2, $|T_1 \widetilde{\cap}_\delta T_2| \geq |T_1 \cap T_2|$.

Based on this Lemma, we can deduce that fuzzy-token similarity will never have a smaller value than the corresponding token-based similarity. One advantage of this property is that for a string pair, if they are similar evaluated by token-based similarity, then they are still similar for fuzzy-token similarity.

Next we compare fuzzy-token similarity with edit similarity. We find that edit similarity is also a special case of the fuzzy-token similarity as stated in Lemma 3.

Lemma 3: Given two strings s_1 and s_2, let token sets $T_1 = \{s_1\}$ and $T_2 = \{s_2\}$, we have $|T_1 \widetilde{\cap}_{\delta=0} T_2| = \text{NED}(s_1, s_2)$.

Based on the above analysis, fuzzy-token similarity is a generalization of token-based similarity and character-based similarity, and is more powerful than them as experiments proved in Section VI. Fuzzy-token similarity also has some different properties from existing similarities which pose new challenges when using it to quantify the similarity.

III. String Similarity Join Using Fuzzy-Token Similarity

In this section, we study the similarity-join problem using fuzzy-token similarity to compute similar string pairs.

A. Problem Formulation

Let S and S' be two collections of strings, and R and R' be the corresponding collections of token sets. For $T \in R$ and $T' \in R'$, let $\mathcal{F}_\delta(T, T')$ denote the fuzzy-token similarity of T and T', where $\mathcal{F}_\delta$ could be FJACCARD$_\delta$, FCOSINE$_\delta$, and FDICE$_\delta$. We define the similarity-join problem as follows.

Definition 2: (Fuzzy token matching based string similarity join): Given two collections of strings S and S', and a threshold τ, a fuzzy token matching based string similarity join is to find all the pairs $(s, s') \in S \times S'$ such that $\mathcal{F}_\delta(T, T') \geq \tau$, where $T(T')$ is the token set of $s(s')$.

A straightforward method to address this problem is to enumerate each pair $(T_1, T_2) \in R \times R'$ and compute their fuzzy-token similarity. However this method is rather expensive, and we propose an efficient method, called Fast-Join.

B. A Signature-Based Method

We adopt a signature-based method [15]. First we generate signatures for each token set, which have a property that: given two token sets T_1 and T_2 with signature sets $Sig(T_1)$ and $Sig(T_2)$ respectively, T_1 and T_2 are similar only if $Sig(T_1) \cap Sig(T_2) \neq \phi$. Based on this property we can filter large numbers of dissimilar pairs and obtain a small set of candidate pairs. Finally, we verify the candidate pairs to generate the finial results. We call our method as Fast-Join.

Signature Schemes: It is very important to devise a high-quality scheme in this framework, as such signature can prune large numbers of dissimilar pairs. Section IV and Section V study how to generate high-quality signatures.

The filter step: This step generates candidates of similar pairs based on signatures. We use an *inverted index* to generate candidates [15] as follows. Each signature in the signature sets has an inverted list of those token sets whose signature sets contain the signature. In this way, two token sets in the same inverted lists are candidates as their signature sets have overlaps. For example, given token sets T_1, T_2, T_3, T_4, with $Sig(T_1) = \{ad, ac, dc\}$, $Sig(T_2) = \{be, cf, em\}$, $Sig(T_3) = \{ad, ab, dc\}$, and $Sig(T_4) = \{bm, cf, be\}$. The inverted list of ad is $\{T_1, T_3\}$. Thus (T_1, T_3) is a candidate. As there is no signature whose inverted list contains both T_1 and T_2, they are dissimilar and can be pruned. To find similar pairs among the four token sets, we generate two candidates (T_1, T_3) and (T_2, T_4) and prune the other four pairs.

We can optimize this framework using the all-pair based algorithm [3]. In this paper, we focus on how to generate effective signatures and use this framework as an example. Our method can be easily extended to other frameworks.

The refine step: This step verifies the candidates to generate the final results. Given two token sets T_1 and T_2, we construct a weighted bigraph as described in Section II-B. As it is expensive to compute the maximum weighted matching, we propose an improved method. We compute an upper bound of the maximal weight by relaxing the "matching" condition, that is we allow that the edges in M can share a common vertex. We can compute this upper bound by summing up the maximum weight of edges of every token in T_1 (or T_2). If this upper bound makes $\mathcal{F}_\delta(T_1, T_2)$ smaller than τ, we can prune the pair (T_1, T_2), since $\mathcal{F}_\delta(T_1, T_2)$ is no larger than its upper bound and thus will also be smaller than τ.

IV. Signature Scheme of Token Sets

In the signature-based method, it is very important to define a high-quality signature scheme, since a better signature scheme can prune many more dissimilar pairs and generate smaller numbers of candidates. In this section we propose a high-quality signature scheme for token sets.

A. Existing Signature Schemes

Let us first review existing signature schemes for exact search, i.e., $\delta = 1$. Consider two token sets $T_1 = \{t_1, t_2, \ldots, t_n\}$ and $T_2 = \{t'_1, t'_2, \ldots, t'_m\}$ where t_i denotes a token in T_1 and t'_j denotes a token in T_2. Suppose T_1 and T_2 are similar if $|T_1 \cap T_2| \geq c$, where c is a constant. A simple signature scheme is $Sig(T_1) = T_1$ and $Sig(T_2) = T_2$. Obviously if T_1 and T_2 are similar, their overlap is not empty, that is $Sig(T_2)$ and $Sig(T_2)$ have common signatures. As this method involves large numbers of signatures, it will lead to low efficiency. A well-known improved method is to use prefix filtering [6], which selects a subset of tokens as signatures. To use prefix filtering, we first fix a global order on all signatures (i.e. tokens). We then remove the $\lceil c-1 \rceil$ signatures with largest order from $Sig(T_1)$ and $Sig(T_2)$ to obtain the new signature set $Sig_p(T_1)$ and $Sig_p(T_2)$. Note that if T_1 and T_2 are similar, $|Sig_p(T_1) \cap Sig_p(T_2)| \neq \phi$ [6].

For example, consider two token sets $T_1 = \{$"nba", "kobe", "bryant"$\}$, $T_2 = \{$"nba", "tracy", "mcgrady"$\}$ and a threshold $c = 2$. They cannot be filtered by the simple signature scheme, as $Sig(T_1) = T_1$ and $Sig(T_2) = T_2$ have overlaps. Using alphabetical order, we can remove "nba" from $Sig(T_1)$ and "tracy" from $Sig(T_2)$, and get $Sig_p(T_1) = \{$"bryant", "kobe"$\}$ and $Sig_p(T_2) = \{$"nba", "mcgrady"$\}$. As they have no overlaps, we can prune them.

However, it is not straightforward to extend this method to support $\delta \neq 1$ as we consider fuzzy token matching. For example, consider the token sets $\{$"hoston", "mcgrady"$\}$ and $\{$"houston", "macgrady"$\}$. Clearly they have large *fuzzy overlap* but have no overlap. To address this problem, we propose an effective signature scheme for *fuzzy overlap*.

B. Token-Sensitive Signature

As the similarity function $\mathcal{F}_\delta$ is rather complicated and it is hard to devise an effective signature scheme for this similarity, we simplify it and deduce an Equation that if $\mathcal{F}_\delta(T_1, T_2) \geq \tau$, then there exists a constant c such that $|T_1 \ \widetilde{\cap}_\delta \ T_2| \geq c$. Then we propose a signature scheme $Sig^\delta(\cdot)$ satisfying: if $|T_1 \ \widetilde{\cap}_\delta \ T_2| \geq c$, then $Sig^\delta(T_1) \cap Sig^\delta(T_2) \neq \phi$. We can

devise a pruning technique: if $Sig^\delta(T_1) \cap Sig^\delta(T_2) = \phi$, we have $|T_1 \widetilde{\cap}_\delta T_2| < c$ and $\mathcal{F}_\delta(T_1, T_2) < \tau$, thus we can prune (T_1, T_2). Section IV-C gives how to deduce c for different similarity functions. Here we discuss how to devise effective signature schemes for $|T_1 \widetilde{\cap}_\delta T_2| \geq c$.

Signature scheme for $|T_1 \widetilde{\cap}_\delta T_2| \geq c$: Given two token sets $T_1 = \{t_1, t_2, \ldots, t_n\}$ and $T_2 = \{t'_1, t'_2, \ldots, t'_m\}$, we study how to generate the signature sets $Sig^\delta(T_1)$ and $Sig^\delta(T_2)$ for the condition $\delta \neq 1$ such that if $|T_1 \widetilde{\cap}_\delta T_2| \geq c$, then $Sig^\delta(T_1) \cap Sig^\delta(T_2) \neq \phi$. Recall that $T_1 \widetilde{\cap}_\delta T_2$ denotes the maximum weight matching of their corresponding weighted bigraph G. Each edge in G for vertexes $t_i \in T_1$ and $t'_j \in T_2$ is $\text{NED}(t_i, t'_j) \geq \delta$. We construct another bigraph G' with the same vertexes and edges as G except that the edge weights are assigned as follows. We first generate the signatures of tokens t_i and t'_j, denoted as $sig^\delta(t_i)$ and $sig^\delta(t'_j)$ respectively, such that if $\text{NED}(t_i, t'_j) \geq \delta$, $sig^\delta(t_i) \cap sig^\delta(t'_j) \neq \phi$. (We will discuss how to generate the signature scheme for tokens in Section V.) Then for each edge of vertexes t_i and t'_j in G', we assign its weight to $|sig^\delta(t_i) \cap sig^\delta(t'_j)|$. As there exists an edge in G between t_i and t'_j, we have $sig^\delta(t_i) \cap sig^\delta(t'_j) \neq \phi$, thus $|sig^\delta(t_i) \cap sig^\delta(t'_j)| \geq 1 \geq \text{NED}(t_i, t'_j)$. Obviously the maximal matching weight in G is no larger than that in G'. Without loss of generality, let $M = \{(t_1, t'_1), (t_2, t'_2), \ldots, (t_k, t'_k)\}$ be the maximal weight matching of G' where each element (t_i, t'_i) in M denotes an edge of G' with the edge weight of $|sig^\delta(t_i) \cap sig^\delta(t'_i)|$. Thus the maximal matching weight of G' is $\sum_{i=1}^{k} |sig^\delta(t_i) \cap sig^\delta(t'_i)|$. Based on the definition of matching, no two edges in M have a common vertex. Hence,

$$\sum_{i=1}^{k} |sig^\delta(t_i) \cap sig^\delta(t'_i)| \leq |\Big(\biguplus_{i=1}^{k} sig^\delta(t_i)\Big) \cap \Big(\biguplus_{i=1}^{k} sig^\delta(t'_i)\Big)|$$
$$\leq |\Big(\biguplus_{i=1}^{n} sig^\delta(t_i)\Big) \cap \Big(\biguplus_{i=1}^{m} sig^\delta(t'_i)\Big)|$$

where $\biguplus$ denotes the union operation for multisets[1].

Base on above analysis, we have $|T_1 \widetilde{\cap}_\delta T_2| \leq |Sig^\delta(T_1) \cap Sig^\delta(T_2)|$ as formalized in Lemma 4. Thus, we use $Sig^\delta(T_1) = \biguplus_{i=1}^{n} sig^\delta(t_i)$ and $Sig^\delta(T_2) = \biguplus_{i=1}^{m} sig^\delta(t'_i)$ as the signatures of T_1 and T_2 respectively, such that if $Sig^\delta(T_1) \cap Sig^\delta(T_2) = \phi$, $|T_1 \widetilde{\cap}_\delta T_2| \leq 0 < c$.

Lemma 4: For each token set $T_1 = \{t_1, t_2, \ldots, t_n\}$ and $T_2 = \{t'_1, t'_2, \ldots, t'_m\}$, $|T_1 \widetilde{\cap}_\delta T_2| \leq |Sig^\delta(T_1) \cap Sig^\delta(T_2)|$ where $Sig^\delta(T_1) = \biguplus_{i=1}^{n} sig^\delta(t_i)$ and $Sig^\delta(T_2) = \biguplus_{i=1}^{m} sig^\delta(t'_j)$.

Obviously we can use prefix filtering to improve this signature scheme. We fix a global order and then generate $Sig_p^\delta(T_1)$ from $Sig^\delta(T_1)$ by removing the last $\lceil c - 1 \rceil$ signatures with largest order. Example 2 gives an example.

Example 2: Consider the collection of token sets $\mathcal{R}$ in Figure 2. Given $\delta = 0.8$ and $c = 2.4$, we aim to generate a signature set for each token set in $\mathcal{R}$ such that if two token sets are similar (i.e. $|T_i \widetilde{\cap}_{0.8} T_j| \geq 2.4$), then their corresponding signature sets have overlaps (i.e. $Sig^\delta(T_i) \cap Sig^\delta(T_j) \neq \phi$).

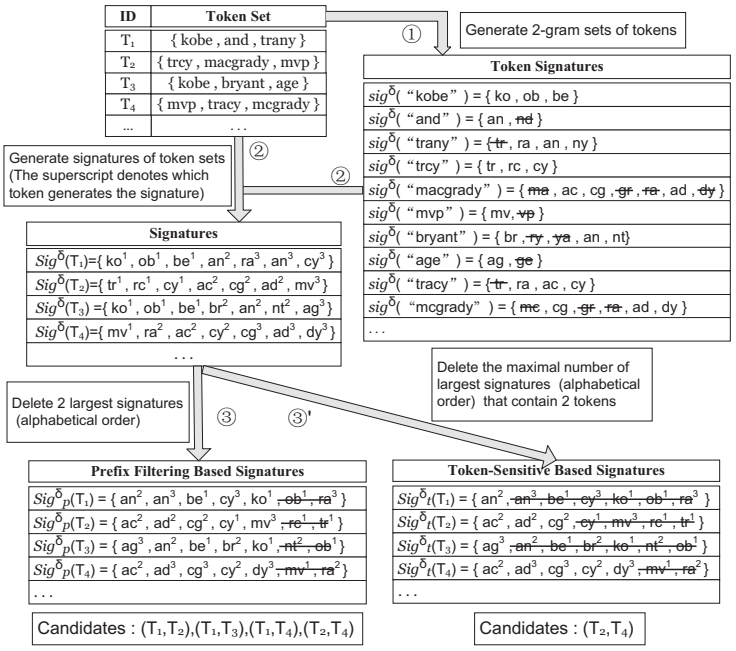


Fig. 2. Prefix filtering and token-sensitive signatures of one sample collection of token sets $\mathcal{R}$ ($\delta = 0.8$, $c = 2.4$)

At the first step, as shown in "Token Signatures", we collect all the tokens in $\mathcal{R}$ and generate a signature set for each token. Here we choose some q-grams (substrings of the token that consists of q consecutive characters) as token's signatures [20], which will be explained in Section V. For instance, the signature set of `macgrady` is {"`ac`", "`cg`", "`ad`"}. We find that if two tokens are similar (e.g. NED("`macgrady`", "`mcgrady`") $\geq$ 0.8), they at least share one signature (e.g. "`ad`").

At the second step, we generate signatures $Sig^\delta(T_i)$ as the union of its tokens' signatures. For example, consider the token set T_2 = {"`trcy`", "`macgrady`","`mvp`"}, we have $Sig^\delta(T_2)$= {"`tr`^{1}", "`rc`^{1}", "`cy`^{1}", "`ac`^{2}", "`cg`^{2}", "`ad`^{2}", "`mv`^{3}"}. Each signature has a superscript that denotes which token generates this signature. For instance, "`ac`^{2}" denotes that the signature "`ac`" is generated from the second token "`macgrady`". Note that $Sig^\delta(T_i)$ is a multiset. For example, $Sig^\delta(T_1)$ contains two "`an`" from the second and the third tokens respectively.

At the third step, to generate signatures using prefix filtering, we delete $\lceil c-1 \rceil = 2$ largest signatures (if we use alphabetical order) from $Sig^\delta(T_j)$ and generate $Sig_p^\delta(T_j)$. For instance, we can get $Sig_p^\delta(T_2)$ by removing "`rc`" and "`tr`" from $Sig^\delta(T_2)$ since they are the two largest signatures based on alphabetical order. Using this signature scheme, $Sig_p^\delta(T_3)$ have no overlap with both $Sig_p^\delta(T_2)$ and $Sig_p^\delta(T_4)$, so we can filter (T_2, T_3) and (T_3, T_4). For other token-set pairs such as (T_2, T_4), because $Sig_p^\delta(T_2)$ and $Sig_p^\delta(T_4)$ have common signatures, they will be considered as the candidate pair for further verification.

Token-Sensitive Signature: We propose a novel signature scheme that can remove many more signatures than prefix filtering. As an example, consider the token sets T_1 and T_3

[1]In this paper, we use multiset which is a generalization of a set. A multiset can have more than one membership, that is there may be multiple instances of a member in a multiset.

Algorithm 1: TokenSensitiveSignature(T, c)

Input: T : a token set
c : a fuzzy-overlap threshold
Output: $Sig_t^\delta(T)$: the token-sensitive signature set of T

```
1  begin
2      Sig_t^δ(T) = ⊎_{t∈T} sig^δ(t);
3      Let H be a hash table storing token ids;
4      for each s^tid ∈ Sig_t^δ(T) in decreasing global order on
       signatures do
5          if tid ∉ H then
6              Add tid into H;
7              if H.size() ≥ c then
8                  break;
9          Remove s^tid from Sig_t^δ(T);
10     return Sig_t^δ(T);
11 end
```

Fig. 3. Algorithm of generating token-sensitive signatures for a token set

in Figure 2. $Sig^\delta(T_1)$ and $Sig^\delta(T_3)$ have a large overlap {"an", "be", "ko", "ob"}. Thus based on prefix filtering, when $c = 2.4$ they will not be filtered. Here we have an observation that these signatures are only generated from two tokens. For example, the overlap {"an^2", "be^1", "ko^1", "ob^1"} in T_3 is generated from two tokens "kobe" and "bryant". That is T_3 at most has two similar tokens with T_1. However, if $|T_1 \widetilde{\cap}_{0.8} T_3| \geq 2.4$, T_3 has at least $\lceil c \rceil = 3$ tokens similar to T_1. Therefore, T_1 and T_3 should be filtered. Based on this observation, we devise a new filter condition in Lemma 5.

Lemma 5: Given two token sets T_1 and T_2, and a threshold c, if signatures in $Sig^\delta(T_1) \cap Sig^\delta(T_2)$ are only generated from smaller than $\lceil c \rceil$ tokens in T_1 (or T_2), then the token pair (T_1, T_2) can be pruned.

We can use this filter condition to reduce the size of signature set and call it token-sensitive signature scheme. Given a token set T, we generate its token-sensitive signature set $Sig_t^\delta(T)$ as follows. Different from prefix filtering signature scheme which removes the last $\lceil c - 1 \rceil$ signatures, token-sensitive signature scheme removes the maximal number of largest signatures (in the global order on signatures) that are generated from at most $\lceil c - 1 \rceil$ distinct tokens. That is if we remove one more signatures, then the removed signatures are generated from $\lceil c \rceil$ tokens.

Lemma 6 shows token-sensitive signature scheme generates no larger number of signatures than the prefix-filtering signature scheme. This is because if the last $\lceil c \rceil$ signatures come from $\lceil c \rceil$ different tokens, then both of signature schemes will remove $\lceil c-1 \rceil$ signatures; otherwise, token-sensitive signature scheme will remove more than $\lceil c - 1 \rceil$ signatures but prefix filtering signature scheme only remove $\lceil c - 1 \rceil$ signatures.

Lemma 6: Given the same global order and the same signature scheme for tokens, the token-sensitive signature scheme generates no larger number of signatures than the prefix filtering signature scheme, i.e., $Sig_t^\delta(T) \subseteq Sig_p^\delta(T)$.

We give the pseudo-code of token-sensitive signature scheme in Algorithm 3. Firstly, $Sig_t^\delta(T)$ is initialized as the union of the signature sets of T's tokens. Then we scan the signatures in $Sig_t^\delta(T)$ based on the pre-defined global order decreasingly. For each signature s^{tid}, we check whether the token tid has occurred before. We use a hash table $\mathcal{H}$ to store the occurred tokens. If tid has occurred (i.e. $tid \in \mathcal{H}$), we remove s^{tid} from $Sig_t^\delta(T)$. If tid has not occurred (i.e. $tid \notin \mathcal{H}$), we add tid into $\mathcal{H}$ and if $\mathcal{H}$.size() $\geq c$, we stop scanning the following signatures and return the signature set $Sig_t^\delta(T)$; otherwise, we remove s^{tid} from $Sig_t^\delta(T)$ and scan the next signature. Example 3 shows how this algorithm works.

Example 3: Consider the token set T_1 in Figure 2. Given $\delta = 0.8$ and $c = 2.4$, we first initialize $Sig^\delta(T_1)$ = {"an^2", "an^3", "be^1", "cy^3", "ko^1", "ob^1", "ra^3"} with signatures sorted in alphabetical order. We scan the signatures in $Sig^\delta(T_1)$ from back to front. Initially, $\mathcal{H} = \{\}$. For the first signature "ra^3, it comes from the third token "trany" in T_1, since $3 \notin \mathcal{H}$, we add 3 into $\mathcal{H}$. As the size of $\mathcal{H} = \{3\}$ is smaller than 2.4, we remove "ra^3" from $Sig^\delta(T_1)$ and scan the next signature "ob^1". Since "ob^1" comes from the first token and $1 \notin \mathcal{H}$, we add 1 into $\mathcal{H}$. As the size of $\mathcal{H} = \{1, 3\}$ is smaller than 2.4, we remove "ob^1" from $Sig^\delta(T_1)$. Note that the prefix filtering signature scheme will stop here, but the token-sensitive signature scheme will scan the next signature "ko^1". Since "ko^1" comes from the first token and $1 \in \mathcal{H}$, we can directly remove "ko^1" from $Sig^\delta(T_1)$ and scan the following signatures. We can also remove "cy^3", "be^1", "an^3" as they come from the first or the third tokens which have already been added into $\mathcal{H}$. Finally, we stop at the signature "an^2". Since "an^2" comes from the second token and $2 \notin \mathcal{H}$, we add 2 into $\mathcal{H}$. As the size of $\mathcal{H} = \{1, 2, 3\}$ is no smaller than 2.4, we stop removing signatures and return the final signature set $Sig_t^\delta(T_1) = \{$"an^2"$\}$.

Figure 2 shows the token-sensitive signatures of the token sets in $\mathcal{R}$. Compared with prefix-filtering signature scheme, it significantly reduces the size of a signature set and filters more token-set pairs. In Example 2, prefix-filtering signature scheme can only prune (T_2, T_3) and (T_3, T_4), but since $Sig_t^\delta(T_1) \cap Sig_t^\delta(T_2) = \phi$ and $Sig_t^\delta(T_1) \cap Sig_t^\delta(T_3) = \phi$ and $Sig_t^\delta(T_1) \cap Sig_t^\delta(T_4) = \phi$, token-sensitive signature scheme can further filter the token-set pairs (T_1, T_2) and (T_1, T_3) and (T_1, T_4).

C. Deducing Constant c

In this section, we deduce how to compute the constant c, such that if $\mathcal{F}_\delta(T_1, T_2) \geq \tau$, then there exists a constant c such that $|T_1 \widetilde{\cap}_\delta T_2| \geq c$.

Fuzzy-Dice Similarity:

$$\frac{2 \cdot |T_1 \widetilde{\cap}_\delta T_2|}{|T_1| + |T_2|} \geq \tau \implies \frac{2 \cdot |T_1 \widetilde{\cap}_\delta T_2|}{|T_1| + |T_1 \widetilde{\cap}_\delta T_2|} \geq \tau$$
$$\implies |T_1 \widetilde{\cap}_\delta T_2| \geq \frac{\tau}{2 - \tau} \cdot |T_1| \quad (1)$$

Fuzzy-Cosine Similarity:

$$\frac{|T_1 \widetilde{\cap}_\delta T_2|}{\sqrt{|T_1| \cdot |T_2|}} \geq \tau \implies \frac{|T_1 \widetilde{\cap}_\delta T_2|}{\sqrt{|T_1| \cdot |T_1 \widetilde{\cap}_\delta T_2|}} \geq \tau$$
$$\implies |T_1 \widetilde{\cap}_\delta T_2| \geq \tau^2 |T_1| \quad (2)$$

Fuzzy-Jaccard Similarity:

$$\frac{|T_1 \widetilde{\cap}_\delta T_2|}{|T_1|+|T_2|-|T_1 \widetilde{\cap}_\delta T_2|} \geq \tau \implies \frac{|T_1 \widetilde{\cap}_\delta T_2|}{|T_1|-|T_1 \widetilde{\cap}_\delta T_2|+|T_1 \widetilde{\cap}_\delta T_2|} \geq \tau \implies |T_1 \widetilde{\cap}_\delta T_2| \geq \tau \cdot |T_1| \quad (3)$$

Thus given a token set T_1, we can deduce that $c = \frac{\tau}{2-\tau} \cdot |T_1|$ for Fuzzy-Dice similarity, $c = \tau^2|T_1|$ for Fuzzy-Cosin similarity, and $c = \tau \cdot |T_1|$ for Fuzzy-Jaccard similarity.

We can prove that if $\mathcal{F}_\delta(T_1, T_2) \geq \tau$, then $Sig_t^\delta(T_1) \cap Sig_t^\delta(T_2) \neq \phi$. We only show the proof of Fuzzy-Jaccard similarity. Fuzzy-Dice similarity and Fuzzy-Cosin similarity can be proved similarly. If $\text{FJACCARD}_\delta(T_1, T_2) \geq \tau$, then $|T_1 \widetilde{\cap}_\delta T_2| \geq \max(c_1, c_2)$ where $c_1 = \tau \cdot |T_1|$ and $c_2 = \tau \cdot |T_2|$. Let $Sig_t^\delta(T_1)$ and $Sig_t^\delta(T_1)'$ be the signature set of T_1 when the fuzzy-overlap threshold is c_1 and $\max(c_1, c_2)$ respectively. Let $Sig_t^\delta(T_2)$ and $Sig_t^\delta(T_2)'$ be the signature set of T_2 when the fuzzy-overlap threshold is c_2 and $\max(c_1, c_2)$ respectively. As $|T_1 \widetilde{\cap}_\delta T_2| \geq \max(c_1, c_2)$, $Sig_t^\delta(T_1)' \cap Sig_t^\delta(T_2)' \neq \phi$. As $\max(c_1, c_2)$ is no smaller than c_1 and c_2, $Sig_t^\delta(T_1)' \subseteq Sig_t^\delta(T_1)$ and $Sig_t^\delta(T_2)' \subseteq Sig_t^\delta(T_2)$, thus $Sig_t^\delta(T_1) \cap Sig_t^\delta(T_2) \neq \phi$.

V. Signature Schemes for Tokens

As we need to use the signatures of tokens for generating the signatures of token sets, in this section, we study effective signature schemes for tokens.

A. Extending Existing Signature Schemes to Support Edit Similarity

Many signature schemes [7], [20], [16], [2], [19] are proposed to evaluate edit distance. They generate signature sets for tokens t and t', such that if $\text{ED}(t, t')$ is no larger than an edit-distance threshold λ, then their signature sets have overlaps. But for edit similarity, tokens with different lengths might have different edit-distance thresholds. In order to use existing signature schemes, given an edit-similarity threshold δ, for a token t we can compute its maximal edit-distance threshold λ such that for any token t' if $\text{NED}(t, t') \geq \delta$, then $\text{ED}(t, t') \leq \lambda$. As $\text{NED}(t, t') = 1 - \frac{\text{ED}(t,t')}{\max(|t|,|t'|)} \geq \delta$, we have $1 - \frac{\text{ED}(t,t')}{|t|+\text{ED}(t,t')} \geq \delta$, that is $\text{ED}(t, t') \leq \frac{1-\delta}{\delta} \cdot |t|$. Thus we can set $\lambda = \frac{1-\delta}{\delta} \cdot |t|$. For example, consider the token "`tracy`" and $\delta = 0.8$. For any token t' such that $\text{NED}(\text{"}\texttt{tracy}\text{"}, \texttt{t}') \geq 0.8$, the edit distance between t' and "`tracy`" is no larger than $\frac{1-0.8}{0.8} \cdot |5| = 1.25$. Next we review existing signature schemes for tokens. Note that they are designed for edit distance instead of edit similarity, we extend them to support edit similarity.

q-gram-based signature scheme [7], [20] utilizes the idea that if two tokens are similar, they will have enough common q-grams where a q-gram is a substring with length q. To extend q-gram-based signature scheme to support edit similarity, for a token t we compute its maximal edit-distance threshold $\lambda = \frac{1-\delta}{\delta} \cdot |t|$ based on the given edit-similarity threshold δ. We generate t's signature set using the edit-distance threshold λ. However the q-gram-based signature scheme is ineffective for short tokens as it will result in a large number of candidates which need to be further verified.

Deletion-based neighborhood generation [16]**:** We can use the same idea as the q-gram-based signature scheme to extend the deletion-based neighborhood generation to support edit similarity. However this scheme will generate a large number of signatures for long tokens, even for a large edit-similarity threshold.

Part-Enum [2] uses the pigeon-hole principle to generate signatures. For a token t, it first obtains the q-gram set represented as a feature vector. For two tokens, if their edit distance is within λ, then the hamming distance between their feature vectors is no larger than $q \cdot \lambda$. Based on this property, to generate the signatures of the token t with the edit-distance threshold λ, Part-Enum only needs to generate the signatures of the feature vector of t with the hamming-distance threshold $q \cdot \lambda$. It divides the feature vector into $\lceil \frac{q\cdot\lambda+1}{2} \rceil$ partitions, and based on the pigeon-hole principle there exists at least one partition whose hamming distance is no larger than 1. For each partition, it further divides the partition into multiple sub-partitions. All of the sub-partitions compose the signatures of t. To extend Part-Enum to support edit similarity, we cannot simply generate signatures with the maximal edit-distance threshold. This is because edit distance will affect the number of partitions. For example, given the edit-similarity threshold $\delta = 0.8$ and $q = 1$, for "`macgrady`" the maximal edit-distance threshold $\lambda = \frac{1-0.8}{0.8} \cdot |8| = 2$, and Part-Enum needs to divide its feature vector into $\lceil \frac{1\cdot 2+1}{2} \rceil = 2$ partitions. But for "`mcgrady`", the maximal edit-distance threshold $\lambda = \frac{1-0.8}{0.8} \cdot |7| = 1$, and Part-Enum needs to divide its feature vector into $\lceil \frac{1\cdot 1+1}{2} \rceil = 1$ partition. Although $\text{NED}(\text{"}\texttt{mcgrady}\text{"}, \text{"}\texttt{macgrady}\text{"}) \geq 0.8$, their signature sets have no overlap. To solve this problem, for a token t we compute the minimum length $\delta \cdot |t|$ of a token t' such that $\text{NED}(t, t') \geq \delta$. When generating the signatures for t, we consider the maximal edit-distance threshold $\lfloor \frac{1-\delta}{\delta} \cdot l \rfloor$ for each possible length l of the token t', i.e. $l \in [\delta \cdot |t|, |t|]$. For example, consider the token "`macgrady`". The length range is $[0.8 \cdot 8, 8]$. Two lengths 7 and 8 satisfy this range. For them, we respectively compute the maximal edit-distance thresholds for $l = 7$, $\lfloor \frac{1-0.8}{0.8} \cdot |7| \rfloor = 1$ and for $l = 8$, $\lfloor \frac{1-0.8}{0.8} \cdot |8| \rfloor = 2$. The signature set of "`macgrady`" for $\delta = 0.8$ is the union of its signature set with the edit-distance thresholds 1 and 2. However Part-Enum needs to tune many parameters to generate signatures, and it generates larger numbers of candidates as it ignores the position information.

Partition-ED [19] is a partition-based signature scheme to solve approximate-entity-extraction problem. It also uses the pigeon-hole principle to generate signatures. Different from Part-Enum, it directly partitions a token instead of the feature vector of a token. Each token t will generate two signature sets, one is called query signature set $sig_q^\delta(t)$ and the other is called data signature set $sig_d^\delta(t)$. For two tokens t and t', if $\text{ED}(t, t') \leq \lambda$, then $sig_q^\delta(t) \cap sig_d^\delta(t') \neq \phi$. Given an edit-

distance threshold λ, to obtain $sig_q^\delta(t)$ it divides t into $\lceil\frac{\lambda+1}{2}\rceil$ partitions, and based on the pigeon-hole principle there exists at least one partition whose edit distance is no larger than 1. It adds 0- and 1-deletion neighborhoods of each partition into $sig_q^\delta(t)$ [16]. To obtain $sig_d^\delta(t)$, it still divides t into $\lceil\frac{\lambda+1}{2}\rceil$ partitions. But for each partition, it also needs to shift and scale it to generate more partitions [19]. For all generated partitions, it adds their 0- and 1-deletion neighborhoods into $sig_d^\delta(t)$

To extend Partition-ED to support edit similarity, for the query signature set $sig_q^\delta(t)$, we only need to generate $sig_q^\delta(t)$ with the edit-distance threshold $\frac{1-\delta}{\delta}\cdot|t|$. For the data signature set, as the same reason as Part-Enum, since the edit distance can affect the number of partitions, we compute the minimum length $\delta\cdot|t|$ and the maximum length $\frac{|t|}{\delta}$ of a token t' such that $\text{NED}(t,t')\geq\delta$. We generate $sig_q^\delta(t)$ with the edit-distance threshold $\lfloor\frac{1-\delta}{\delta}\cdot l\rfloor$ for each possible length l of t', i.e. $l\in[\delta\cdot|t|,|t|]$. For example, consider the token "`macgrady`" and $\delta=0.8$. The length range is $[0.8\cdot 8,\frac{8}{0.8}]$. Four lengths 7,8,9,10 satisfy this range. We generate $sig_d^\delta(t)$ with the edit-distance thresholds $\lfloor\frac{1-0.8}{0.8}\cdot|7|\rfloor=1$, $\lfloor\frac{1-0.8}{0.8}\cdot|8|\rfloor=2$, $\lfloor\frac{1-0.8}{0.8}\cdot|9|\rfloor=2$ and $\lfloor\frac{1-0.8}{0.8}\cdot|10|\rfloor=2$. However, Partition-ED will generate many redundant signatures. For example, for the strings with lengths 9 as their edit-distance threshold with "`macgrady`" should be no larger than $(1-\delta)*\max(9,|\text{"}\texttt{macgrady}\text{"}|)=1.8$, thus we do not need to generate signatures with the edit-distance threshold 2. Similarly, for strings with lengths 7 and 8, we only need to generate signatures with the edit-distance threshold 1. To address this problem, we propose a new signature scheme Partition-NED in Section V-B. Figure 4 compares the number of signatures generated by Partition-ED and Partition-NED for different lengths of tokens ($\delta=0.75$). We can see when the length of token is larger than 8, Partition-ED will generate many more signatures than Partition-NED. For example, Partition-ED generates 125 signatures for the tokens whose length is 10, and Partition-NED only generates 56 signatures. Experimental result in Section VI shows our algorithm achieves the best performance when using the Partition-NED signature scheme for generating signatures of tokens.

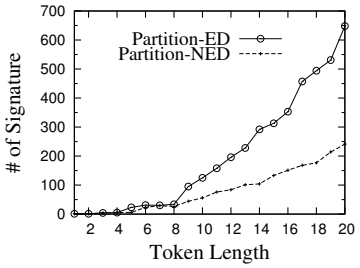


Fig. 4. Comparison of the number of signatures between Partition-ED and Partition-NED for different length of tokens ($\delta=0.75$)

B. Partition-NED Signature Scheme

As discussed in Section V-A, when extending existing signature schemes to support edit similarity, they have some limitations. To address these limitations, in this section we propose a new signature scheme for edit similarity called Partition-NED.

Overview of Partition-NED: For each token, we generate the same query signature set $sig_q^\delta(t)$ as Partition-ED. To generate the data signature set $sig_d^\delta(t)$, we compute the length range $[\delta\cdot|t|,\frac{|t|}{\delta}]$ of a token t' such that $\text{NED}(t,t')\geq\delta$. For each token t' with the length $|t'|\in[\delta\cdot|t|,\frac{|t|}{\delta}]$, t' is divided into $d=\lceil\frac{\lambda+1}{2}\rceil$ partitions where $\lambda=\lfloor\frac{1-\delta}{\delta}\cdot|t'|\rfloor$ is the maximal edit-distance threshold for the token t'. Based on the pigeon-hole principle, if $\text{NED}(t,t')\geq\delta$, there exists at least one partition whose edit distance with a substring of t is within 1. If we can find the corresponding substrings in t for each partition, we only need to add 0- and 1-deletion neighborhoods of them into $sig_d^\delta(t)$, then $sig_d^\delta(t)\cap sig_q^\delta(t')\neq\phi$. For example, consider the token t ($|t|=9$) in Figure 5. Given $\delta=0.75$, we can compute the length range $[0.75\cdot 9,\frac{9}{0.75}]$ of a token t' such that $\text{NED}(t,t')\geq 0.75$. There are six lengths 7,8,9,10,11,12 satisfying the range. For each token t' with the length $|t'|\in\{7,8,9,10,11,12\}$, e.g. the token t' ($|t'|=12$) in Figure 5, we compute its maximal edit-distance threshold $\lambda=\lfloor\frac{1-0.75}{0.75}\cdot 12\rfloor=4$ and get $d=\lceil\frac{4+1}{2}\rceil=3$ partitions. Since $\lambda=4$ and $d=3$, based on the pigeon-hole principle, there at least exists one partition whose edit distance with a substring of t is within 1. Therefore, the problem is how to find such substrings of t. In the following, we give the algorithm to solve this problem and propose two effective punning techniques to reduce the number of substrings.

Algorithm description: Consider two tokens $t=c_1c_2\cdots c_m$ and $t'=c'_1c'_2\cdots c'_n$. Suppose t' is divided into d partitions: $t'[1:\ell]=c_1\ldots c_\ell;t'[\ell+1:2\ell+1]=c_{\ell+1}\ldots c_{2*\ell};\cdots;t'[(d-1)*\ell+1:n]=c_{(d-1)*\ell+1}\ldots c_n$, where $\ell=\lfloor\frac{n}{d}\rfloor$. For example, in Figure 5 the token t' is divided into $d=3$ partitions $t'[1:4]$, $t'[5:8]$ and $t'[9:12]$, where $\ell=\lfloor\frac{12}{3}\rfloor=4$. Let $t[p_i:q_i]=c_{p_i}c_{p_i+1}\cdots c_{q_i}$ denote the i-th partition of t. Let $\lambda=(1-\delta)\cdot max(|t|,|t'|)$ be the edit-distance threshold between t and t'. For example, in Figure 5 if $\text{NED}(t,t')\geq 0.75$, then $\text{ED}(t,t')\leq(1-0.75)\cdot max(9,12)=3$, thus the edit-distance threshold is $\lambda=3$. For the partitions of t', we consider three cases to find corresponding substrings in t.

Case 1 - the first partition: Suppose the first partition $t'[p_1=1:q_1]$ has one or zero edit errors. For this partition, we select the substrings from t whose start position is 1 and lengths are within $[\vartheta-1,\vartheta+1]$ where ϑ denotes the length of $t[p_1:q_1]$. Thus we select the corresponding substrings $t[1:3]$, $t[1:4]$ and $t[1:5]$ from the token t as shown in Figure 5.

Case 2 - the last partition: Suppose the last partition $t'[p_d:n]$ has one or zero edit errors. For this partition, we select the substrings from t whose end position is n and lengths are within $[\vartheta-1,\vartheta+1]$ where ϑ denotes the length of $t'[p_d:n]$. Thus we select the corresponding substrings $t[5:9]$, $t[6:9]$ and $t[7:9]$ from the token t as shown in Figure 5.

Case 3 - the middle partitions: Suppose a middle partition $t'[p_i:q_i]$ ($i\neq 1,d$) has one or zero edit errors. To find its corresponding substrings in t, we know their lengths are within $[\vartheta-1,\vartheta+1]$ where ϑ denotes the length of $t'[p_i:q_i]$, then we have to determine its start positions of the corresponding

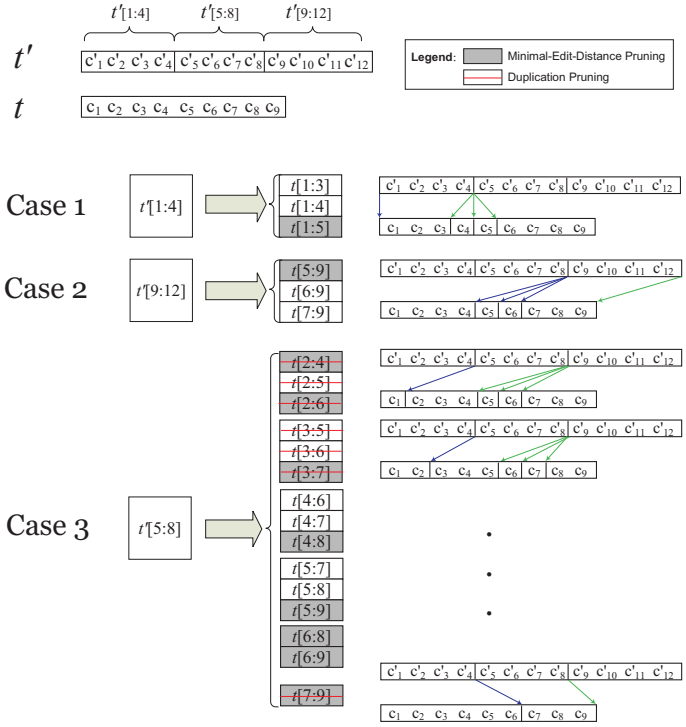


Fig. 5. For the partitions of t', we find eight corresponding substrings t[1:3], t[1:4], t[6:9],t[7:9], t[4:6], t[4:7], t[5:7] and t[5:8] of t ($\delta = 0.75$)

substrings. Wang et al. [19] presented that there are at most λ insertions or deletions before $t'[p_i : q_i]$, thus the start positions of the corresponding substrings must be within $[p_i - \lambda, p_i + \lambda]$. For each start position in this range, we need to consider the substrings whose lengths are within $[\vartheta - 1, \vartheta + 1]$ where ϑ denotes the length of $t'[p_i : q_i]$. Consider the middle partition $t'[5 : 8]$ in Figure 5. Since $\lambda = 3$, the start positions are within $[2, 8]$. For each start position in $[2, 8]$, we select three substrings whose lengths are within $[3, 5]$. For example, we select $t[2 : 4]$, $t[2 : 5]$ and $t[2 : 6]$ for the start position 2. We only select $t[7 : 9]$ for the start postilion 7 since $t[7 : 10]$ and $t[7 : 11]$ exceeds the length of t.

In Figure 5, for all the partitions of t', we totally find 21 corresponding substrings of t. Next, we propose two pruning techniques to reduce unnecessary substrings.

Minimal-Edit-Distance Pruning: Suppose $t[p_i : q_i]$ is the corresponding substring of the partition $t'[p'_i : q'_i]$. When computing the edit distance between t and t', $t[p_i : q_i]$ and $t'[p'_i : q'_i]$ should be aligned, and their prefix strings $t[1 : p_i - 1]$ and $t'[1 : p'_i - 1]$ should be aligned, and their suffix strings $t[q_i + 1 : m]$ and $t'[q'_i + 1 : n]$ should be aligned. So the edit distance $\text{ED}(t, t')$ is the sum of $\text{ED}(t[p_i : q_i], t'[p'_i : q'_i])$, $\text{ED}(t[1 : p_i - 1], t'[1 : p'_i - 1])$ and $\text{ED}(t[q_i + 1 : m], t'[q'_i + 1 : n])$. We know that the edit distance between two strings is no smaller than their length difference. Thus we can compute the minimum of the edit distance,

$$\text{ED}(t, t') \geq |\xi| + |p_i - p'_i| + |(m - q_i) - (n - q'_i)| \quad (4)$$

where $|\xi| = |(q_i - p_i) - (q'_i - p'_i)|$ is the length difference between $t[p_i : q_i]$ and $t'[p'_i : q'_i]$.

If the right side of Equation 4 is larger than λ, then we can prune the substrings $t[p_i : q_i]$. For example, in Figure 5 we can prune the corresponding substring $t[3 : 7]$ for the partition $t'[5 : 8]$ since the minimum of $\text{ED}(t, t')$ is $|1| + |3 - 5| + |(9 - 7) - (12 - 8)| = 5$ and 5 is larger than $\lambda = 3$.

Duplication Pruning: Recall three cases of selecting the corresponding substrings, we consider each partition independently, and thus some conditions may be repeatedly considered. For example, consider the substring $t[3 : 5]$ for the partition $t'[5 : 8]$ in Figure 5. On the left $t[1 : 2]$ of $t[3 : 5]$, it needs at least two edit operations to align $t[1 : 2]$ and $t'[1 : 4]$. Therefore, there exists at most one edit error on the right $t[6 : 9]$ of $t[3 : 5]$ due to the total edit distance $\lambda = 3$. Note that the condition that $t[6 : 9]$ has one or zero edit error has been considered in Case 2, and thus we can prune the substring $t[3 : 5]$.

Formally, to find the substrings of t, we first consider the first partition and the last partition. Then we consider the middle partitions from right to left. For the partition $t'[p'_i : q'_i]$ and let k denote the number of partitions behind $t'[p'_i : q'_i]$. We can prune the substrings in t' with start positions larger than $p'_i + \lambda - 2k$ (or smaller than $p'_i - (\lambda - 2k)$). This is because for each of such substrings, e.g. $t[p_i : q_i]$, the edit operations before $t[p_i : q_i]$ will be larger than $\lambda - 2k$ and correspondingly the edit operations after $t[p_i : q_i]$ will be smaller than $2k$ (otherwise the total edit distance is larger than λ). As there are k partitions behind $t[p_i : q_i]$, there at least exists one partition with zero or one edit error. As this partition has been considered, we can prune the substring $t[p_i : q_i]$.

In Figure 5, using minimal-edit-distance pruning we can prune 10 substrings and using duplication pruning we can prune 8 substrings. Using both of them, we can reduce the number of substrings from 21 to 8. We guarantee the correctness of Partition-NED as formalized in Lemma 7.

Lemma 7: Given two tokens t and t', and signature sets $sig_d^\delta(t)$ and $sig_q^\delta(t')$ generated using Partition-NED, we have if $\text{NED}(t, t') \geq \delta$, then $sig_d^\delta(t) \cap sig_q^\delta(t') \neq \phi$.

VI. Experimental Study

We used two real data sets and evaluated the effectiveness and the efficiency of our proposed methods.

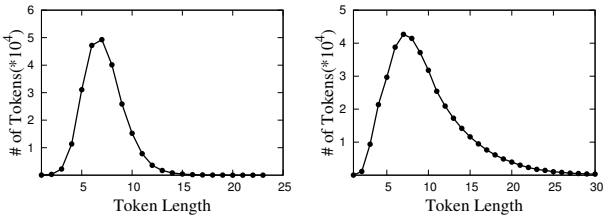
Data sets: 1) AOL Query Log[2]: We generate two sets of strings and each data sets included one million distinct real keyword queries. 2) DBLP Author: We extracted author names from DBLP dataset[3]. We also generate two sets of strings and each data sets included 0.6 million real person names. Table I illustrates detailed statistical information of the data sets, which gives the number of strings, the average number of tokens in a string, the maximal number of tokens in a string, and the minimal number of tokens in a string. Figures 6(a)-6(b) show the length distribution of tokens.

[2]http://www.gregsadetsky.com/aol-data/
[3]http://www.informatik.uni-trier.de/~ley/db

TABLE I
DATASET STATISTICS

Data Sets	Sizes	*avg_token_no*	*max_token_no*	*min_token_no*
Query Log	1,000,000	3.35	132	1
Author	613,542	2.77	8	1

We implemented all the algorithms in C++ and compiled using GCC 4.2.3 with -O3 flag. We used inverse document frequency (IDF) to sort the signatures. All the experiments were run on a Ubuntu Linux machine with an Intel Core 2 Quad E5420 2.50GHz processor and 4 GB memory.



(a) Author (b) Query Log

Fig. 6. Token length distribution

A. Result Quality

In this section, we compared the result quality for different similarity functions. We chose 100,000 queries from the Query Log dataset and computed the similar string pairs using jaccard similarity, fuzzy-jaccard similarity, edit similarity, GES and AGES. We first compared the number of similar string pairs generated using jaccard similarity and fuzzy-jaccard similarity as shown in Table II.

TABLE II
RESULT QUALITY FOR JACCARD AND FUZZYJACCARD SIMILARITY ($\delta = 0.8$) (THE PRECISION IS COMPUTED BY EVALUATING 100 RESULTS.)

τ	Jaccard		Fuzzy Jaccard	
	# of Results	Precision(%)	# of Results	Precision(%)
0.95	127	100	212	99
0.9	132	99	560	100
0.85	166	99	986	98
0.8	405	94	1520	93
0.75	1100	90	2344	86
0.7	1201	69	2698	84

We see that our similarity generates more similar string pairs. For example, when $\tau = 0.8$ and $\delta = 0.8$, fuzzy-jaccard returned 1520 similar pairs and jaccard found 405 results.

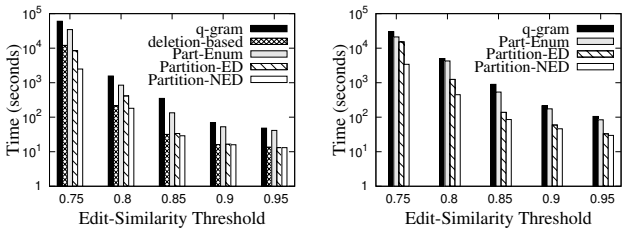
In addition, to evaluate result quality, we randomly selected 100 results from the generated pairs and asked five research members from our group to evaluate the results blindly. We can see that the method using fuzzy-jaccard similarity also achieved high result quality. For example, when $\tau = 0.7$, fuzzy-jaccard similarity achieved 84% precision. This is because we consider *fuzzy overlap*, which can find similar pairs with typos and inconsistences.

We also compared fuzzy-jaccard similarity with edit similarity and got similar results. For example, when $\delta = 0.75$, the precision of edit similarity is only 27%, while that of fuzzy-jaccard similarity is 90% ($\tau = 0.8$).

We compared fuzzy-jaccard similarity with existing hybrid similarity functions GES and AGES. We found GES missed a lot of similar query pairs. For example, when $\delta = 0.8$ and $\tau = 0.8$, GES only returned 486 pairs (precision 97%), while fuzzy jaccard returned 1520 results (precision 93%). This is because GES give a low similarity value to the similar query pairs where the same keywords occurred in different positions. Although AGES ignores the positions of tokens and returned more results, its precision is rather low. For example, when $\delta = 0.8$ and $\tau = 0.8$, AGES returned 25017 results and the precision was only 6% (about $25017 \times 6\% = 1501$ relevant pairs). Fuzzy-jaccard similarity returned 1520 results, and the precision was 93% (about $1520 \times 93\% = 1414$ relevant pairs). Thus Fuzzy-jaccard similarity has nearly the same recall with AGES, but achieves much higher precision than AGES.

B. Evaluation on Different Signature Schemes for Tokens

In this section, we compared the performance of different token signature schemes. We implemented five methods: q-gram based method [20], deletion-based neighborhood generation [16], Part-Enum [2], Partition-ED [19] and Partition-NED. We extended them to support edit similarity using the methods in Section V-A. We used the token-sensitive signature scheme for generating token sets. Figure 7 gives the results.



(a) Author (b) Query Log

Fig. 7. Performance for different token signature schemes ($\tau = 0.8$)

We see that the q-gram based method achieved the worst performance as it can only use small q for short tokens, but small q resulted in large numbers of false-positive results. Part-Enum also performed worse since converting a token to the feature vector destroyed the position information of grams. The deletion-based neighborhood generation scheme achieved higher performance for the Author data set as the tokens are usually short in person names. But for the Query Log dataset, the method generated large numbers of signatures for long tokens and achieved very low performance, and it did not report any result within 10^6 seconds. Thus in the figure we did not show the results of the deletion-based neighborhood generation. Partition-NED performed the best of all the signature schemes. When the edit-similarity threshold is large, Partition-ED has the comparable performance with Partition-NED. However when the edit-similarity threshold becomes smaller, Partition-ED will be less efficient than Partition-NED. This is because Partition-ED generated large numbers of signatures, but Partition-NED used the pruning techniques to remove unnecessary signatures.

In addition, we compared the numbers of token signatures generated from Partition-ED and Partition-NED. Figure 8 shows the results. We can see our method can reduce large numbers of signatures. For instance, on the Query Log dataset, for $\delta = 0.8$, Partition-NED generated $2.8{*}10^7$ signatures while Partition-ED only generated $1.8 * 10^7$ signatures.

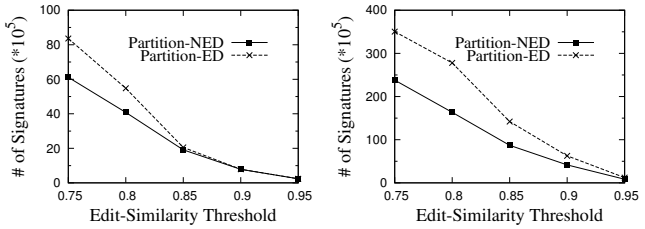


Fig. 8. Comparison of the numbers of token signatures between Partition-ED and Partition-NED ($\tau = 0.8$)

As Partition-NED achieved the highest performance, we used Partition-NED for generating token signatures in the remainder experiments of this paper.

C. Evaluation on Signature Schemes of Token Sets

In this section, we compared the performance of token-sensitive signature scheme and prefix-filtering signature scheme. We first compared the number of removed signatures. Figure 9 shows the results.

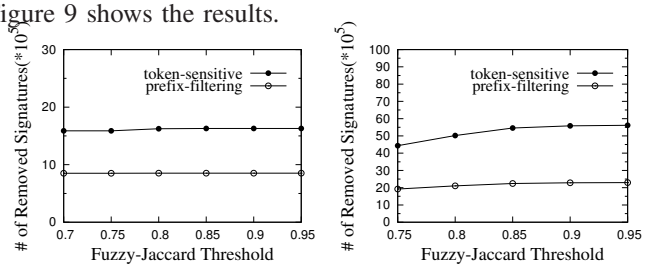


Fig. 9. Comparison of the numbers of removed token-set signatures between prefix filtering and token-sensitive prefix filtering ($\delta = 0.85$)

We can see that token-sensitive signature scheme can remove many more signatures as it considered token information in the removal step. For example, on the Author dataset, for $\tau = 0.8$, the token-sensitive signature scheme can remove $1.5 * 10^6$ signatures and the prefix-filtering signature scheme only removed $0.9 * 10^6$ signatures.

We also compared the number of candidates gotten from the two token-set signature schemes. Figure 10 shows the results. We see that token-sensitive signature scheme generated fewer candidates than prefix-filtering signature scheme. This is because it removed many more unnecessary signatures. For example, on Query Log, for $\delta = 0.85$, token-sensitive signature scheme generated less than $1.2{*}10^6$ candidates, while prefix-filtering signature scheme generated $1.3{*}10^7$ candidates.

Finally, we compared the running time of using the two token-set signature schemes to solve the similarity-join problem and Figure 11 shows results. We can see the algorithm

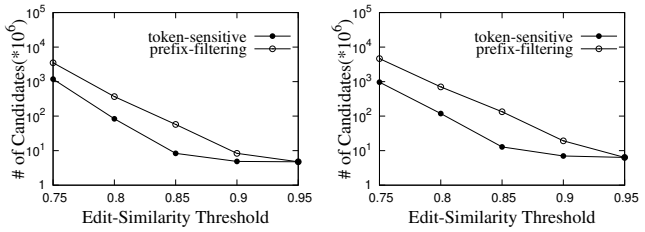


Fig. 10. Comparison of the numbers of candidates between prefix filtering and token-sensitive prefix filtering ($\tau = 0.8$)

using the token-sensitive signature scheme is 3 to 5 times faster than that using the prefix-filtering signature scheme, as the former can remove large numbers of unnecessary token signatures. For example, on the Author dataset, for $\tau = 0.8$, if using the token-sensitive signature scheme, the algorithm took less than 30s, while if using the prefix-filtering signature scheme, the time increased to 130s.

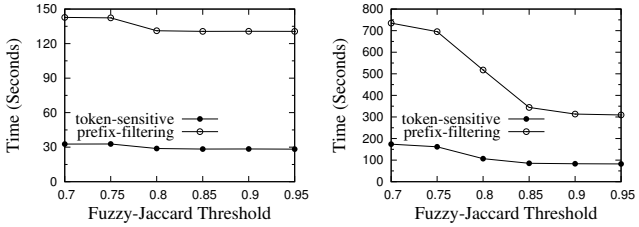


Fig. 11. Performance for different token-set signature schemes ($\delta = 0.85$)

D. Put Everything Together

In this section, we further evaluated the algorithm of solving the similarity-join problem, which included three phases: (1) generating signatures; (2) filtering dissimilar pairs and computing candidates; (3) verifying the candidates to get the final results. We used token-sensitive signature scheme for token sets and Partition-NED for token signatures. Figure 12 shows the results by varying the fuzzy-jaccard threshold τ.

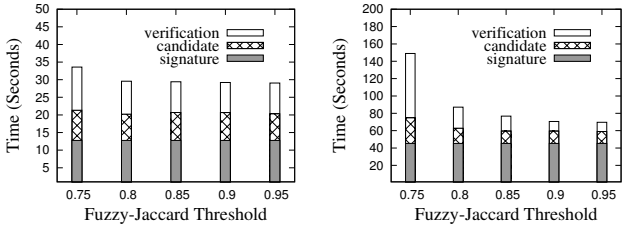


Fig. 12. Performance for different steps ($\delta = 0.85$)

For the Author dataset, three phases took the similar amount of time. For the Query Log dataset, the phase of generating signatures was rather expensive. This is because in the data set the tokens have larger length, which resulted in larger edit-distance thresholds. When τ became smaller the filter and the verification time increased. The reason is that a smaller τ will result in more candidate pairs.

E. Evaluation on Other Similarity Functions

We evaluated the performance of different fuzzy-token similarities, fuzzy-jaccard, fuzzy-dice, and fuzzy-cosine. Figure 13 shows the results. We see that fuzzy-dice and fuzzy-cosine took more time than fuzzy-jaccard. This is because for the same τ, they deduced a smaller fuzzy-overlap threshold than fuzzy-jaccard. We also evaluate the result quality of the three similarities. We find that when fixing the same thresholds δ and τ, fuzzy-jaccard archived higher precision but returned fewer relevant pairs than the other two similarities. For example, when $\delta = 0.85$ and $\tau = 0.8$, fuzzy-jaccard returned 1029 relevant pairs with the precision 95%, while fuzzy-dice returned 3298 pairs with the precision 71% and fuzzy-dice returned 3324 pairs with the precision 70%.

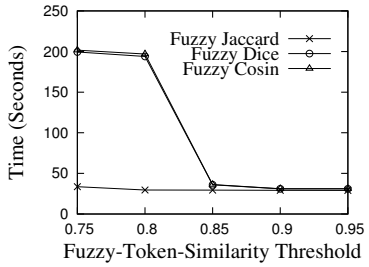


Fig. 13. Performance for different functions on the Author dataset ($\delta = 0.85$)

VII. Related Work

There are some studies on fuzzy token matching based similarity. Chaudhuri et al. [5] proposed generalized edit similarity (GES), which extends the character-level edit operator to the token-level edit operator. However GES is sensitive to token positions. They also derived an approximation of generalized edit similarity (AGES) which ignores the positions of tokens. However AGES does not obey the symmetry property, which may lead to inconsistent results. Our proposed fuzzy-token similarity overcomes these limitations. Arasu et al. [1] proposed a transformation-based framework for similarity join by using functions to define similar pairs, such as synonyms. Jestes et al. [11] studied probabilistic string similarity joins with expected edit distance constrains. However, the two methods need some extra inputs such as string transformations or probabilistic string attributes. In contrast, Fast-Join needs little human effort, and thus is an application-independent method to combine two types of similarity measures. More importantly, our similarity can subsume existing ones. A big benefit of our method is that it can be easily extended to support existing similarity functions.

Jacox et al. [10] studied the metric-space similarity join. The method cannot solve our problem since fuzzy-token similarity does not obey the triangle inequality. Chaudhuri et al. [6] proposed the prefix-filtering signature scheme for effective similarity join. Although the method can be used to solve our problem, it was quite expensive. Therefore, we proposed token-sensitive signature scheme which is proved to be better than the prefix-filtering signature scheme. In the experiment we have extensively compared the two signature schemes. The experimental results also proved our claim.

There are also many other studies on string similarity join [7], [15], [2], [3], [22], [20], [18], [17], which focus on either character-based similarity or token-based similarity, and approximate string searching [14], [8], [13], [9], [23], [12], which given a query string and a set of strings, finds all similar strings of the query string in the string set.

VIII. Conclusion

In this paper we have studied the problem of string similarity join. We proposed a new similarity function by combing token-based similarity and character-based similarity. We proved that existing similarities are special cases of fuzzy-token similarity. We proposed a signature-based framework to address the similarity join using fuzzy-token similarity. We proposed token-sensitive signature scheme, which is superior to the state-of-the-art signature schemes. We extended existing signature schemes for edit distance to support edit similarity. We devised a partition-based token signature scheme and developed pruning techniques to improve the performance. The experimental results on real datasets show that our method achieves high result quality and performance.

IX. Acknowledgement

This work is partly supported by the National Natural Science Foundation of China under Grant No. 61003004 and No. 60873065, the National High Technology Development 863 Program of China under Grant No. 2009AA011906, the National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302206, and National S&T Major Project of China.

References

[1] A. Arasu, S. Chaudhuri, and R. Kaushik. Transformation-based framework for record matching. In *ICDE*, pages 40–49. IEEE, 2008.
[2] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.
[3] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.
[4] D. P. Bertsekas. A simple and fast label correcting algorithm for shortest paths. *Netw.*, 23(7):703–709, 1993.
[5] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD Conference*, pages 313–324, 2003.
[6] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, pages 5–16, 2006.
[7] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.
[8] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, pages 267–276, 2008.
[9] M. Hadjieleftheriou, N. Koudas, and D. Srivastava. Incremental maintenance of length normalized indexes for approximate string matching. In *SIGMOD Conference*, pages 429–440, 2009.
[10] E. H. Jacox and H. Samet. Metric space similarity joins. *ACM Trans. Database Syst.*, 33(2), 2008.
[11] J. Jestes, F. Li, Z. Yan, and K. Yi. Probabilistic string similarity joins. In *SIGMOD Conference*, pages 327–338, 2010.
[12] M.-S. Kim, K.-Y. Whang, J.-G. Lee, and M.-J. Lee. n-Gram/2L: A space and time efficient two-level n-gram inverted index structure. In *VLDB*, pages 325–336, 2005.
[13] H. Lee, R. T. Ng, and K. Shim. Extending q-grams to estimate selectivity of string matching with low edit distance. In *VLDB*, pages 195–206, 2007.
[14] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, 2008.
[15] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, pages 743–754, 2004.
[16] B. S. T. Bocek, E. Hunt. Fast Similarity Search in Large Dictionaries. Technical Report ifi-2007.02, Department of Informatics, University of Zurich, April 2007. http://fastss.csg.uzh.ch/.
[17] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD Conference*, pages 495–506, 2010.
[18] J. Wang, G. Li, and J. Feng. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *PVLDB*, 3(1):1219–1230, 2010.
[19] W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit distance constraints. In *SIGMOD Conference*, 2009.
[20] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 2008.
[21] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, 2009.
[22] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, 2008.
[23] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *SIGMOD Conference*, pages 915–926, 2010.