2019 IEEE 35th International Conference on Data Engineering (ICDE)

MF-Join: Efficient Fuzzy String Similarity Join with Multi-level Filtering

Jin Wang[♯], Chunbin Lin[†], Carlo Zaniolo[♯]
[♯] Computer Science Department, University of California, Los Angeles.
[†] Amazon AWS.
{jinwang,zaniolo}@cs.ucla.edu; lichunbi@amazon.com

***Abstract*—As an essential operation in data integration and data cleaning, similarity join has attracted considerable attention from the database community. In many application scenarios, it is essential to support fuzzy matching, which allows approximate matching between elements that improves the effectiveness of string similarity join. To describe the fuzzy matching between strings, we consider two levels of similarity, i.e., element-level and record-level similarity. Then the problem of calculating fuzzy matching similarity can be transformed into finding the weighted maximal matching in a bipartite graph.**

In this paper, we propose MF-Join, a multi-level filtering approach for fuzzy string similarity join. MF-Join provides a flexible framework that can support multiple similarity functions at both levels. To improve performance, we devise and implement several techniques to enhance the filter power. Specifically, we utilize a partition-based signature at the element-level and propose a frequency-aware partition strategy to improve the quality of signatures. We also devise a count filter at the record-level to further prune dissimilar pairs. Moreover, we deduce an effective upper bound for the record-level similarity to reduce the computational overhead of verification. Experimental results on two popular datasets shows that our proposed method clearly outperforms state-of-the-art methods.

I. Introduction

Similarity join is an essential operation in many applications, such as data cleaning and integration [4], personalized recommendation [5] and near duplicate object elimination [33]. Given two collections of strings, string similarity join aims at finding all similar string pairs from the two collections. To identify similarity strings, most existing studies utilize either character-based similarity functions, e.g., edit distance, or token-based similarity functions, e.g., Jaccard and Cosine, to quantify similarity of two strings.

Furthermore, to allow fuzzy matching between strings, previous study [26] proposed the fuzzy similarity functions, such as Fuzzy-Jaccard and to make a combination of above two categories of similarities. Though these fuzzy similarity functions are effective, there is a limitation in their application scope as they only support character-based functions, i.e. edit similarity, to evaluate the similarity between elements in two strings. However, many real world applications also require supporting token-based similarity metrics between elements in string fuzzy matching. Taking the application of data integration between web tables [21] as an example, we can model each row as a string and use string similarity to identify whether two rows are similar. Given two tables shown in Table I, we want to know whether record R_3= "Michael Franklin, University of Chicago, 5730 S Ellis Avenue" in table $\mathcal{R}$ is similar to record S_2="Michael J. Franklin, 5730 S Ellis Ave" in $\mathcal{S}$. To evaluate the similarity between them, applying either character-based or token-based similarity functions would fail to identify similar rows due to low similarity values. It is also not proper to directly use Fuzzy-Jaccard proposed in [26] since the structural information that each element belongs to a particular table cell will be lost. In this case, it is better to use token-based similarity functions to measure the similarity between elements.

TABLE I
EXAMPLE RECORD SETS $\mathcal{R}$ AND $\mathcal{S}$

Name	Affiliation	Address
Michael Stonebraker	MIT	32 Vassar Street MA
Michael F. Carey	UC Irvine	
Michael Franklin	University of Chicago	5730 S Ellis Avenue

(a) Record set $\mathcal{R}$

FullName	Address	Organization
Michael J. Carey	Irvine CA	UCI
Michael J. Franklin	5730 S Ellis Ave	
Michael Stonebraker	32 Vassar St MA	MIT CSAIL

(b) Record set $\mathcal{S}$

To overcome such limitations, in this paper we generalize the problem of string fuzzy matching into a two-level similarity scheme. Although the fuzzy string matching problem has been studied in previous work [26], we aim at generalizing it in a unified framework, which supports a wider range of similarity functions. Given a string, we model it as a *record* which consists of multiple *element*s. In this way, we enable fuzzy string matching by allowing similar elements to be matched with each other. Here the element-level similarity can be measured with both character-based and token-based functions. Then the record-level similarity serves as the metric to decide the Fuzzy Similarity between two string records. Following previous work [16], [26] related to schema matching and data integration, the problem of measuring Fuzzy Similarity can be modeled as finding the maximum weighted matching of a bipartite graph. Although the recent Silkmoth framework [6] is designed for the problem of relatedness discovery and can also be adopted to support fuzzy string similarity join, it shares with Fast-Join a common performance limitation, i.e., they both adopt q-gram based signatures to filter out dissimilar records. As there are significant overlaps between q-grams, they will produce a large number of candidates and thus reduce the

2375-026X/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDE.2019.00042

filtering power. For fuzzy string similarity join, the verification is much more expensive than that of character and token-based similarity functions. The time complexity of computing Fuzzy Similarity is $\mathcal{O}(n^3)$ where n is the average size of elements in string records. Therefore, it is essential to improve the filter power so as to avoid serious performance issues.

To address this problem, we propose a **M**ulti-level **F**iltering framework MF-Join to efficiently support fuzzy string similarity join. More precisely, we devise a hierarchical filtering strategy to generate candidates from two levels, i.e., element-level and record-level. To identify similar elements, we adopt a partition based approach which generates signatures by splitting each element into disjoint segments. Two elements are similar only if they have a common segment. In this way, we can reduce the cardinality of signatures and strengthen the filter power. To improve the quality of segment signatures, we propose a frequency-aware partition strategy to trim down false positives. In addition, we propose a record-level filter which can prune record pairs that do not share enough similar elements. We also employ an effective upper bound of record-level similarity to enable early termination during verification. Experimental results on widely used datasets demonstrate the superior performance of our proposed techniques.

To sum up, this paper makes the following contributions:

- We propose an efficient framework MF-Join for fuzzy string similarity join, which is flexible to support multiple similarity functions.
- We devise and implement a multi-level filtering framework to enhance the filter power from multiple aspects: (i) a partition based signature in the element-level; (ii) a count based filter in the record-level; (iii) an effective upper bound computation during the process of verification.
- We conduct experiments on two public datasets to evaluate the efficiency of our proposed techniques. The results show that our method obviously outperforms state-of-the-art methods. Since the performance advantage comes from the combination of many optimizations, we also conduct a comprehensive series of experiments to isolate the marginal effect of each individual optimization.

The rest of this paper is organized as following. We formalize the problem and review related work in Section II. We introduce the framework of MF-Join and the signature mechanism in Section III. We propose the element-level filter technique and optimize the signature generation process in Section IV. We devise the record-level filter and improve the verification step in Section V. Necessary discussions are made in Section VI. Experimental results are reported in Section VII. Finally, Section VIII concludes this paper.

II. Preliminaries

We formally define the problem in Section II-A, and survey the related work in Section II-B.

A. Problem Definition

We first introduce some necessary notations to formally describe the fuzzy string similarity join problem. In a string

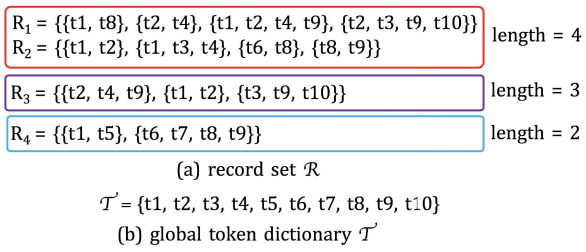


Fig. 1. Example Dataset.

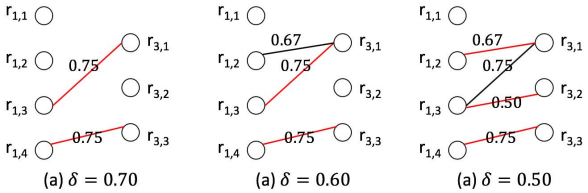


Fig. 2. Bipartite graphs with different δ.

database $\mathcal{R}$, each string $R \in \mathcal{R}$ is called a *record*. The i^{th} record in $\mathcal{R}$ is denoted as R_i. The *record-level similarity* between two records R and S is denoted as $SIM(R, S)$. Each record consists of several *element*s. We denote the i^{th} element of record R as r_i, whereas j^{th} element of R_i is denoted as $r_{i,j}$. And the set of all elements in the dataset is denoted as $\mathcal{E}$. For two elements r_i and s_j, their *element-level similarity* is denoted as $sim(r_i, s_j)$. In this paper, we use Jaccard as the similarity metric for both record-level and element-level. Notice that our framework can support different kinds of similarity metrics at both levels, as will be discussed in Section VI. At the lowest level, each element consists of several *token*s, which is the basic unit of a string record. We construct a global dictionary $\mathcal{T}$ of all tokens in a dataset. The i^{th} token in $\mathcal{T}$ is denoted as t_i. The length of record R_i and element $r_{i,j}$ is denoted as $|R_i|$ and $|r_{i,j}|$, respectively.

Consider the example record set $\mathcal{R}$ in Figure 1(a). Regarding the record length, we have $|R_1| = 4$ and $|R_4| = 2$. Similarly, we have $|r_{1,3}| = 4$. In addition, as there are 10 distinct tokens in $\mathcal{R}$, thus the cardinality of global dictionary $\mathcal{T}$ is 10 as shown in Figure 1 (b).

Following the previous studies [26], [6], we adopt Fuzzy Overlap to enable fuzzy string similarity join. Given two records R, S and an element-level similarity threshold δ, we can construct a bipartite graph $\mathcal{G} = \{X, Y, E\}$ where the nodes in X and Y are elements in R and S, respectively. For any two elements r_i and s_j, if $sim(r_i, s_j) \geq \delta$, there is an edge between the corresponding nodes $< X_i, Y_j > \in E$. The details are shown in Definition 1.

Definition 1 (Fuzzy Overlap): Given two records R, S and an element-level threshold δ, let $\mathcal{G}$ be the corresponding bipartite graph, then the Fuzzy Overlap between R and S, denoted as $R\widetilde{\cap}_\delta S$, is the maximum weighted bipartite matching value over $\mathcal{G}$.

Example 1: Consider records $R_1, R_3 \in \mathcal{R}$ in Figure 1 (a).

When $\delta = 0.7$, there are only two edges in the bipartite graph, i.e., $(r_{1,3}, r_{3,1})$ and $(r_{1,4}, r_{3,3})$ since $sim(r_{1,3}, r_{3,1}) = 0.75 > 0.70$ and $sim(r_{1,4}, r_{3,3}) = 0.75 > 0.70$. The corresponding bipartite graph is shown in Figure 2(a). When decreasing the value of δ, more edges may appear in the bipartite graph. For example, when $\delta = 0.60$, the edge $(r_{1,2}, r_{2,1})$ becomes available (see Figure 2(b)) as $sim(r_{1,2}, r_{2,1}) = 0.67 > 0.60$. Similarly, as shown in Figure 2(c), when $\delta = 0.50$, another edge is added into the graph.

The highlighted edges (in red color) in Figure 2 forms the maximum bipartite matching, which can be adopted to compute Fuzzy Similarity. For example, we have $|R_1 \widetilde{\cap}_{0.50} R_3| = 0.67 + 0.50 + 0.75 = 1.92$ in Figure 2(c). Notice that, in this case, the edge $(r_{1,3}, r_{3,1})$ is not selected since nodes $r_{1,3}$ and node $r_{3,1}$ have been covered by selected edges $(r_{1,2}, r_{3,1})$ and $(r_{1,3}, r_{3,2})$ respectively. ■

With the definition of Fuzzy Overlap, we denote the record-level similarity function with constraint of element-level threshold δ as SIM_δ. Then SIM_δ is named as Fuzzy Similarity, which serves as the metric to evaluate the similarity between string records. If we use *Jaccard* as the record-level similarity function, the Fuzzy Similarity between records R and S can be calculated as following:

$$\text{SIM}_\delta(R, S) = \frac{|R \widetilde{\cap}_\delta S|}{|R| + |S| - |R \widetilde{\cap}_\delta S|} \tag{1}$$

For example, in Figure 2 again, we have: $\text{SIM}_{0.70}(R_1, R_3) = \frac{1.5}{4+3-1.5} = 0.27$, $\text{SIM}_{0.60}(R_1, R_3) = \frac{1.5}{4+3-1.5} = 0.27$, and $\text{SIM}_{0.50}(R_1, R_3) = \frac{1.92}{4+3-1.92} = 0.38$.

Next we formally define our problem in Definition 2.

Definition 2 (Fuzzy String Similarity Join): Given two record set $\mathcal{R}$ and $\mathcal{S}$, the element-level similarity threshold δ and the threshold of Fuzzy Similarity τ, fuzzy string similarity join aims at finding all pairs of records $R \in \mathcal{R}$ and $S \in \mathcal{S}$, where $\text{SIM}_\delta(R, S) \geq \tau$.

In this paper, we focused on the self-join problem where $\mathcal{R} = \mathcal{S}$, but it is very easy to extend our framework to the R-S Join problem.

B. Related Work

String similarity join has been a popular topic in the database community. Two extensive experimental studies are presented in [10] and [15]. Most existing studies adopted a filter-and-verification framework. It has also been adopted in other problems like similarity search [35], [36] and approximate entity extraction [28]. A majority of them utilized q-gram as the signature and are based on prefix filter [4], [1], [2] and count filter [8] technique. To improve the power of prefix filter, the position filter [33] and mis-match filter [31] techniques are proposed. Wang et al. [27] proposed an adaptive framework to dynamically select the length of prefix with the help of a cost model. Some studies proposed other kinds of signatures to improve filter power. Qin et al. [17] adopted an asymmetric signature mechanism to further improve the performance. Li et al. [12] and Wang et al. [29] aimed at reducing the filter cost in the process of probing inverted lists.

Another category of studies utilized disjoint segments as signature and relied on the pigeon hole theory to decide the filtering condition. They have been adopted in string similarity join problem for both character-based [13] and set-based similarity metrics [7]. Arasu et al. [1] proposed the PartEnum method which adopted disjoint chunks as signature to support set similarity join over multiple metrics. Qin et al. [19] generalized the pigeon hole theory to support multiple similarity search problems. Wang et al. [25] adopted similar idea for the similarity search problem. All above studies focused on either token-based or character-based similarity functions and cannot be directly adopted to fuzzy string similarity join.

There are also studies aiming at supporting fuzzy string similarity join. Wang et al. [26] proposed new similarity function by combining token-based and character-based functions and devised Fast-Join algorithm to improve the performance. However, it only supports edit similarity as the similarity metric in element-level. Deng et al. [6] proposed Silkmoth to support relatedness similarity queries, which can also work for the problem of fuzzy string similarity join. Compared with Fast-Join, it can support different kinds of element-level similarity functions like MF-Join did. Our work falls into this category.

Some other studies aimed at addressing string similarity join problem with different similarity metrics and application scenarios. Lu et al. [14] focused on the problem of string similarity join with synonyms. Yang et al. [34] worked on another similarity metric, i.e. set containment. Xiao et al. [32] solved the top-k set similarity join problem. Qin et al. [18] proposed a novel framework for similarity search with Hamming distance as the similarity metrics, which also provided very good insight for other similarity metrics. Vernica et al. [24] and Rong et al. [20] proposed effective distributed algorithms for string similarity join under MapReduce framework. Another line of work is to address the string similarity join problem approximately. The Locality Sensitive Hashing (LSH) technique has been adopted for this problem to speed up the query processing with theoretical guarantee for the error bound [22]. Some studies [3], [23], [37] utilized the similarity idea for nearest neighbor search problems.

III. Overall Framework

In this section, we introduce the overall framework of MF-Join (Section III-A) and the basic filtering mechanism in the element-level (Section III-B).

A. Architecture Overview

We propose a filter-and-verification framework for the problem of fuzzy string similarity join. To filter dissimilar record pairs, we generate a *signature* for each record in the dataset. Given the element-level threshold δ, we denote the signature of record $R \in \mathcal{R}$ and an element $r_i \in R$ as $\Psi_\delta(R)$ and $\Psi_\delta(r_i)$, respectively. Here we have $\Psi_\delta(R) = \uplus_{i=1}^{|R|} \Psi_\delta(r_i)$. To guarantee the correctness of filtering mechanism, given

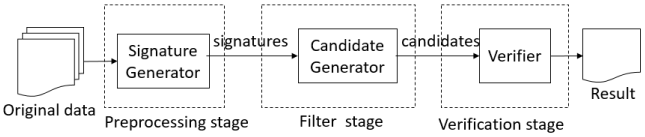


Fig. 3. Overall framework of MF-Join.

two records $R, S \in \mathcal{R}$ and record-level threshold τ, if $\text{SIM}_\delta(R, S) < \tau$, it should satisfy $\Psi_\delta(R) \cap \Psi_\delta(S) = \emptyset$. Correspondingly, given two elements $r_i \in R$, $s_j \in S$ and the element-level threshold δ, if $sim(r_i, s_j) < \delta$, it should satisfy $\Psi_\delta(r_i) \cap \Psi_\delta(s_j) = \emptyset$. Such a signature mechanism allows MF-Join to perform filtering and prune dissimilar record pairs.

For the problem of fuzzy string similarity join, there are two thresholds: the element-level one and the record-level one. Therefore, we propose a two-layer filtering mechanism to fully utilize both thresholds and improve the filter power. For the element-level filter, we generate signatures for each element w.r.t. δ and add them into the inverted index. If two elements has overlapping over their signatures, they become *candidate element pair*. Then their corresponding records could be a potential *candidate record pair*. For the record-level filter, we deduce a stricter filtering condition with the help of τ. Then we can remove more false positives even when they have candidate element pairs.

To sum up, MF-Join consists of three stages: preprocessing, filter and verification as is shown in Figure 3.

- Preprocessing stage. The Signature Generator performs preprocessing to create signatures for all the records. (Section IV)
- Filter stage. The Candidate Generator filters out dissimilar records using both element-level (Section IV) and record-level (Section V-A) filters to generate candidate pairs. The inverted index over the whole dataset will also be constructed during this stage.
- Verification stage. The Verifier calculates Fuzzy Similarity for all candidate record pairs and returns the final results. We also propose an upper bounding technique to enable optimization in finer granularity (Section V-B).

B. Partition based Signature

We first look at how to perform filtering with the element-level similarity threshold δ. To reach this goal, we employ a *partition based signature* to generate the signatures for each element of a record. The basic idea is that we split all elements of a record into several disjoint segments. If two records do not have a common segment, they cannot be similar.

First we introduce how to decide number of segments d. Given two elements r_i, s_j and the element-level level threshold δ, if $\text{JAC}(r_i, s_j) \geq \delta$, then we have $|r_i \cap s_j| \geq \frac{\delta}{1+\delta}(|r_i|+|s_j|)$. Consider the set symmetric difference, i.e. all the different tokens, between r_i and s_j, the cardinality is $|r_i - s_j| + |s_j - r_i|$. According to the length filter, we also have $|s_j| \in [\delta|r_i|, \frac{|r_i|}{\delta}]$. So we have:

$$\begin{aligned} |r_i - s_j| + |s_j - r_i| &\leq |r_i| + |s_j| - \frac{2\delta}{1+\delta}(|r_i| + |s_j|) \\ &= \frac{1-\delta}{1+\delta}(|r_i| + |s_j|) \\ &\leq \frac{1-\delta}{\delta}|r_i| \end{aligned} \quad (2)$$

Therefore, given threshold δ and element r_i, we need to split it into $\mathcal{F}(|r_i|, \delta) = \lceil \frac{1-\delta}{\delta}|r_i| \rceil + 1$ disjoint segments. Then if two elements do not share a common segment, according to the pigeon hole principle, the value of $|r_i \cap s_j|$ cannot reach $\frac{\delta}{1+\delta}(|r_i| + |s_j|)$. Therefore, we can safely assert that the two elements are dissimilar.

To generate the signatures of an element in a record, we first need to guarantee the correctness of segmentation. That is, for a given segment number d, the same token from any element should be allocated to the same segment. Otherwise, there will be false negative in the results. We denote the partition strategy for d segments as P_d. Given an element r_i and a token $t \in r_i$, the segment id token t belongs to w.r.t. P_d is denoted as $P_d(r_i, t)$. And the formal definition of correctness is summarized in Definition 3.

Definition 3 (Correctness of Partition): Given the number of segments d, a partition strategy P_d is correct iff for any two element $r_i, r_j \in \mathcal{E}$ that have a common token t, $P_d(r_i, t) = P_d(r_j, t)$ always holds.

A straightforward way of satisfying Definition 3 is to use hash functions: given a token t and the number of segments d, we allocate t to the $hash(t) \ mod \ d$ segment. Another example of correct partition strategy is *even partition*. To enable this mechanism, we first need to get the global token dictionary $\mathcal{T}$. Then given a segment number d, we evenly split all tokens in $\mathcal{T}$ into d disjoint sub-dictionaries according to their subscriptions. Here the i^{th} sub-dictionary is denoted as $\mathcal{T}_i$. In the even partition, following the idea of [13], the first $\lfloor |\mathcal{T}|/(\lceil \frac{|\mathcal{T}|}{d} \rceil) \rfloor$ sub-dictionaries will hold $\lfloor \frac{|\mathcal{T}|}{d} \rfloor$ tokens while the remaining ones will hold $\lceil \frac{|\mathcal{T}|}{d} \rceil$ tokens. Then given an element $r_i \in \mathcal{E}$, we will enumerate all its tokens. If a token $t \in r_i$ belongs to the $\mathcal{T}_k$, it should be put into the segment with id k [1]. More sophisticated partition schemes will be introduced later, in Section IV.

Example 2: Consider the global token dictionary in Figure 2(b). Assume the segment number d = 3, then the first 2 sub-dictionaries hold 3 tokens, while the last sub-dictionary contains 4 tokens. That is $\mathcal{T}_1 = \{t_1, t_2, t_3\}$, $\mathcal{T}_2 = \{t_4, t_5, t_6\}$, and $\mathcal{T}_3 = \{t_7, t_8, t_9, t_{10}\}$. Consider the element $r_{3,1} = \{t_4, t_2, t_9\}$ in Figure 2(a), token t_2 is all assigned to the segment with id 1, while tokens t_4 and t_9 are assigned to segments 2 and 3, respectively. ■

Finally we can generate the signatures for all the records using the above partition strategies. Given an element $r_i \in R$ and the segment number d, we denote its set of segments as $\mathcal{X}(r_i, d)$. And the signature of R can be obtained as the union of segments from all its elements, where $\Psi_\delta(R) =$

[1]Here empty segment is allowed and it can be handled by our join algorithm.

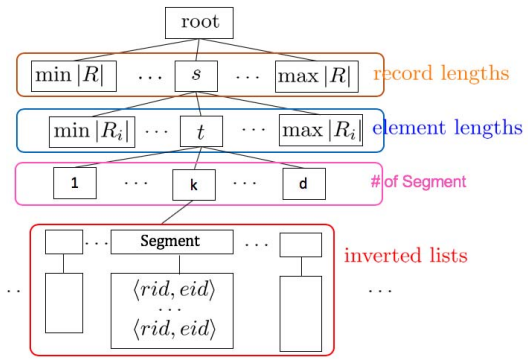


Fig. 4. Hierarchical tree structure.

$\cup_{r_i \in R} \mathcal{X}(r_i, d_i)$, where $d_i = \mathcal{F}(|r_i|, \delta)$. The Element-level filter can be formally concluded as Lemma 1.

Lemma 1 (Element-level Segment Filter): Given two records $R, S \in \mathcal{R}$ and the element-level similarity threshold δ, for elements $r_i \in R$ and $s_j \in S$, if $\Psi_\delta(r_i) \cap \Psi_\delta(s_j) = \emptyset$, then $sim(r_i, s_j) < \delta$. If $\Psi_\delta(R) \cap \Psi_\delta(S) = \emptyset$, R and S cannot be similar.

IV. Effective Fuzzy Join Algorithm

In this section, we propose the MF-Join framework using the element-level filter introduced in Section III. We first introduce the join algorithm based on partition based signature (Section IV-A) and then devise a frequency-aware partition strategy to improve it (Section IV-B).

A. The Join Algorithm with Element-level Filter

In the previous section, we introduce the basic mechanism of partition based signature. Now we propose a join algorithm based on it. In order to help accelerate the join process, we first group records and elements by lengths. Then for each record-element length combination, we construct inverted index for all the segments inside it. Technically, we build a hierarchical tree structure, which contains three components: (i) record length layer, (ii) element length layer, and (iii) inverted list layer. Figure 4 visualizes the hierarchical tree structure. More precisely,

- Each node in the record length layer represents a length of records. The node m points to all the records with length m are grouped in its subtree.
- Each node in the element layer corresponds to a length of elements. The node n points to all the elements whose lengths are n and are inside the records with length m.
- Each inverted list is a key-value pair where the key is a segment and the value is a list of record-id (rid) and element-id (eid) pairs. The inverted lists $\mathcal{L}_{m,n}$ are constructed by the elements with length n contained in the records with length m. The inverted lists belong to the segment with id k in $\mathcal{L}_{m,n}$ are denoted as $\mathcal{L}^k_{m,n}$. The segments in the group $\mathcal{L}^k_{m,n}$ is denoted as $\mathtt{X}^{m,n}_k$

In the filtering stage, we use the first two layers in the hierarchical tree structure to perform *length filter*, and use the inverted list layer to perform element-level filter.

Algorithm 1: MF-Join Framework($\mathcal{R}$, δ, τ)

Input: $\mathcal{R}$: The set of records; δ: The element-level similarity threshold; τ: The record-level similarity threshold

Output: $\mathcal{A}$: The set of similarity record pairs

```
1  begin
2      Initialize result set A, candidate set C as ∅;
3      Group all records by length, traverse R based on the
       lengths in a decreasing order;
4      Sort all tokens in R according to a global order and
       construct the global dictionary T;
5      foreach record R ∈ R do
6          for m ∈ [τ|R|, |R|] do
7              foreach element r_i ∈ R do
8                  for n ∈ [δ|r_i|, |r_i|/δ] do
9                      Split r_i into d_i = F(n, δ) disjoint
                       segments X(r_i, d_i);
10                     for k = 1 to d_i do
11                         Find candidates for segment X^{m,n}_k
                           by traversing L^k_{m,n};
12                     Add candidate record pairs into C;
13             Repartition all r_i ∈ R into F(|r_i|, δ) disjoint
               segments, add them into L_{|R|,|r_i|};
14     for All record pairs ⟨R, S⟩ ∈ C do
15         if Verify(R, S, τ) is True then
16             Add ⟨R, S⟩ into A;
17     return A;
18 end
```

Algorithm 1 demonstrates the similarity join process. We first conduct necessary preparations, i.e. initializing the result set and candidate set, grouping all the records by length and creating the global dictionary of all tokens in the dataset (line: 2 to 4). Then for each record R, we enumerate its elements to generate the candidates. For each element $r_i \in R$, we inspect the inverted lists $\mathcal{L}_{m,n}$ satisfying both the record-level and element-level length filter. Specifically, for the elements with length n, we split r_i into $\mathcal{F}(n, \delta)$ disjoint segments (line: 9). Then we lookup the inverted indexes for these segments to collect candidates (line: 11). Next we repartition r_i into $\mathcal{F}(|r_i|, \delta)$ disjoint segments and add the pairs of rid and eid into the corresponding inverted lists (line: 12). Finally we verify the record-level similarity and add those similar pairs into the result (line: 14).

Complexity The time complexity of Algorithm 1 is analyzed as following. First we need to sort and group the records in $\mathcal{R}$ by record length, in $\mathcal{O}(z \log z)$ time ($z = |\mathcal{R}|$). Then for each record, we need to perform the element level filtering for all its elements to find the candidates. Suppose the average length of all records is $\bar{l_r}$, the average length of all elements is $\bar{l_e}$ and the average size of all inverted lists is $\bar{l}$, the filter cost is $\mathcal{O}(z\bar{l_r}\bar{l_e}\bar{l})$ in total. If the total number of candidate pairs is c, the verification cost is $\mathcal{O}(c\bar{l_r}\bar{l_e}^2)$. And the total

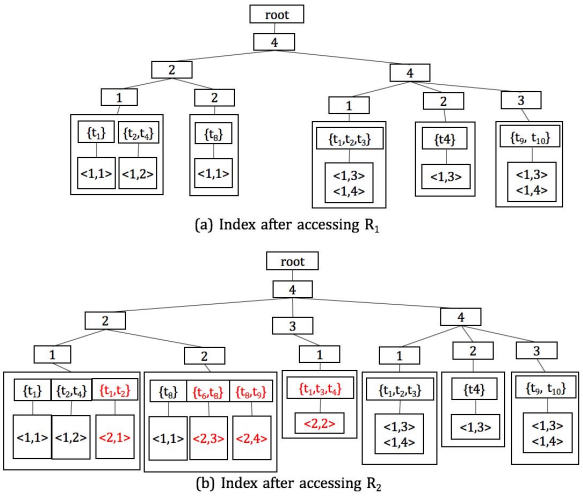


Fig. 5. Example of MF-Join framework.

time complexity can be obtained by adding up them together.

Example 3: Consider the records in Figure 1, assume $\delta = 0.7$. We first access R_1 ($|R_1| = 4$). Inside R_1, each of the first two elements $r_{1,1}, r_{1,2}$ have two tokens, while the last two elements have four tokens. Initially the index is empty. We split $r_{1,1}$ into $\lceil \frac{1-0.7}{0.7} 2 | \rceil + 1 = 2$ segments. Here we use an even partition, thus $\langle 1, 1 \rangle$ is inserted into the inverted list with segment $\{t_1\}$ as its key in the first bucket as well as the inverted list with segment $\{t_8\}$ as its key in the second bucket. This is because t_1 belongs to the first partition while t_8 belongs to the second partition. Figure 5(a) shows the complete index after processing R_1. When accessing element $r_{2,1}$, we need to consider the inverted lists $\mathcal{L}_{m,n}$ where $m \in [2, 4]$ and $n \in [1, 3]$. For example, it needs to access $\mathcal{L}_{4,2}$ but not $\mathcal{L}_{4,4}$. As $\{t_1, t_2\}$ is assigned to the first segment, we get a candidate record pair (R_1, R_2) from $\mathcal{L}_{4,2}$. In addition, after accessing R_2, the index is updated as shown in Figure 5(b). Notice that $r_{2,2}$ has three tokens, so there is a new node with value 3 in the second level in Figure 5(b). Then we access R_3, as it only cares $m \in [2, 3]$, the whole index in Figure 5(c) can be skipped. Similarly, R_4 is also pruned. So the final candidate is $\langle R_1, R_2 \rangle$. ■

B. Frequency-aware Incremental Partition

Previously we have introduced a simple hash function and even partition methods for segmentation. Although an even partition can guarantee that each sub-dictionary has the same number of tokens, it is actually very ineffective as it cannot get rid of the skewness problem. For a given partition strategy P_d, if tokens belonging to the same sub-dictionary appear together frequently in elements, there will be some segments that appear frequently in the whole dataset, and the corresponding inverted lists will be much longer. In this case, such high-frequent segments acts similar as "stop words" in information retrieval, which will result in weaker filter power.

To address this problem, we investigate the partition strategy that can help reduce the number of high-frequent segments. One straightforward way is to find segments with high frequency using an existing frequent pattern mining technique [9] in the preprocessing stage. Then when allocating the tokens into sub-dictionaries, we will try to put tokens belonging to high-frequent segments into different sub-dictionaries and thus make the overall partition balanced. However, this method is rather expensive in time complexity *since the preprocessing time should also be considered as part of the total execution time, such a heavy overhead cannot be accepted*. Also it is difficult to set proper hyper-parameters of the mining algorithm in this scenario. Moreover, collecting and storing the co-occurrence information requires non-trivial time and space overhead. Therefore, we cannot directly adopt the co-occurrence information to construct the sub-dictionaries though its effectiveness.

Nevertheless, we can make use of the token frequency to find a balanced partition following this route. To estimate the co-occurrence of tokens, we approximate it by an assumption: if two tokens appear frequently, then there is also great chance for them to occur in the same segment. Given the segment number d, we denote the total frequency of tokens in $\mathcal{T}_i$ as F_i ($i \in [1 \cdots d]$). Then the higher total frequency a sub-dictionary has, the higher probability it contains high-frequent segments. Therefore, a good partition strategy should make the maximum total frequency among all sub-dictionaries as small as possible. Based on this idea, we can formally define the optimal partition strategy in Definition 4.

Definition 4 (Optimal Partition Strategy): Given the global token dictionary $\mathcal{T}$ and the number of segments d, an optimal partition strategy will split $\mathcal{T}$ into d disjoint sub-dictionaries with the minimum value of $F_{max} = \max_{i \in [1,d]} F_i$.

Theorem 1 (NP-Completeness): Finding an optimal partition strategy is NP-Complete.

Proof [Sketch]: We can prove it by reducing from Subset Sum[2], which is a known NP-Complete problem. ■

Unfortunately, as is proved in Theorem 1, finding an optimal partition is a NP-Complete problem. To address this issue, we design a frequency-aware algorithm to find an approximate solution. The basic idea is that given a token t, we will simply allocate it to the sub-dictionary $\mathcal{T}_{min}$ currently with the minimum total frequency $F_{min} = \min_{i \in [1,d]} F_i$. Then we can construct all d sub-dictionaries by just scanning $\mathcal{T}$ once. Moreover, to improve the quality of partition, we allocate tokens with larger frequency first. Indeed, if a token with large frequency is allocated late when currently all sub-dictionaries have already been balanced, the total frequency of one sub-dictionary will be obviously larger than others. This will increase the value of F_{max}. If we can process such tokens earlier, there will not be such a problem. Therefore, we first sort all tokens of $\mathcal{T}$ in the decreasing order of token frequency before allocation. It is easy to see that this frequency-aware partition strategy satisfied the correctness, which is formally stated in Theorem 2.

[2]https://en.wikipedia.org/wiki/Subset_sum_problem

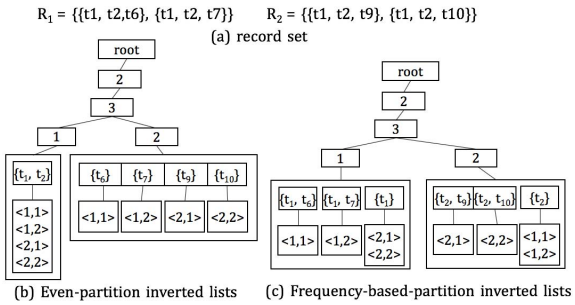


Fig. 6. Even partition vs. frequency-aware partition.

Theorem 2 (Correctness): The frequency-aware partition strategy satisfied the correctness defined in Definition 3.

Example 4: Consider the records in Figure 6(a) and $\delta = 0.7$, we have $d = 2$. With the even partition approach, we will get the inverted lists in Figure 6(b). Notice that the lengths of different inverted lists varied greatly, which causes skewness problem. But if we adopt the frequency-aware partition method, then we will get more balanced inverted lists shown in Figure 6(c). More precisely, we first sort the tokens based on their frequencies: $(t_1, 4), (t_2, 4), (t_6, 1), (t_7, 1), (t_9, 1)$ and $(t_{10}, 1)$. The frequencies of other tokens are all 0. Then we first assign t_1 to the first sub-dictionary as t_1 has the highest frequency. Now $F_1 = 4$. Then we assign t_2 to $\mathcal{T}_2$ as currently $F_2 = 0 < F_1$. We can assign the rest tokens in the similar way. ■

Though the frequency-aware partition strategy is effective, the computation overhead is also heavier. For constructing P_d, we need to keep finding the sub-dictionary with minimum total frequency after allocating each token, which requires $\mathcal{O}(\log d)$ time using priority queue. So the total time is $\mathcal{O}(|\mathcal{T}| \log d)$, which is expensive when making on-line computation for each element. We notice that for elements with the same segment number, the partition strategy is the same. So we can precompute the partition strategies for each segment number d and apply it to each element in the on-line step. Then given an element with l tokens, the time for on-line constructing all its segment is just $\mathcal{O}(l)$.

The performance of preprocessing can be further improved by incrementally constructing P_d from P_{d+1}. The basic idea is that after we constructing P_{d+1}, we can just reallocate the tokens in one sub-dictionary to the remaining d ones. Then we can get P_d by just visiting tokens in the sub-dictionary P_{d+1} instead of re-allocating all tokens in $\mathcal{T}$. And the time complexity will be $\mathcal{O}(|P_{d+1}| \log d)$, where $P_{d+1} \subseteq \mathcal{T}$.

Algorithm 2 demonstrates the process of frequency-aware partition strategy. We start from the maximum segment number $d_\top$. To initialize the algorithm, we sort all tokens in descent order of token frequency (line 2). The total frequency of each group is initialized as 0 (line 3). Next for each token in the global dictionary, we assign it to the group with minimum total frequency (line 5). If there is a tie, we will assign it to the group with the smaller subscription. After assigning a token, we will update the total frequency of the assigned group and

Algorithm 2: Frequency-aware Incremental Partition(x, $\mathcal{T}$, $d_\perp$, $d_\top$)

Input: $\mathcal{T}$: The global token dictionary;
$d_\perp$,$d_\top$: The lower/upper bound of segment number
Output: $\mathcal{P}$: the set of token maps for all segment numbers

```
1  begin
2      Sort all tokens in T in the decreasing order of frequency;
3      Initial d_⊤ sub-dictionaries in P_{d_⊤} as empty;
4      for token t ∈ T do
5          Add t into the partition T_min with minimum total frequency;
6          Update the frequency of T_min and decide the new partition with minimum total frequency;
7      P = P ∪ P_{d_⊤};
8      for d = d_⊤ − 1 to d_⊥ do
9          Construct sub-dictionary P_d;
10         P = P ∪ P_d;
11     return P;
12 end
```

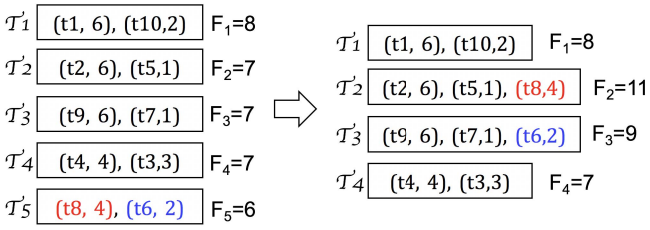


Fig. 7. Example of incremental partition (from $d = 5$ to $d = 4$).

the current group with minimum frequency (line 6). We can obtain $P_{d_\top}$ after allocating all tokens. Then we incrementally create the partition strategy for other segment numbers. We create P_d from P_{d+1} by allocating the tokens in P_{d+1} into the first d sub-dictionaries (line 9). Finally we obtain partition strategies for all segment numbers (line 11).

Example 5: Figure 7 shows an example of creating P_4 from P_5. We allocate the tokens in 5^{th} sub-dictionary (i.e., $\mathcal{T}_5$) of P_5 into the first 4 sub-dictionaries. Inside $\mathcal{T}_5$, there are two tokens t_8 and t_6 with frequencies 4 and 2 respectively. We first move token t_8 into $\mathcal{T}_2$ as currently F_2 has the smallest value[3]. Then we update F_2 from 7 to 11. Similarly, t_6 is relocated to $\mathcal{T}_3$ and F_3 is updated from 7 to 9. ■

V. Further Optimizations

In this section, we propose advanced pruning techniques to further reduce computational overhead. We first devise a record-level filter technique in Section V-A. We then propose an improvement in the verification stage in Section V-B.

[3]Actually, at this point, $F_2 = F_3 = F_4 = 7$, we can choose any one from the $\mathcal{T}_2$, $\mathcal{T}_3$, and $\mathcal{T}_4$.

Finally we discuss how to generalize our framework in Section VI.

A. Record-level Filtering Algorithm

Although the partition based signature has strong filter power, there is still large room for improvement as we have not yet utilized the record-level threshold in pruning. Previously, once two records have a pair of candidate elements, they will be regarded as a candidate record pair. This will result in many false positives as above filter condition never takes the record-level similarity threshold into consideration. To avoid this problem, we further utilize the record-level threshold τ to design effective filtering techniques.

Given a candidate pair $\langle R, S\rangle$ and the record-level threshold τ, it is easy to see that if $\text{SIM}_\delta(R, S) \geq \tau$, the cardinality of $R\widetilde{\cap}_\delta S$ must be no smaller than a particular value $\mathsf{LB}(R, S)$. According to the property of Jaccard, we can deduce such a lower bound as $\mathsf{LB}(R, S) = \lceil(|R| + |S|)\frac{\tau}{1+\tau}\rceil$. Then if the number of potentially matched elements cannot reach $\mathsf{LB}(R, S)$, we can safely prune $\langle R, S\rangle$ even they satisfy the element-level filter.

In order to utilize such information, we need to collect all candidate element pairs for each pair of candidate records. There are some elements that do not have any candidate element and thus make no contribution to increase the value of $|R\widetilde{\cap}_\delta S|$. We call such elements *orphan element*. In the pairs of candidate records returned by the element-level filter, if an element $r_i \in R$ does not have common segment with $\forall s_j \in S$, then it is an orphan element. The formal definition is shown in Definition 5.

Definition 5 (Orphan Element): Given two records $R, S \in \mathcal{R}$ and the element-level threshold δ, if $\forall s_j \in S$, $sim(r_i, s_j) < \delta$. Then we call r_i an Orphan Element of R.

For example, consider two records $R = \{\{t_1, t_2, t_3\}, \{t_4\}\}$ and $S = \{\{t_1, t_2, t_3, t_4\}, \{t_5, t_6\}\}$, assume the element-level threshold $\delta = 0.7$, then the element $r_2 = \{t_4\}$ is an *orphan element*, as $sim(r_2, s_1) = 0.25$ and $sim(r_2, s_2) = 0$.

We can make use of the orphan elements to estimate the upper bound of $|R\widetilde{\cap}_\delta S|$. Given a candidate pair $\langle R, S\rangle$, we denote the number of orphan elements in record R as $\mathcal{N}^R_{\langle R,S\rangle}$. If there is a large number of orphan elements, we can safely prune this pair since there is no possibility that their Fuzzy Similarity can reach τ. We formally stated it in Lemma 2.

Lemma 2 (Record-level Filter): Given two records R, S and the record-level similarity threshold τ, we can get the number of orphan elements $\mathcal{N}^R_{\langle R,S\rangle}$ and $\mathcal{N}^S_{\langle R,S\rangle}$ respectively. If $\min(|R| - \mathcal{N}^R_{\langle R,S\rangle}, |S| - \mathcal{N}^S_{\langle R,S\rangle}) < \lceil(|R| + |S|)\frac{\tau}{1+\tau}\rceil$, then R and S cannot be similar.

Proof: As is shown above, if $\text{SIM}_\delta(R, S) \geq \tau$, we should have the fuzzy overlap $|R\widetilde{\cap}_\delta S| \geq \lceil(|R|+|S|)\frac{\tau}{1+\tau}\rceil$. According to the definition, the upper bound of number of elements in $R(S)$ that potentially has a matched element in $S(R)$ is $|R| - \mathcal{N}^R_{\langle R,S\rangle}$ $(|S| - \mathcal{N}^S_{\langle R,S\rangle})$. Therefore, the upper bound of $|R\widetilde{\cap}_\delta S|$ would be $\min(|R| - \mathcal{N}^R_{\langle R,S\rangle}, |S| - \mathcal{N}^S_{\langle R,S\rangle})$. If this value cannot reach $\lceil(|R| + |S|)\frac{\tau}{1+\tau}\rceil$, we can safely prune $\langle R, S\rangle$. ■

Example 6: Consider the records $R = \{t_1, t_2, t_3\}, \{t_4\}\}$ and $S = \{\{t_1, t_2, t_3, t_4\}, \{t_5, t_6\}\}$ again, we have $\mathcal{N}^R_{\langle R,S\rangle} = 1$ and $\mathcal{N}^S_{\langle R,S\rangle} = 1$ since $r_2 = \{t_4\}$ and $s_2 = \{t_5, t_6\}$ are orphan elements. Thus $\min(|R| - \mathcal{N}^R_{\langle R,S\rangle}, |S| - \mathcal{N}^S_{\langle R,S\rangle}) = \min(2 - 1, 2 - 1) = 1$. If the given threshold $\tau = 0.8$, then $\min(|R| - \mathcal{N}^R_{\langle R,S\rangle}, |S| - \mathcal{N}^S_{\langle R,S\rangle}) = 1 < \lceil(|R| + |S|)\frac{\tau}{1+\tau}\rceil = 3$. According to Lemma 2, $\text{SIM}_\delta(R, S)$ is guaranteed to be smaller than τ (i.e., 0.8), so $\langle R, S\rangle$ is pruned. ■

To utilize Lemma 2, we can just collect the information of candidate elements for each pair of candidate records in line 12 of Algorithm 1. Then in line 15 of Algorithm 1, we will first check whether the candidate pair satisfies Lemma 2. If not, we can directly discard it.

B. Finer Granularity Optimization for Verification

Next we discuss how to improve the verification stage. Given a candidate pair of records $\langle R, S\rangle$, the verification of record-level similarity consists of two steps. The first step is to construct the bipartite graph $\mathcal{G}$ by calculating the element-level similarity between each $r_i \in R$ and $s_j \in S$. If $sim(r_i, s_j) \geq \delta$, there will be a corresponding edge in the bipartite graph. The second step is to perform weighted bipartite matching over $\mathcal{G}$ and obtain the value of $|R\widetilde{\cap}_\delta S|$. This can be done with the help of the well-known KM algorithm [11].

First of all, we can construct the bipartite graph with the help of intermediate results returned by the element-level filter. According to Lemma 1, we will calculate the similarity between two elements only when they share a common segment. If there are y candidate element pairs, we need y calculations. Without such information, we need to verify the element-level similarity between all $|R| * |S|$ pairs of elements. For example, consider two records R and S whose sizes are 4 and 6. Without the help of information of candidates, we need to verify all 24 pairs. Assume there are 3 candidate element pairs, then we just need to verify these three pairs.

The process of performing record-level filter can also be refined by checking whether $sim(R_i, S_j) \geq \delta$ for each candidate pairs of elements. If two elements have common segment, it does not definitely mean that they are similar. By verifying the element-level similarity, we can remove false positives and get the exact values of $\mathcal{N}^R_{\langle R,S\rangle}$ and $\mathcal{N}^S_{\langle R,S\rangle}$.

Next we will show that the expensive weighted bipartite matching step can be avoided after the construction of $\mathcal{G}$. To reach this goal, we need to deduce a upper bound $\mathsf{UB}(R, S)$ of the weighted maximum bipartite matching for candidate pair $\langle R, S\rangle$. If this upper bound is smaller than τ, then the two records cannot be similar. Then we can avoid running the expensive KM algorithm.

To deduce such an upper bound, we utilize the König's Theorem [30]. The basic idea is that for a given bipartite graph, there is an equivalence between the Bipartite Matching and the Vertex Cover problems. Therefore, determining the weighted maximum bipartite matching is equivalent to finding the maximum weighted vertex cover from the graph. As the exact algorithm is as expensive as the KM algorithm, we will use a greedy approximation to find a upper bound of it. That

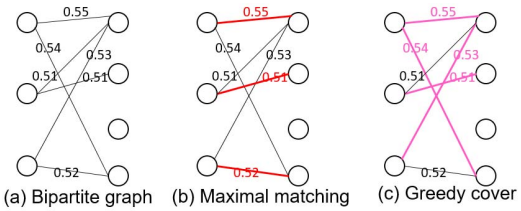


Fig. 8. Upper Bounding the Bipartite Matching Result

is, we first sort all the edges by weights. Then we greedily pick up the maximum weighted edge which can include a new node until all nodes are covered. In this way, we can get an upper bound $\mathsf{UB}(R, S)$ with the selected edges, which is formally stated in Theorem 3.

Theorem 3: Given a bipartite graph $\mathcal{G}$, the upper bound $\mathsf{UB}(R, S)$ can be obtained by above greedy algorithm. If $\mathsf{UB}(R, S) < \frac{\tau(|R|+|S|)}{1+\tau}$, we can safely prune $\langle R, S\rangle$.

Example 7: Given two records R, S with 3 and 4 elements respectively. Assume $\tau = \delta = 0.5$ and we have the bipartite graph shown in Figure8(a). The true value of maximal weighted matching is $0.55 + 0.51 + 0.52 = 1.58$ (shown in Figure 8(b)). By applying the greedy algorithm, we have $\mathsf{UB}(R, S) = 2.13$ (selected edges are colored pink in Figure 8(c)), which is less than $\frac{0.5\times(3+4)}{1+0.5} = 2.3$. Thus, we can safely prune $\langle R, S\rangle$. ■

Compared with our approach, Fast-Join [26] did not apply optimization in the verification stage. Although Silkmoth [6] adopted some refinement strategies after generating candidates, it will directly adopt the KM algorithm to calculate the fuzzy overlap once the bipartite graph is constructed. As our approach can make use of Theorem 3 to avoid the expensive KM algorithm, there will be less overhead in this stage. Finally, we have the whole verification process in Algorithm 3.

VI. Discussion

In this section, we introduce how to generalized our MF-Join framework to different record-level and element-level similarity functions.

A. Support Different Record-Level Similarity Functions

TABLE II
The settings for different record-level similarity functions

Function	Length Filter	Bound for Lemma 2
Jaccard	$[\lvert R\rvert * \tau, \frac{\lvert R\rvert}{\tau}]$	$\lceil(\lvert R\rvert + \lvert S\rvert)\frac{\tau}{1+\tau}\rceil$
Cosine	$[\lvert R\rvert * \tau^2, \frac{\lvert R\rvert}{\tau^2}]$	$\sqrt{\lvert R\rvert * \lvert S\rvert} * \tau$
Dice	$[\lvert R\rvert * \frac{\tau}{2-\tau}, \lvert R\rvert * \frac{2-\tau}{\tau}]$	$\frac{\tau}{2}(\lvert R\rvert + \lvert S\rvert)$

To support different record-level similarity functions, we need different lower bound of $|R\widetilde{\cap}_\delta S|$. In this case, only the filter technique in Lemma 2 and the length filter will be influenced by the record-level similarity function. We show the details in Table II.

Algorithm 3: Verify(R, S, τ, δ)

Input: R, S: Two candidate records; τ: The record-level similarity threshold;
δ: The element-level similarity threshold
Output: Boolean value: whether they are similar.

```
begin
    Construct the bipartite graph G = ⟨R, S, E⟩
    according to the intemediate results of Algorithm 1;
    Initialize set of nodes N = ∅, set of selected edges
    T = ∅;
    Sort all edges in E by weight;
    while |N| ≠ |R| + |S| do
        Find the edge e with maximum weight that can
        increase |N|;
        Add e into T;
        Add nodes associated to e into N;
    Calculate UB(R, S) by adding up the weights of all
    edges in T;
    if UB(R, S) < τ then
        return False;
    Compute SIM_δ(R, S) by weighted bipartite matching
    over G;
    if SIM_δ(R, S) ≥ τ then
        return True;
    return False;
end
```

B. Support Different Element-Level Similarity Functions

For different element-level similarity functions, the number of segments $\mathcal{F}(l, \delta)$ in Algorithm 1 should be different. The cases for Cosine and Dice are straightforward. For Cosine, the number of segments should be $\frac{1-\delta^2}{\delta^2}l$, while for Dice the value is $2\frac{1-\delta}{\delta}l$.

For Edit Similarity as the element-level similarity, we need to first transform the element into a set of q-chunks and then generate the partition based signatures. Specifically, for a string r with length l, the number of q-chunks is $\lceil l/q\rceil$. Given two strings r, $s(|r| > |s|)$ and the threshold of edit similarity δ, as we have $\mathrm{EDS}(r, s) \geq \delta$ and also $\mathrm{EDS}(r, s) < \frac{l}{l+\lceil l/q\rceil-\mathcal{F}(l,\delta)}$. So we can get $\frac{l}{l+\lceil l/q\rceil-\mathcal{F}(l,\delta)} > \delta$. Following this route we can have the bound of $\mathcal{F}(l, \delta)$ as is detailed in Lemma 3.

Lemma 3 (Segment Number for Edit Similarity): Given an element with string length l and the element-level similarity threshold δ, the number of segments used for element-level filter in Algorithm 1 is $\mathcal{F}(l, \delta) = l + \lceil l/q\rceil - \frac{l}{\delta} + 1$.

VII. Evaluation

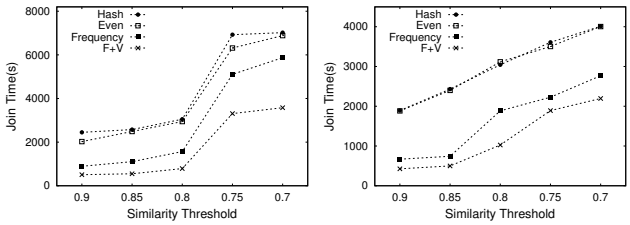
In this section, we conduct an extensive set of experiments to demonstrate the efficiency of our proposed techniques.

A. Experiment Setup

We evaluate our proposed techniques on two real world datasets which have been widely used in related studies.

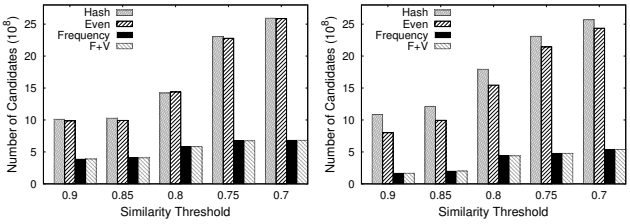
TABLE III
STATISTICS OF DATASETS

Dataset	Cardinality	Avg Record Length	Avg Element Length	Element-level similarity	Record-level similarity
Query Log	1.2 million	3.29	5.64	Edit Similarity	Jaccard
DBLP	1 million	4.77	5.36	Jaccard	Dice



(a) QUERY LOG (b) DBLP

Fig. 9. Effect of Proposed Techniques: Join Time



(a) QUERY LOG (b) DBLP

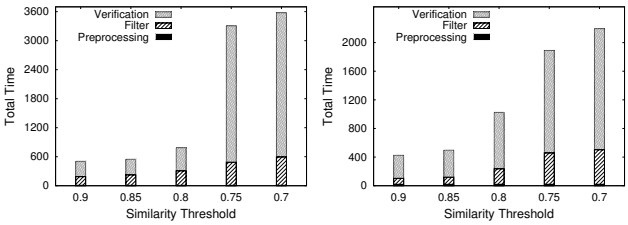
Fig. 10. Effect of Proposed Techniques: Number of Verifications

QUERY LOG [4] is a collection of query log from search engines. Each word in a line is regarded as an element. DBLP [5] contains multiple attributes of the publication, such as title, authors and venue. We use all the non-numerical attributes from the original XML data. Each attribute is regarded as an element and each word in it is regarded as a token. For each dataset, we use different record-level and element-level similarity functions. The detailed information is shown in Table III.

We compare our framework with two state-of-the-art methods Fast-Join [26] and Silkmoth [6]. For Fast-Join, we obtain the code from its authors; as there is no public available code for Silkmoth, we implement it by ourselves. The metric for evaluation is the overall join time. The element-level similarity threshold is set as $\delta = 0.8$ by default. The reason is that if δ is too small, it will involve many dissimilar element pairs and make the join results not reasonable; while a larger δ value is too strict and will filter out many similar records. All experiments are conducted on a server with an Intel Xeon(R) CPU processor, 16 GB RAM, running Ubuntu 14.04.1. All the algorithms are implemented in C++ and compiled with GCC 4.8.4.

B. Effect of Proposed Techniques

We first evaluate the effect of our proposed techniques in this paper. We implement four methods: Hash uses the hash function for segmentation; Even utilizes the even partition for segmentation; Frequency adopts the frequency-aware method for segmentation; F+V further integrates Frequency with the upper bounding technique proposed in Section V-B. The results of join time are shown in Figure 9. We can see that Frequency significantly outperforms Hash and Even as it can avoid frequent segments by properly allocating the tokens. For example, on dataset QUERY LOG when $\tau = 0.85$, Hash and Even took 2575.24 and 2487.63 seconds, respectively. Frequency reduces the time to 1104.52 seconds while F+V only requires 549.32 seconds. The performance of Even is comparable to Hash. The reason is that although Even can partition the global dictionary evenly, it could not avoid the frequent segments. Then the filter power will be hurt and thus lead to poor performance. Compare with Frequency, F+V performs better under all settings. This demonstrates the effectiveness of the proposed verification technique which can avoid the computation of maximum weighted bipartite matching in some cases.



(a) QUERY LOG (b) DBLP

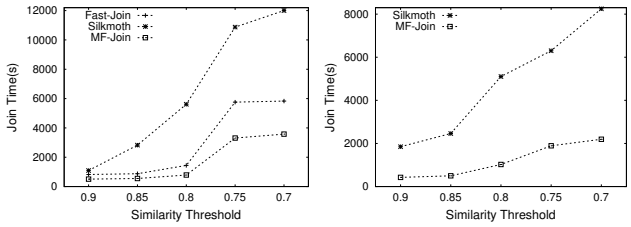
Fig. 11. Join Time Breakdown

We then evaluate the number of candidates to judge the filter power and further demonstrate the effectiveness of proposed techniques. Figure 10 shows the results of above methods. We can see that it is consistent with the results in Figure 9. For example, on dataset DBLP when $\tau = 0.8$, the number of candidates for Hash, Even and Frequency is 1722094749, 1538984732 and 482984711, respectively. Although F+V cannot reduce the number of candidates, it can terminate the verification stage after constructing the bipartite graph. Thus it can also help improve the performance.

We also break down the overall join time into three parts to make further analysis. Here we report the results of our best method F+V in Figure 9. The detailed execution time of preprocessing, filtering and verification stage is shown in Figure 11. We can see that the preprocessing time is trivial compared with the filtering and verification stage. In most cases the verification time dominates the overall join time. This result makes sense as the verification of fuzzy matching is much more expensive than other simple similarity functions. It also demonstrates the requirement of devising effective filter techniques to reduce false positive. As shown above, the filter

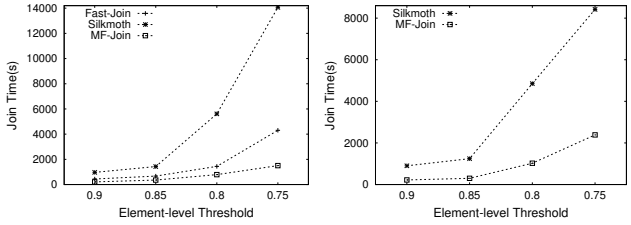
[4]http://www.gregsadetsky.com/aol-data/

[5]https://dblp.uni-trier.de/xml/



(a) QUERY LOG (b) DBLP

Fig. 12. Compare with State-of-the-art Methods: Varying τ



(a) QUERY LOG (b) DBLP

Fig. 13. Compare with State-of-the-art Methods: Varying δ

cost is reasonable and will not lead to much overhead while keeping strong filter power.

C. Comparing with State-of-the-art Methods

We compare our method with state-of-the-art methods Fast-Join and Silkmoth. As Fast-Join only supports Edit Similarity as the element-level similarity function, we only compare with it on QUERY LOG dataset. Here MF-Join is the F+V method introduced before. From the results shown in Figure 12, we have the following observations:

First, we can see that MF-Join achieves the best results on all settings and has obvious performance gain over existing methods. For example, on dataset QUERY LOG when $\tau = 0.75$, the total join time for Fast-Join, Silkmoth and MF-Join is 5758.02, 10873.22 and 3308.67 seconds, respectively. Second, on the QUERY LOG dataset, Fast-Join performs better than Silkmoth although both of them adopts q-gram based signatures. One reason can be that Fast-Join has devised the token sensitive signature which can further reduce the number of candidates while Silkmoth just directly utilized the variants of q-grams. Third, MF-Join outperforms Silkmoth under all settings. This can be attributed to the two-fold reasons. On the one hand, we adopt partition based signature in the element-level, which could reduce the number of signatures for all records and thus lead to better filter power. On the other hand, we enable the early termination in the verification stage by deducing an upper bound of record-level similarity. Then we can avoid doing bipartite matching even if we have constructed the bipartite graph. Therefore, MF-Join can save more unnecessary computation compared with Silkmoth.

In the next experiment, we vary the value of element-level similarity threshold δ and report the overall join time. For this experiment, we fix the value of record-level similarity threshold τ as 0.8. The results are shown in Figure 13. We can see that with varying value of δ, our method still performs better than state-of-the-art methods. The general trend of performance comparison is similar with those in Figure 12. The benefits mainly come from the effect of partition based signatures, which have greater filter power for different element-level similarity thresholds.

TABLE IV
INDEX SIZE (MB) WHEN $\delta = 0.8$

Dataset	Fast-Join	Silkmoth	MF-Join
Query Log	106.7	84.2	59.7
DBLP	99.2	81.6	56.18

As the major memory consumption comes from the inverted index, we also evaluate the index size of different methods. The results are shown in Table IV. We can see that among all the methods, MF-Join has the minimum index size. The reason is that as we adopt the partition based signature in the element-level, our signatures are disjoint segments. Compared with q-grams that have more overlapping with each other, MF-Join has much fewer signatures. Therefore, the total index size will also be smaller. The reason that Silkmoth has smaller index size than Fast-Join might be that as Fast-Join utilizes the token sensitive signature, it needs more space to record the related information in the inverted lists.

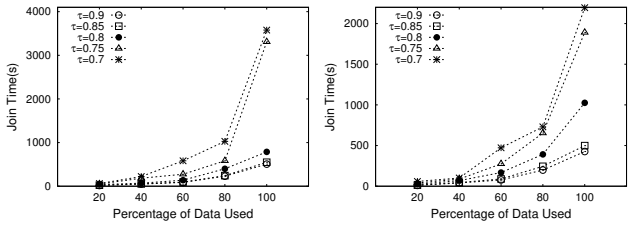
D. Evaluation of Effectiveness

Next we look at the effectiveness of Fuzzy Similarity. Since the effectiveness of Fuzzy-Jaccard on QUERY LOG dataset has been justified by [26], here we just look at the case of DBLP. We compare the result quality of our similarity function, i.e., Fuzzy Similarity with Jaccard and Dice similarity by evaluating the number of results and precision with different record-level threshold τ ranging from 0.75 to 0.95. We follow the settings in [26] to choose $100,000$ records from the DBLP dataset and calculate the precision based on 100 randomly selected similar pairs from the produced results. Table V reports the quality results of Jaccard, Dice similarity and Fuzzy Similarity (with element-level threshold $\delta = 0.8$).

TABLE V
RESULT QUALITY.

τ	Jaccard		Dice		Fuzzy Similarity	
	# Results	Precision	# Results	Precision	# Results	Precision
0.75	975	84%	449	35%	2144	86%
0.80	623	91%	302	39%	1589	92%
0.85	482	94%	228	48%	1132	99%
0.90	313	97%	146	53%	653	99%
0.95	204	100%	113	61%	327	100%

We can make following observations: Firstly, Fuzzy Similarity generates more similar string pairs than others under the same threshold settings. This is because Fuzzy Similarity considers two levels of similarities, i.e., element-level and record-level similarities. For example, when $\tau = 0.80$, Fuzzy Similarity outputs 1589 similar pairs, while Jaccard and Dice only return 623 and 302 pairs respectively. Secondly, Fuzzy Similarity achieves the highest precision. For example, when $\tau = 0.90$, the precision values of Fuzzy Similarity is 99% while those of Jaccard and Dice are 97% and 53%.



(a) QUERY LOG (b) DBLP

Fig. 14. Scalability

E. Scalability

Finally we evaluate the scalability. We vary the number of records in each dataset and report the overall join time. The results are shown in Figure 14. We can see that the join time increases steadily with the increasing number of records for all thresholds. For example, on DBLP for $\tau = 0.75$. When the size of dataset scales from 20% to 100%, the corresponding total join time is 49.5, 183.65, 274.39, 582.3 and 3308.67 seconds. This shows its potential scalability on larger datasets.

VIII. CONCLUSION

In this paper, we introduced MF-Join, an efficient framework for fuzzy string similarity join which flexibly supports multiple similarity functions. We devised and implemented a multi-level filtering mechanism to enhance the filter power from multiple aspects. Specifically, we proposed the partition based signature at the element-level as well as a frequency-aware partition strategy to improve it. We devised a count-based filter at the record-level to further prune dissimilar candidates. We also deduced an effective upper bound for fuzzy similarity to reduce the computation overhead in the verification stage. Experimental results on two popular datasets demonstrate the efficiency of our proposed techniques.

REFERENCES

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

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

[3] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *STOC*, pages 327–336, 1998.

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

[5] A. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *WWW*, pages 271–280, 2007.

[6] D. Deng, A. Kim, S. Madden, and M. Stonebraker. Silkmoth: An efficient method for finding related sets with maximum matching constraints. *PVLDB*, 10(10):1082–1093, 2017.

[7] D. Deng, G. Li, H. Wen, and J. Feng. An efficient partition based method for exact set similarity joins. *PVLDB*, 9(4):360–371, 2015.

[8] 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.

[9] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, pages 1–12, 2000.

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

[11] H. W. Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

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

[13] G. Li, D. Deng, J. Wang, and J. Feng. PASS-JOIN: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[14] J. Lu, C. Lin, W. Wang, C. Li, and H. Wang. String similarity measures and joins with synonyms. In *SIGMOD*, pages 373–384, 2013.

[15] W. Mann, N. Augsten, and P. Bouros. An empirical evaluation of set similarity join techniques. *PVLDB*, 9(9):636–647, 2016.

[16] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *ICDE*, pages 117–128, 2002.

[17] J. Qin, W. Wang, Y. Lu, C. Xiao, and X. Lin. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *SIGMOD*, pages 1033–1044, 2011.

[18] J. Qin, Y. Wang, C. Xiao, W. Wang, X. Lin, and Y. Ishikawa. GPH: similarity search in hamming space. In *ICDE*, pages 29–40, 2018.

[19] J. Qin and C. Xiao. Pigeonring: A principle for faster thresholded similarity search. *PVLDB*, 12(1):28–42, 2018.

[20] C. Rong, C. Lin, Y. N. Silva, J. Wang, W. Lu, and X. Du. Fast and scalable distributed set similarity joins for big data analytics. In *ICDE*, pages 1059–1070, 2017.

[21] A. D. Sarma, L. Fang, N. Gupta, A. Y. Halevy, H. Lee, F. Wu, R. Xin, and C. Yu. Finding related tables. In *SIGMOD*, pages 817–828, 2012.

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

[23] Y. Sun, W. Wang, J. Qin, Y. Zhang, and X. Lin. SRS: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *PVLDB*, 8(1):1–12, 2014.

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

[25] J. Wang, G. Li, D. Deng, Y. Zhang, and J. Feng. Two birds with one stone: An efficient hierarchical framework for top-k and threshold-based string similarity search. In *ICDE*, pages 519–530, 2015.

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

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

[28] J. Wang, C. Lin, M. Li, and C. Zaniolo. An efficient sliding window approach for approximate entity extraction with synonyms. In *EDBT*, 2019.

[29] X. Wang, L. Qin, X. Lin, Y. Zhang, and L. Chang. Leveraging set relations in exact set similarity join. *PVLDB*, 10(9):925–936, 2017.

[30] D. B. West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

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

[32] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, 2009.

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

[34] J. Yang, W. Zhang, S. Yang, Y. Zhang, and X. Lin. Tt-join: Efficient set containment join. In *ICDE*, pages 509–520, 2017.

[35] Y. Zhang, X. Li, J. Wang, Y. Zhang, C. Xing, and X. Yuan. An efficient framework for exact set similarity search using tree structure indexes. In *ICDE*, pages 759–770, 2017.

[36] Y. Zhang, J. Wu, J. Wang, and C. Xing. A transformation-based framework for knn set similarity search. *IEEE Trans. Knowl. Data Eng.*, 2019.

[37] Y. Zheng, Q. Guo, A. K. H. Tung, and S. Wu. Lazylsh: Approximate nearest neighbor search for multiple distance functions with a single index. In *SIGMOD*, pages 2023–2037, 2016.