TokenJoin: Efficient Filtering for Set Similarity Join with Maximum Weighted Bipartite Matching

Alexandros Zeakis
National and Kapodistrian University of Athens & "Athena" RC
Greece
alzeakis@di.uoa.gr

Dimitrios Skoutas
"Athena" RC
Greece
dskoutas@athenarc.gr

Dimitris Sacharidis
Université Libre de Bruxelles
Belgium
dimitris.sacharidis@ulb.be

Odysseas Papapetrou
Eindhoven University of Technology
Netherlands
o.papapetrou@tue.nl

Manolis Koubarakis
National and Kapodistrian University of Athens
Greece
koubarak@di.uoa.gr

ABSTRACT

Set similarity join is an important problem with many applications in data discovery, cleaning and integration. To increase robustness, fuzzy set similarity join calculates the similarity of two sets based on maximum weighted bipartite matching instead of set overlap. This allows pairs of elements, represented as sets or strings, to also match approximately rather than exactly, e.g., based on Jaccard similarity or edit distance. However, this significantly increases the verification cost, making even more important the need for efficient and effective filtering techniques to reduce the number of candidate pairs. The current state-of-the-art algorithm relies on similarity computations between pairs of elements to filter candidates. In this paper, we propose token-based instead of element-based filtering, showing that it is significantly more lightweight, while offering similar or even better pruning effectiveness. Moreover, we address the top-k variant of the problem, alleviating the need for a user-specified similarity threshold. We also propose early termination to reduce the cost of verification. Our experimental results on six real-world datasets show that our approach always outperforms the state of the art, being an order of magnitude faster on average.

PVLDB Reference Format:
Alexandros Zeakis, Dimitrios Skoutas, Dimitris Sacharidis, Odysseas Papapetrou, and Manolis Koubarakis. TokenJoin: Efficient Filtering for Set Similarity Join with Maximum Weighted Bipartite Matching. PVLDB, 16(4): 790 - 802, 2022.
doi:10.14778/3574245.3574263

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/alexZeakis/TokenJoin.

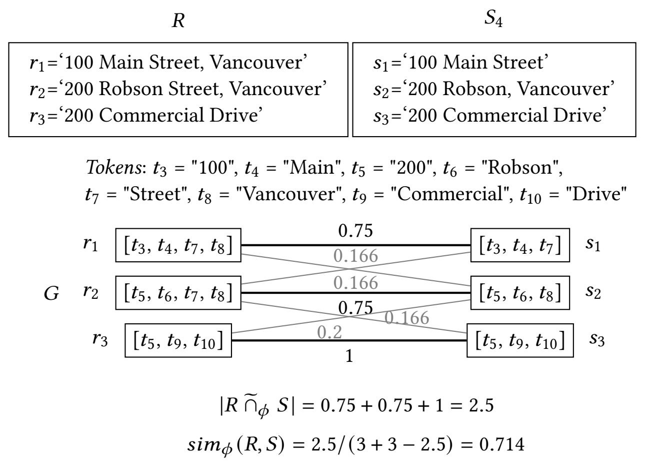
Proceedings of the VLDB Endowment, Vol. 16, No. 4 ISSN 2150-8097.
doi:10.14778/3574245.3574263

1 INTRODUCTION

Set similarity join computes all pairs of sets in a collection or between two collections having similarity score above a given threshold. It is a fundamental task in data discovery, cleaning and integration. For example, it can be used to match names, addresses, publications, social media posts, etc. Traditional approaches measure the similarity between two sets based on set overlap, i.e., the number of elements they have in common [7, 10, 13, 15]. This only considers identical elements, thus it is not robust to misspellings or other variations. To overcome this limitation, some works [6, 22, 23] have proposed *fuzzy* set similarity join, where the similarity between two sets R and S is based on fuzzy overlap [29]. The elements of R and S form a bipartite graph G with edge weights representing element similarities. Then, the similarity of R and S is measured based on the maximum weighted matching in G. The latter is a one-to-one mapping between nodes that maximizes the sum of edge weights [8]. In the fuzzy setting, each element itself is represented as a set (e.g., by splitting a phrase into words or a word into q-grams), thus allowing approximate matching between elements.

Example 1. *Figure 1 shows two sets R and S_4 (part of the running example shown later in Figure 2). Each set consists of three elements, with each element representing a street address. There is only one identical element between R and S, hence their Jaccard similarity is $Jac(R, S) = 1/5 = 0.2$. In the fuzzy setting, each element is split into a set of tokens, and the bipartite graph G is constructed. The maximum weighted matching consists of the edges shown in bold. In addition to (r_3, s_3), this includes the element pairs (r_1, s_1) and (r_2, s_2), which have Jaccard similarity 0.75 based on their tokens. The matching score between R and S is $|R \widetilde{\cap}_\phi S| = 2.5$, which results in a similarity score $sim_\phi(R, S) = 0.714$ (see Section 3 for formal definitions).*

Algorithms for set similarity join (both traditional and fuzzy) follow a filter-verification framework to reduce the number of pairwise comparisons between sets. The filtering phase applies one or more filters to prune candidate pairs. Then, the remaining pairs are verified to determine whether their similarity score exceeds the given threshold. Filtering involves a tradeoff between efficiency and effectiveness. More elaborate filters may prune more candidates, but the extra overhead of the filter itself may not pay off in practice [13]. Hence, the main challenge is to design filtering algorithms that are



R
S_4
r_1='100 Main Street, Vancouver'
r_2='200 Robson Street, Vancouver'
r_3='200 Commercial Drive'
s_1='100 Main Street'
s_2='200 Robson, Vancouver'
s_3='200 Commercial Drive'
Tokens: t_3 = "100", t_4 = "Main", t_5 = "200", t_6 = "Robson", t_7 = "Street", t_8 = "Vancouver", t_9 = "Commercial", t_10 = "Drive"
G
r_1 [t_3, t_4, t_7, t_8]
r_2 [t_5, t_6, t_7, t_8]
r_3 [t_5, t_9, t_10]
s_1 [t_3, t_4, t_7]
s_2 [t_5, t_6, t_8]
s_3 [t_5, t_9, t_10]
0.75
0.166
0.166
0.75
0.166
0.2
1

$$|R \,\widetilde{\cap}_{\phi}\, S| = 0.75 + 0.75 + 1 = 2.5$$

$$sim_{\phi}(R, S) = 2.5/(3 + 3 - 2.5) = 0.714$$

Figure 1: Example showing set similarity based on maximum weighted bipartite matching.

simultaneously efficient and effective. This becomes even more crucial in the fuzzy setting, where the verification cost is much higher compared to set overlap. Specifically, the complexity of computing the maximum weighted bipartite matching is $O(n^3)$, where n refers to the number of elements in each set [8].

The current state-of-the-art algorithm for fuzzy set similarity join is SilkMoth [6]. To generate candidates for a set R, SilkMoth constructs a signature, which consists of a relatively small subset of its tokens. If S contains at least one of these signature tokens, it is a candidate. Subsequently, to refine candidates, SilkMoth proposes two filters that are applied sequentially. The first, called *Check Filter* (CF), computes the similarity score of all pairs of elements $r \in R$ and $s \in S$ that have a common signature token. The second, called *Nearest Neighbor Filter* (NNF), identifies for each element $r \in R$ its most similar element $s \in S$.

SilkMoth places a lot of emphasis on optimal signature construction, aiming to use *as few tokens as possible*. This seems a reasonable goal, since a smaller signature generates fewer candidates. However, it leads to two important shortcomings. First, every set S that contains at least one signature token is accepted as a candidate and needs to be processed by the subsequent refinement filters. Second, these filters rely on *similarity computations between entire elements*, which incurs a high processing cost. We show that both of these can be avoided.

Instead of focusing on the design of optimal signatures, we introduce the concept of *token utility*. Intuitively, since the similarity score of two elements depends on their common tokens, tokens hold sufficient information to prune candidates. By capturing this, token utility enables a purely token-based filtering approach that completely avoids the computation of similarity scores between pairs of elements. If a token t is present in R but not in S, we directly use the utility of t to establish an upper bound on the similarity score between R and S.

Overall, our approach can prune more candidates with fewer computations. First, keeping track of token utilities during candidate generation allows us to discard a large portion of candidates before refinement. In our experiments, the number of candidates entering the refinement stage is around 80% lower compared to SilkMoth. Second, during refinement, we keep processing individual tokens instead of computing similarity scores over entire elements. This is significantly less expensive, while retaining comparable pruning power. Due to these advantages, our method always outperforms SilkMoth, being an order of magnitude faster on average.

Nevertheless, we observe that verification still constitutes a bottleneck for large sets. To mitigate this, we propose two early termination criteria, based on upper and lower bounds, which accelerate the identification of false positives and true positives, respectively. Finally, we propose a top-k algorithm, which does not require a user-specified similarity threshold. To the best of our knowledge, this is the first algorithm for top-k fuzzy set similarity join.

Our main contributions are as follows:

- We propose a novel filtering approach for set similarity join with maximum weighted bipartite matching, which is purely token-based. To this end, we define and use the notion of token utility. Our method supports both Jaccard similarity and normalized edit similarity between elements.
- We introduce three token-based filtering criteria for candidate refinements. Given a candidate pair of sets (R, S), the first utilizes information only from tokens contained in R, while the latter combine token utilities from both R and S.
- We address the top-k fuzzy set similarity join problem. We first derive two baseline algorithms by adapting previous works to this problem. We then show how to extend our token-based filtering algorithm, using candidate prioritization and threshold initialization.
- We describe how to accelerate verification using two early termination conditions, employing upper and lower bounds to determine whether a candidate pair constitutes a match.
- We conduct a comprehensive experimental evaluation, comparing our proposed algorithms against respective baselines using six real-world datasets. The results show that our token-based filtering technique is significantly more efficient compared to element-based filtering.

The rest of the paper is structured as follows. Section 2 discusses related work on set similarity joins. Section 3 formally defines the problem. Section 4 introduces token utilities and the obtained upper bounds on element and set similarities. Section 5 describes our threshold-based join algorithm, while Section 6 extends our approach to top-k join. Section 7 shows how to accelerate verification using early termination. Finally, Section 8 presents our experimental evaluation, and Section 9 concludes the paper.

2 RELATED WORK

Set similarity join is a fundamental problem that has been extensively studied in the literature [7, 10, 13, 15, 20, 29]. Proposed methods typically follow the filter-verification framework. One or more filters are applied to generate and prune candidates. The remaining ones are verified to find the true matches. Different filters provide a tradeoff between efficiency and effectiveness. More elaborate ones may prune more candidates but the extra overhead may not pay off [13]. Hence, the main challenge is to devise filtering techniques that are both efficient and effective. Below, we outline existing works for traditional and fuzzy set similarity join.

Traditional Set Similarity Join. These works consider similarity functions based on set overlap (e.g., Jaccard, Cosine, Dice) [4, 17, 18, 21, 25, 28]. There also exist works that address string similarity join with character-based similarity functions (e.g., edit distance) [9, 16, 24, 26]. Nevertheless, string similarity join can be transformed to set similarity join by representing each string as a set of q-grams.

The *size filter* [1] is based on the fact that similar sets must have similar sizes. If two sets r and s have Jaccard similarity not lower than δ, then their sizes must satisfy the condition $\delta \cdot |s| \leq |r| \leq |s|/\delta$. Being both simple and effective, it is employed by most methods as the first filter in a sequence of increasingly more complex filters.

Another very common filter is the *prefix filter* [2, 4]. If two sets r and s have Jaccard similarity not lower than δ, then their prefixes must contain at least one common token. The prefix of a set r refers to its first π_r items under a global ordering, where $\pi_r = \lfloor (1-\delta) \cdot |r| \rfloor + 1$. If the prefixes have no common tokens, then even if all remaining tokens match, the Jaccard similarity cannot exceed the specified threshold. This filter reduces the number of candidates by only considering tokens in the prefix of a set. Any token ordering can be used. In practice, it is preferable to sort the tokens in increasing order of their frequency in the collection, so that prefixes contain less frequent tokens.

Extensions to prefix filter have also been proposed. PPJoin [28] proposes *positional filtering*, which additionally considers the positions where the common tokens in the prefix occur. PPJoin+ [28] proposes *suffix filtering*, which partitions the suffix of a set into two subsets and calculates the maximum number of potentially matching tokens in each one. GroupJoin [3] groups together all sets with same prefixes to prune multiple candidates in batch. AdaptJoin [21] extends the prefix size by $n-1$. It then prunes sets that contain less than n common tokens in their extended prefixes.

Distributed algorithms have been proposed for scaling to larger datasets [7]. Approximate algorithms have been proposed for low similarity thresholds [5, 19, 31]. Other works focus on similarity search over a collection that has been indexed offline [12, 32, 33]. JOSIE [34] addresses set similarity search for finding joinable tables in data lakes. Finally, a top-k algorithm has been presented in [27], which prioritizes candidate generation and verification.

Fuzzy Set Similarity Join. Comparing sets based on maximum weighted bipartite matching, as opposed to set overlap, is more robust to misspellings or other variations. It allows elements to match approximately rather than exactly, which can be considered as a hybrid similarity function based on fuzzy overlap [29]. Since this increases the verification cost, the performance of the employed filters becomes even more crucial.

The first method to address this problem was FastJoin [22]. Its main idea is that if two sets have a bipartite matching score at least θ, then they must contain at least $\lceil \theta \rceil$ elements with similarity score higher than 0. Since these elements must share at least one common token, there must be at least $\lceil \theta \rceil$ common tokens. Hence, given a set R that contains n tokens, any selection of $n - \lceil \theta \rceil + 1$ tokens from R can be used as a signature for R. If a set S does not contain any of these signature tokens, it cannot match with R. Essentially, this adapts the idea of prefix filtering to the fuzzy setting.

SilkMoth [6] optimizes this signature scheme, reducing the number of used tokens to generate fewer candidates. Any set S that contains a signature token is then processed by the subsequent refinement filters. These include the Check Filter (CF) and the Nearest Neighbor Filter (NNF), which both rely on element comparisons. CF calculates for each element $r \in R$ an individual threshold θ_r. It then computes the similarity score of all pairs of elements (r, s) that contain a common signature token. If none of these pairs has similarity at least θ_r, S is pruned. Then, NNF finds for each element $r \in R$ its most similar element $s \in S$ and assigns r to it. Although this may not be a valid matching, since multiple elements of R may have the same nearest neighbor in S, it provides an upper bound for the matching score between R and S. If this is lower than the threshold, S is pruned.

MF-Join [23] addresses a different variant of the problem, requiring two similarity thresholds δ_1 and δ_2 as input, which apply to sets and elements, respectively. Only element pairs with similarity score at least δ_2 may participate in the bipartite matching. The algorithm uses the element-level threshold δ_2 to filter candidates. However, manually selecting appropriate values for similarity thresholds is often not straightforward in practice, especially tuning two thresholds simultaneously. As opposed to that, we address the top-k problem variant. Specifying the number of results to be returned is more intuitive than choosing similarity thresholds.

3 PROBLEM DEFINITION

We address the problem of fuzzy set similarity join, where the similarity of two sets R and S is based on the maximum weighted matching in the bipartite graph G corresponding to their elements. In contrast to traditional set similarity join, which relies on set overlap, this allows elements of R and S to match approximately.

Assume a collection $\mathcal{D}$ of sets, where each set $R \in \mathcal{D}$ comprises one or more elements. Each element $r \in R$ further consists of a set of tokens $t \in r$. For example, a set R may represent the title of a publication, an element $r \in R$ a word contained in the title, and a token $t \in r$ a q-gram contained in that word. We use T_R to denote the multiset of all tokens appearing in the elements of R, and T_{RS} to denote the common tokens between R and S.

Set Similarity. Given two sets R and S, we construct a bipartite graph $G = ((V_R, V_S), E)$, where the nodes in V_R and V_S correspond to the elements of R and S, respectively. Each edge $(v_r, v_s) \in E$ is associated with a weight $\phi(r, s) \in [0, 1]$, where ϕ is a function that measures element similarity. Following previous works [6, 22], we define set similarity as described below. An illustrative example has been presented in Figure 1.

Definition 1 (Maximum Weighted Bipartite Matching). *Assume a bipartite graph G representing a pair of sets (R, S). The maximum weighted matching score, denoted by $|R \,\widetilde{\cap}_\phi\, S|$, is defined as the maximum sum of weights of a subset of edges in G such that no two edges have a common vertex in V_R or V_S.*

Definition 2 (Set Similarity). *Given two sets R and S, and a similarity function ϕ between their elements, we compute the similarity score of the pair (R, S) as follows:*

$$sim_\phi(R, S) = \frac{|R \,\widetilde{\cap}_\phi\, S|}{|R| + |S| - |R \,\widetilde{\cap}_\phi\, S|} \tag{1}$$

$|R \widetilde{\cap}_\phi S|$ is also referred to as fuzzy overlap, and $sim_\phi(R, S)$ as fuzzy Jaccard similarity [29]. As will be explained later, our algorithms translate a given threshold δ on $sim_\phi(R, S)$ to an equivalent threshold θ on $|R \widetilde{\cap}_\phi S|$. Then, candidates are filtered by establishing upper bounds on $|R \widetilde{\cap}_\phi S|$. Therefore, although we focus our discussion on fuzzy Jaccard, our methods can be similarly applied to other functions that are based on fuzzy overlap, including fuzzy Cosine and fuzzy Dice similarities, which are defined as $|R \widetilde{\cap}_\phi S|/\sqrt{(|r| \cdot |s|)}$ and $(2 \cdot |R \widetilde{\cap}_\phi S|)/(|r| + |s|)$, respectively [29].

Element Similarity. We consider both token-based and character-based similarity measures. For the former, each element is represented as a set of tokens, and we use Jaccard similarity:

$$\phi_{jac}(r, s) = \frac{|r \cap s|}{|r \cup s|} \tag{2}$$

For the latter, each element is represented as a string. We use the normalized edit similarity [30] to derive a score between 0 and 1:

$$\phi_{neds}(r, s) = 1 - \frac{ED(r, s)}{\max(|r|, |s|)} \tag{3}$$

where $ED(r, s)$ is the edit distance between r and s.

Problem Statement. We address the problem of fuzzy set similarity join, where the similarity score of two sets is based on the maximum weighted bipartite matching. To simplify the presentation, we focus on self-join, which involves a single collection $\mathcal{D}$. It is straightforward to adapt our algorithms to foreign join, which finds pairs of similar sets between two different collections $\mathcal{D}_1$ and $\mathcal{D}_2$. We mention such adaptations, where relevant. We consider two problem variants defined below. The threshold-based variant has been previously studied [6, 22, 23]. To the best of our knowledge, our work is the first one to also consider the top-k variant.

Problem 1 (Threshold-Based Fuzzy Set Similarity Join). *Given a collection $\mathcal{D}$ of sets and a similarity threshold $\delta \in [0, 1]$, find all pairs of sets (R, S) in $\mathcal{D}$ such that $sim_\phi(R, S) \geq \delta$.*

Problem 2 (Top-k Fuzzy Set Similarity Join). *Given a collection $\mathcal{D}$ of sets and an integer $k > 0$, find k pairs of sets (R, S) having the highest similarity score among all pairs of sets in $\mathcal{D}$.*

4 TOKEN UTILITIES AND BOUNDS

Our filtering approach is purely token-based. It avoids element comparisons, which are more expensive, without sacrificing pruning power. In this section, we define the concept of *token utility* and we show how it can be used to establish upper bounds on the similarity score between both elements and sets. For element similarity, we consider both Jaccard similarity and normalized edit similarity.

Token Utility. Recall that each element consists of a set of tokens, and the similarity between two elements $r \in R$ and $s \in S$ depends on the tokens they have in common. Based on this, we assign to each token $t \in r$ a utility score that measures the contribution of t to the similarity score of r with another element s. We then use this to derive an upper bound for the similarity score of a pair of elements (r, s), when a token t is contained in r but not in s. We further define the utility of a token t for the whole set R based on the utility of t in each element of R. Similarly, this is used to establish an upper bound on the similarity score of a pair of sets (R, S), by examining the utilities of their common tokens. Formally, we define the utility of a token as follows.

Definition 3 (Element-level Token Utility). *Assume a set R and an element $r \in R$. We define the utility of a token $t \in r$ as:*

$$u_t^r = \frac{1}{|r|} \tag{4}$$

Definition 4 (Set-level Token Utility). *We define the utility of a token t in a set R as the sum of its element-level utilities:*

$$u_t^R = \sum_{r \in R : t \in r} u_t^r \tag{5}$$

Bounds for Jaccard similarity. Lemma 1 shows that we can use the utility of tokens to establish an upper bound on the Jaccard similarity of two elements.

Lemma 1. *The Jaccard similarity of two elements r and s is at most equal to the sum of utilities of their common tokens:*

$$\phi_{jac}(r, s) \leq \sum_{t \in r \cap s} u_t^r \tag{6}$$

Proof. The proof can be easily derived based on the definition of Jaccard similarity (see Equation 2):

$$\phi_{jac}(r, s) = \frac{|r \cap s|}{|r \cup s|} \leq \frac{|r \cap s|}{|r|} = \sum_{t \in r \cap s} \frac{1}{|r|} = \sum_{t \in r \cap s} u_t^r$$

□

Lemma 2 shows how to derive an upper bound for the matching score $|R \widetilde{\cap}_{\phi_{jac}} S|$ between two sets R and S.

Lemma 2. *When using Jaccard as element similarity, the maximum weighted bipartite matching score between two sets R and S is at most equal to the sum of utilities of their common tokens:*

$$|R \widetilde{\cap}_{\phi_{jac}} S| \leq \sum_{t \in T_{RS}} u_t^R \tag{7}$$

where T_{RS} denotes the common tokens between R and S.

Proof. We assign each element $r \in R$ to its nearest neighbor in S, denoted by s_r, allowing multiple elements to be assigned to the same neighbor. According to Definition 1, this provides an upper bound for the maximum weighted matching score:

$$|R \widetilde{\cap}_\phi S| \leq \sum_{r \in R} \phi(r, s_r) \tag{8}$$

Note that this holds regardless of the element similarity being used. For $\phi = \phi_{jac}$, utilizing Lemma 1 and Definition 4, we get:

$$|R \widetilde{\cap}_{\phi_{jac}} S| \leq \sum_{r \in R} \sum_{t \in r \cap s_r} u_t^r \leq \sum_{t \in T_{RS}} u_t^R$$

□

Bounds for normalized edit similarity. The token utility is also defined as in Definition 3. However, each element r is now a string, and $|r|$ refers to the length of the string. When using a similarity function based on edit distance, the typical approach is to compare the q-chunks of the query element r with the q-grams of the candidate element s [10]. Both q-chunks and q-grams are sequences of q consecutive characters, with the difference that q-chunks are non-overlapping. The key property is that if a q-chunk of r does not match with any q-gram of s, the edit distance between r and s is at least 1. It is important to notice that the remaining $q-1$ characters contained in that q-chunk may still match with the respective $q-1$ characters of some q-gram of s. This implies that r and s may have edit similarity higher than 0 even if they do not contain any matching tokens. Based on the above, Lemma 3 shows how to use the utility of tokens to establish an upper bound on the similarity score of two elements in this case.

Lemma 3. *The normalized edit similarity between two elements r and s is at most equal to the sum of utilities of the q-chunks of r that match with a q-gram of s, with an additional offset of $1 - \frac{1}{q}$:*

$$\phi_{neds}(r,s) \leq 1 - \frac{1}{q} + \sum_{t \in r_{qc} \cap s_{qg}} u_t^r \tag{9}$$

where r_{qc} and s_{qg} denote the q-chunks of r and the q-grams of s, respectively.

Proof. Let c denote the number of q-chunks of r, out of a total of $\lceil \frac{|r|}{q} \rceil$ q-chunks, that match with q-grams of s. Among all such s strings, the most similar to r, say s^*, would have the same length with r, and would have the property that each mismatched q-chunk of r requires only 1 edit operation to match with one of its q-grams. That is: $\max(|r|, |s^*|) = |r|$ and $ED(r, s^*) = \lceil \frac{|r|}{q} \rceil - c$. Combining Equation 3 with the above, we get the following, for any pair (r, s) with c matching q-chunks:

$$\phi_{neds}(r,s) \leq \phi_{neds}(r,s^*) = 1 - \frac{\lceil \frac{|r|}{q} \rceil - c}{|r|} \leq 1 - \frac{\frac{|r|}{q} - c}{|r|}$$
$$= 1 - \frac{1}{q} + \frac{c}{|r|} = 1 - \frac{1}{q} + \sum_{t \in r_{qc} \cap s_{qg}} \frac{1}{|r|} = 1 - \frac{1}{q} + \sum_{t \in r_{qc} \cap s_{qg}} u_t^r$$

□

Notice that the upper bound in this case has an offset of $1 - \frac{1}{q}$. As explained earlier, this is due to the fact that two strings r and s may still be similar even if no q-chunks of r match with q-grams of s.

Lemma 4 shows how to derive an upper bound for the matching score $|R \widetilde{\cap}_{\phi_{neds}} S|$ between two sets R and S. Its proof is similar to that of Lemma 2, using Lemma 3 instead of 1.

Lemma 4. *When using normalized edit similarity as element similarity, the maximum weighted bipartite matching score between two sets R and S is at most equal to the sum of utilities of their common tokens with an additional offset of $1 - \frac{1}{q}$ per element:*

$$|R \widetilde{\cap}_{\phi_{neds}} S| \leq \sum_{t \in T_{RS}} u_t^R + \sum_{r \in R} 1 - \frac{1}{q} \tag{10}$$

where T_{RS} here denotes the intersection between the q-chunks of R and the q-grams of S.

5 THRESHOLD JOIN

We leverage the token utilities and bounds presented above to design a purely token-based filtering approach for threshold-based fuzzy set similarity join. First, we show how to translate a given threshold δ on the set similarity $sim_\phi(R, S)$ to an equivalent threshold θ on the maximum weighted bipartite matching score $|R \widetilde{\cap}_\phi S|$.

Matching threshold. This allows our algorithms to directly operate on bounds referring to the matching score, utilizing Lemmas 2 and 4. We derive two thresholds, namely θ_R, which depends only on R, i.e., it applies to any candidate S, and $\theta_{RS} \geq \theta_R$, which applies to a specific pair (R, S).

Lemma 5. *Given two sets R and S, and a threshold $\delta \in [0, 1]$, then:*

$$sim_\phi(R,S) \geq \delta \Rightarrow |R \widetilde{\cap}_\phi S| \geq \theta_{RS} \geq \theta_R$$

where $\theta_{RS} = \frac{\delta}{1+\delta}(|R| + |S|)$ *and* $\theta_R = \begin{cases} \delta \cdot |R| & \textit{for foreign-join} \\ \frac{2\cdot\delta}{1+\delta} \cdot |R| & \textit{for self-join.} \end{cases}$

Proof. Using Definition 2, we get:

$$sim_\phi(R,S) \geq \delta \Rightarrow \frac{|R \widetilde{\cap}_\phi S|}{|R| + |S| - |R \widetilde{\cap}_\phi S|} \geq \delta \Rightarrow$$
$$|R \widetilde{\cap}_\phi S| \geq \frac{\delta}{1+\delta}(|R| + |S|) = \theta_{RS}$$

Next, we notice that θ_{RS} depends on the size of both R and S. According to the size filter [1] (see Section 2), if a set s has Jaccard similarity with a set r at least δ, then its size must be within a certain range from the size of r, in particular $|s| \in [\delta|r|, |r|/\delta]$. It is easy to see that the same holds for fuzzy Jaccard, hence: $sim_\phi(R,S) \geq \delta \Rightarrow |S| \in [\delta|R|, |S|/\delta]$. Consequently, we can replace $|S|$ in θ_{RS} with the lower bound $\delta|R|$ to obtain the threshold $\theta_R = \delta|R|$, which only depends on the size of R. For self-join, it suffices to consider candidates with size $|S| \geq |R|$. Thus, we can substitute $|S|$ with $|R|$ to obtain $\theta_R = \frac{2\cdot\delta}{1+\delta}|R|$. □

Example 2. *Figure 2 introduces a running example that is used throughout Section 5 to better illustrate our method. The left table shows a query set R and four candidate sets S_1, S_2, S_3 and S_4, together with their similarity scores based on maximum weighted matching. Assume a similarity threshold $\delta = 0.7$, in which case only (R, S_4) is a match. The corresponding matching threshold for R is $\theta_R = \frac{2\cdot\delta}{1+\delta}|R| \approx 2.47$. Since $|S_1| = |S_2| = |S_3| = |S_4| = |R|$, $\theta_{RS} = \theta_R$ for each candidate.*

5.1 The TokenJoin Algorithm

We first provide an overview of our algorithm, called TokenJoin (TJ). Then, we explain candidate generation and refinement in more detail. To simplify the discussion, we assume Jaccard as element similarity. Adaptation to normalized edit similarity is straightforward, by substituting the respective token utilities and bounds.

Overview. The high-level process of TJ is outlined in Algorithm 1. To retrieve candidates for a given token, an inverted index $\mathcal{I}$ is constructed over the collection $\mathcal{D}$ (Line 2). This maps each token to an inverted list comprising the sets that contain it. The sets in each list are sorted in increasing order of their size. This allows us to directly apply the size filter while retrieving candidates from the index. For each set R, we generate and refine a set of candidates (Lines 3–5). Each candidate S is then verified. If its score is not lower than the threshold, it is added to the results (Lines 7–9).

Assume a query set R and a candidate set S. TJ iterates over the tokens of R to generate and refine candidates. For each token t, it examines whether t is contained in S, and accordingly uses Lemma 2 to progressively refine an upper bound on the matching score $|R \widetilde{\cap}_\phi S|$. The process terminates when either S is pruned or

Algorithm 1: TokenJoin (TJ)

```
Input: Collection of sets $\mathcal{D}$; Threshold $\delta$
Output: The set of matching pairs $\mathcal{M}$
1  $\mathcal{M} \leftarrow \emptyset$
2  $\mathcal{I} \leftarrow invertedIndex(\mathcal{D})$
3  foreach $R \in \mathcal{D}$ do
4      $C \leftarrow generateCandidates(R, \mathcal{I}, \delta)$
5      $C \leftarrow refineCandidates(R, C, \delta)$
6      foreach $S \in C$ do
7          $sim_\phi(R, S) \leftarrow verify(R, S)$
8          if $sim_\phi(R, S) \geq \delta$ then
9              $\mathcal{M} \leftarrow \mathcal{M} \cup \{(R, S)\}$
10 return $\mathcal{M}$
```

Algorithm 2: Candidate Generation

```
Input: Query set $R$; Inverted Index $\mathcal{I}$; Threshold $\delta$
Output: Initial set of candidates $C$
1  $C \leftarrow \emptyset$
2  $\theta_R \leftarrow \frac{2 \cdot \delta}{1+\delta} |R|$
3  $\sigma \leftarrow |R|$
4  foreach $t \in T_R$ do
5      for $S \in I[t]$ such that $|R| \leq |S| \leq |R|/\delta$ do
6          if $S \in C$ then $S.util \leftarrow S.util + u_t^R$
7          else
8              $S.util \leftarrow u_t^R$
9              $C \leftarrow C \cup \{S\}$
10     $\sigma \leftarrow \sigma - u_t^R$
11     if $\sigma < \theta_R$ then break
12 return $C$
```

all tokens of R are exhausted. During execution, let $T'_R \subseteq T_R$ denote the tokens of R that have been examined, while $T''_R = T_R \setminus T'_R$ the remaining ones. It is easy to see that $T_{RS} \subseteq (T'_R \cap T_S) \cup T''_R \subseteq T_R$, where T_{RS} denotes the common tokens of R and S. The key idea is that, at any point, the upper bound of $|R \widetilde{\cap}_\phi S|$ consists of the sum of utilities of: (i) the visited tokens that have been found to be contained in S, i.e., $T'_R \cap T_S$, and (ii) the remaining tokens in T''_R.

Tokens are visited in increasing order of their frequency in $\mathcal{D}$. This establishes a global ordering, while prioritizing less frequent tokens to reduce the number of generated candidates.[1] During candidate generation, visited tokens are used to generate new candidates but also to implicitly refine the upper bound of existing ones. During refinement, visited tokens are used to explicitly refine the upper bound of existing candidates. The algorithm switches from generation to refinement once the sum of token utilities in T''_R drops below θ_R. After this point no new candidates can accumulate sufficient utility to exceed the matching threshold.

Candidate Generation. The process is outlined in Algorithm 2. First, θ_R is initialized based on Lemma 5 (Line 2). We also use a variable σ, which holds the total utility of the remaining tokens to be examined, i.e., those in the set T''_R. Initially, $T''_R = T_R$, hence we initialize σ to $\sum_{t \in T_R} u_t^R = |R|$ (Line 3). For each examined token t,

[1] SilkMoth proposes an alternative heuristic, aiming to also reduce the number of signature tokens. However, it produces a different token ordering in each set, which is incompatible with our positional filter described in Section 5.2. Moreover, our experiments in Section 8.2 show that any benefit from it is negligible compared to our pre-refinement filter, which reduces the number of initial candidates by around 80%.

Algorithm 3: Candidate Refinement

```
Input: Query set $R$; Set of candidates $C$; Threshold $\delta$
Output: Refined set of candidates $C$
1  foreach $S \in C$ do
2      $\theta_{RS} = \frac{\delta}{1+\delta}(|R| + |S|)$
3      if $S.util + \sigma < \theta_{RS}$ then
4          $C \leftarrow C \setminus \{S\}$
5          continue
6      foreach $t \in T''_R$ do
7          $\sigma \leftarrow \sigma - u_t^R$
8          if $t \in T_S$ then $S.util \leftarrow S.util + u_t^R$
9          else if $S.util + \sigma < \theta_{RS}$ then
10             $C \leftarrow C \setminus \{S\}$
11             break
12 return $C$
```

we retrieve candidates from the inverted index, considering only those that satisfy the size filter (Line 5). If S is an existing candidate, we increment its utility score by u_t^R. Otherwise, we set its utility score to u_t^R and add it to the set of candidates (Lines 6–9). Then, we update σ by subtracting u_t^R. Once σ drops below θ_R, the candidate generation phase ends (Lines 10–11).

Example 3. *The right table in Figure 2 shows the tokens of R sorted in the order indicated by their subscripts. Below each token we show its utility in R and the total utility of the remaining tokens (the remaining lines will be explained later). We generate candidates by iterating over the tokens, until we reach t_5, where $\sigma = 23/12 < \theta_R \approx 2.47$. The collected utility score of each candidate S_1, S_2, S_3 and S_4 is $1/4$, $10/12$, $13/12$ and $13/12$, respectively.*

Candidate Refinement. The process is outlined in Algorithm 3. We now examine each candidate S individually, using the matching threshold θ_{RS} (Lines 1–2). Since candidates with the same size have the same θ_{RS}, in our implementation we precompute θ_{RS} for each $|S|$ and reuse it. For each S, T''_R initially contains the tokens that were not visited during generation, and σ holds their total utility.

Recall that, during generation, we have been maintaining a utility score $S.util$ for each candidate S. We now use this to prune candidates before actual refinement starts. If the sum of $S.util$ and σ, which is an upper bound for $|R \widetilde{\cap}_\phi S|$, is lower than θ_{RS}, S can be directly pruned (Lines 3–5). We call this the *Pre-Refinement Filter*, since no extra tokens have been examined yet. As shown in our experiments, this filter is both very lightweight and effective.

If S is not immediately pruned, we iterate over the remaining tokens in T''_R to progressively refine its upper bound (Lines 6–11). For each token t, we first update σ by subtracting u_t^R. If S contains t, we add u_t^R to its utility score. Otherwise, we check whether the upper bound has dropped below θ_{RS}. If so, S is pruned.

Example 4. *Continuing our example, since $S_1.util + \sigma = 1/4 + 23/12 < \theta_{RS_1}$, S_1 can be immediately pruned without refinement. For the remaining candidates, we iterate over the rest of the tokens of R. S_2 is pruned after examining t_9, since $S_2.util + \sigma = 25/12 + 4/12 < \theta_{RS_2}$. Since S_3 and S_4 contain all the tokens, they cannot be pruned.*

Comparison to SilkMoth. There are two notable differences between TokenJoin and SilkMoth. First, during candidate generation, SilkMoth only examines whether a given token t appears in

Set	Elements	$sim_\phi(R, S_i)$
R	$[[t_3, t_4, t_7, t_8], [t_5, t_6, t_7, t_8], [t_5, t_9, t_{10}]]$	
S_1	$[[t_3, t_7], [[t_4, t_8], [[t_4, t_7]$	0.132
S_2	$[[t_1, t_2, t_4, t_7], [t_1, t_2, t_5, t_8], [t_1, t_2, t_6]]$	0.125
S_3	$[[t_3, t_4, t_9, t_{10}], [t_5, t_6, t_9, t_{10}], [t_7, t_8]]$	0.358
S_4	$[[t_3, t_4, t_7], [t_5, t_6, t_8], [t_5, t_9, t_{10}]]$	0.714

(a) A query set R and four candidate sets $S_1, \ldots, S_4$.

	Generation			Refinement				
Tokens of R	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
Token Utility	1/4	1/4	7/12	1/4	1/2	1/2	1/3	1/3
Remaining Utility	33/12	30/12	23/12	20/12	14/12	8/12	4/12	0
Joint Utility wrt S_3	1/4	1/4	1/4	1/4	1/4	1/4	1/4	1/4
Pruning by TJ			S_1				S_2	
Pruning by TJP			S_1	S_2				
Pruning by TJPJ			S_1	S_2	S_3			

(b) Token utilities and pruned candidates.

Figure 2: Example showing token utilities and candidate pruning.

a candidate S. Instead, we also consider the utility of tokens, which allows us to introduce the pre-refinement filter. Second, to refine candidates, SilkMoth performs element comparisons. Instead, we completely rely on token utilities both to generate and to refine candidates, leveraging all tokens contained in R.

Example 5. *Applying SilkMoth to our example, S_1 and S_2 are pruned by the Nearest Neighbor Filter (NNF) and the Check Filter (CF), respectively. S_3 and S_4 survive both filters and need to be verified.*

5.2 Additional Refinement Filters

The filtering of TJ is extremely lightweight. For each token of R, it simply checks whether it is contained in S, it updates the upper bound accordingly, and compares it with θ_{RS}. Since the used token utilities only depend on R, they can be precomputed and reused across candidates. However, not considering any information from S is a limitation. For instance, if S contains all the tokens of R (like S_3 and S_4 in our example), there is no pruning opportunity.

To overcome this, we introduce two additional filters for candidate refinement. These are still based on token utilities, but also consider information related to S. This introduces some extra computational cost but increases pruning effectiveness. As opposed to TJ, where pruning may only occur when a token t of R is not found in S, these filters enable pruning even when t exists in S. The key observation is that $|R \tilde{\cap}_\phi S|$ is symmetric. Hence, for a common token t, we may use its utility in either R or S. Specifically, we can use the lower of the two, to obtain a tighter upper bound. We describe our two additional filters below.

Positional Filter. This type of filter has been proposed for traditional set similarity join as an extension to prefix filtering [28] (see Section 2). Assume that, under a global token ordering, the most recent common token t of R and S was found at position i in T_R and j in T_S. It is easy to see that there may exist at most $\min(|T_R| - i, |T_S| - j)$ new common tokens between R and S. In our setting, instead of only considering the number of remaining tokens, we consider the sum of their utilities. Let σ_R and σ_S denote the total utility of the remaining tokens in R and S, respectively. Recall that TJ prunes candidates based on the condition $S.util + \sigma < \theta_{RS}$, where $\sigma = \sigma_R$. To enable positional filtering, we set $\sigma = \min(\sigma_R, \sigma_S)$.

Using $\min(\sigma_R, \sigma_S)$, instead of just σ_R, may produce a tighter upper bound. The downside is that σ_S is not readily available. It has to be computed for each S whenever a token t is matched. To facilitate this, we make two adaptations. First, in the inverted index $\mathcal{I}$, each entry for a token t now stores not only the sets that contain t but also the position where t occurs in them. Second, for each set S, we precompute an array S_u of size $|T_S| - 1$ in which $S_u[i] = \sum_{j=i+1}^{|T_S|-1} u^S_{t_j}$, where t_j refers to the j-th token in T_S.

Example 6. *Recall that S_2 was pruned by* TJ *after examining token t_9. In* TJP, *refinement also starts from token t_6 but includes the positional filter. t_6 is contained in $T_{S_2} = \{ t_1, t_2, t_4, t_5, t_6, t_7, t_8 \}$, and the total remaining utility from the subsequent tokens t_7 and t_8 is 1/4 + 1/4 = 1/2. Since at that point $S_2.util$ = 10/12, the upper bound is 10/12 + 1/2 = 16/12 < θ_{RS_2}. Hence,* TJP *can prune S_2 after token t_6.*

Joint Utility Filter. So far, the utility of a token comes from a single set, either R or S. Our next filter introduces a new measure of token utility that applies to a pair of sets (R, S).

Recall from Definition 3 that u^R_t sums the utility of t over all the elements of R that contain t. However, there can be at most $l = \min(|R|, |S|)$ edges in the bipartite matching. Thus, we can restrict u^R_t to the sum of the top-l element-level utilities, denoted by $u^{R,l}_t$. Based on this idea, we define the joint utility as follows.

Definition 5 (Joint Token Utility). *The join utility u^{RS}_t of a token t in a pair of sets (R, S) is defined as:*

$$u^{RS}_t = \min(u^{R,l}_t, u^{S,l}_t) \tag{11}$$

where $u^{R,l}_t \leq u^R_t$ and $u^{S,l}_t \leq u^S_t$ denote the sum of the top-l element-level utilities of t in R and S, respectively, for $l = \min(|R|, |S|)$.

The joint utility filter replaces u^R_t in Lemmas 2 and 4 with the joint token utility $u^{RS}_t \leq u^R_t$, thus increasing pruning effectiveness.

Example 7. *Recall that S_3 was not pruned by either* TJ *or* TJP. TJPJ *performs a second pass over the tokens using the joint utility filter. The joint utility of each token of R with respect to S_3 is listed in Figure 2b. Since S_3 contains all tokens of R, its upper bound after the first pass is $S_3.util$ = 36/12. In the second pass, we update the utility of t_5 from 7/12 to 1/4, thus $S_3.util$ = 32/12. Then, we update the utility of t_7 from 1/2 to 1/4, thus $S_3.util$ = 29/12 < θ_{RS_3}, so S_3 is pruned.*

Enhanced Candidate Refinement Algorithm. We propose two extensions of TJ. The first one, called TJP, includes only the positional filter. After the pre-refinement filter, it iterates over the remaining tokens of R, as in TJ, but it also applies the positional

```
Algorithm 4: ENHANCED CANDIDATE REFINEMENT
Input: Query set $R$; Set of candidates $C$; Threshold $\delta$
Output: Refined set of candidates $C$
1  foreach $S \in C$ do
2      $\theta_{RS} = \frac{\delta}{1+\delta}(|R| + |S|)$
3      if $S.util + \sigma < \theta_{RS}$ then
4          $C \leftarrow C \setminus \{S\}$
5          continue
6      foreach $t \in T''_R$ do
7          $\sigma \leftarrow \sigma - u_t^R$
8          if $t \in T_S$ then
9              $S.util \leftarrow S.util + u_t^R$
10             $i \leftarrow$ position of $t$ in $T_S$
11             if $S.util + S_u[i] < \theta_{RS}$ then
12                 $C \leftarrow C \setminus \{S\}$
13                 continue (Line 1)
14         else if $S.util + \sigma < \theta_{RS}$ then
15             $C \leftarrow C \setminus \{S\}$
16             continue (Line 1)
17     foreach $t \in T_{RS} : u_t^{RS} < u_t^R$ do
18         $S.util \leftarrow S.util - u_t^R + u_t^{RS}$
19         if $S.util < \theta_{RS}$ then
20             $C \leftarrow C \setminus \{S\}$
21             break
22 return $C$
```

filter whenever a token is matched. The second variant, called TJPJ, also adds the joint utility filter. It works like TJP, but for each surviving candidate S it makes a second pass over the tokens of R. During it, for each matched token t, it uses its joint utility u_t^{RS} instead of u_t^R to obtain a possibly tighter upper bound on $|R \widetilde{\cap}_\phi S|$.

The enhanced refinement including all filters is outlined in Algorithm 4. It involves three main parts. The first part (Lines 2–5) corresponds to the pre-refinement filter, and is the same as in TJ. The second part (Lines 6–16) is similar to TJ but extended to also apply the positional filter. The difference is that, for each matched token t, it now also tracks the position of t in T_S, and uses $S_u[i]$ to check for pruning (Lines 10–13). The third part (Lines 17–21) introduces the joint utility filter. This revisits the common tokens between R and S. For each such token t, it computes the joint utility u_t^{RS}. If $u_t^{RS} < u_t^R$, it tightens the upper bound $S.util$ by subtracting u_t^R and adding u_t^{RS}. If $S.util$ now drops below θ_{RS}, S is pruned.

6 TOP-K JOIN

Next, we address the top-k variant of the problem (see Problem 2), where the user specifies the number of results to be returned instead of a similarity threshold.

6.1 Baseline Algorithms

Since previous works have not addressed the top-k problem, we start by designing two baseline algorithms, using two different starting points. First, we extend SILKMOTH to the top-k setting. Second, we adapt a top-k algorithm for traditional set similarity join to the fuzzy setting. We refer to these algorithms as Top-k SilkMoth (SMK) and Top-k Fuzzy Join (FJK), respectively.

Top-k SilkMoth (SMK). Recall that, for each set R, SILKMOTH first selects a subset of its tokens as a signature to generate candidates. To construct the signature, a similarity threshold δ is required. Then, the generated candidates are pruned using the refinement filters CF and NNF, which are applied sequentially. The important observation is that each of these filters computes a progressively tighter upper bound on the matching score $|R \widetilde{\cap}_\phi S|$.

To adapt SILKMOTH to the top-k problem, we apply two main modifications. First, since no user-specified threshold is available, we introduce a threshold initialization process to enable signature construction. Specifically, we reuse the same one that we design for our method, which is described in Section 6.2. Second, instead of examining candidates sequentially, we insert them in a priority queue Q in decreasing order of their current upper bound. For each candidate S pulled from Q, instead of applying all refinement filters, we only apply the next filter in the sequence. Then, we insert S back to Q based on its updated upper bound. If all filters have already been applied, S is verified. In this way, candidate refinement is prioritized based on the upper bounds. More promising candidates will be verified earlier, thus being more likely to increase the current threshold, which is equal to the k-th best pair found so far.

Top-k Fuzzy Join (FJK). To derive an alternative baseline, we adapt the top-k set similarity join algorithm presented in [27]. This algorithm, instead of processing each set R sequentially, it inserts all sets in a priority queue Q, and processes them one token at a time. For each set R pulled from Q, it visits its next token t, and retrieves candidates that contain t. Each new candidate, if not pruned, is verified, and if its score is higher than the current top-k result, it is inserted in the matches. If R has remaining tokens, its upper bound for future candidates is updated, and it is pushed back to Q.

To adapt this to our setting, we follow the same overall process but we replace the filtering criteria and the verification function with the ones for our problem, i.e., the token-based filters described in Section 5, and the verification based on maximum weighted bipartite matching. It is worth noting, however, that this method performs many verifications between candidate pairs. Since, in our case, verifications involve maximum weighted matching computations instead of set overlap, this considerably increases the computational cost.

6.2 Proposed Algorithm

Our proposed algorithm, called TJK, is outlined in Algorithm 5. It uses the same filtering techniques as for threshold-based join. However, to address the top-k problem, TJK also employs candidate prioritization and threshold initialization, as explained below.

Candidate Prioritization. For each set R, candidates are generated in the same way as in TJ (Line 5). For each candidate S that passes the pre-refinement filter, we initialize its status to 0, and we push S to a priority queue Q based on its upper bound (Lines 6–11). Whenever a candidate S is pulled from Q, we recompute θ_{RS}, since δ might have increased in the meantime, and we check for pruning (Lines 13–15). If S is not pruned, and its status is 0 or 1, we apply the next filter, which is the positional or the joint utility filter, respectively (Lines 16–17). If S is still not pruned, we increment its status by 1, and push it back to Q (Lines 18–20). If the status of S was 2, meaning that both filters have already been applied, then S

Algorithm 5: TokenJoinTopK (TJK)

```
Input: Collection of sets $\mathcal{D}$; Number of results $k$
Output: Fixed size sorted list $\mathcal{M}$ containing the top-$k$ matches
$\mathcal{I} \leftarrow invertedIndex(\mathcal{D})$
$\mathcal{M}, \delta \leftarrow initialize()$
foreach $R \in \mathcal{D}$ do
    $Q \leftarrow \emptyset$
    $C \leftarrow generateCandidates(R, \mathcal{I}, \delta)$
    foreach $S \in C$ do
        $\theta_{RS} \leftarrow \frac{\delta}{1+\delta}(|R| + |S|)$
        $S.ub \leftarrow S.util + \sigma$
        if $S.ub \geq \theta_{RS}$ then
            $S.status \leftarrow 0$
            $Q.push(S, S.ub, S.status)$
    while $\neg Q.isEmpty()$ do
        $S \leftarrow Q.pop()$
        $\theta_{RS} \leftarrow \frac{\delta}{1+\delta}(|R| + |S|)$
        if $S.ub < \theta_R$ then continue
        if $S.status < 2$ then
            $S.ub \leftarrow applyNextFilter()$
            if $S.ub \geq \theta_{RS}$ then
                $S.status \leftarrow S.status + 1$
                $Q.push(S, S.ub, S.status)$
        else
            $sim_\phi(R, S) \leftarrow verify(R, S)$
            if $sim_\phi(R, S) > \delta$ then
                $\mathcal{M} \leftarrow \mathcal{M} \cup \{(R, S)\}$
                $\delta \leftarrow \mathcal{M}_k.score$
return $\mathcal{M}$
```

is verified (Line 22). If its similarity score is greater than δ, we add S to the current top-k results and update δ (Lines 23–25).

Threshold Initialization. To bootstrap the process, an initial threshold δ is required (Line 2). It is important that the threshold initialization process is both lightweight and effective. It should provide an initial threshold that is close to the actual one, while not imposing a significant overhead to the entire algorithm. To achieve this, we try to quickly generate a relatively small pool of promising candidates, which will be verified to produce an initial list of top-k matches. Hence, we follow the same candidate prioritization process but with several restrictions: (i) we only iterate over the first $\rho \cdot |\mathcal{D}|$ sets in ascending order of their size (so that performed verifications are less costly); (ii) we set a default threshold δ_g to generate candidates from each R; (iii) we pick only the top-μ of those candidates; (iv) we select the top $\lambda \cdot k$ candidates from all sets, which we then refine and verify to produce an initial list of top-k results. The parameters ρ, δ_g, μ and λ control the trade-off between the speed and effectiveness of the threshold initialization process. In our experiments (see Section 8.3), we have set $\rho = 0.4$, $\delta_g = 0.9$, $\mu = 0.01 \cdot k$ and $\lambda = 2$; these values are intuitive and have provided overall good results across all tested datasets.

7 EFFICIENT VERIFICATION

Even with highly effective filters, many candidate pairs may still remain to be verified. Thus, verification may still constitute a bottleneck, due to the high cost of maximum weighted bipartite matching.

Table 1: Datasets used in the experiments.

Dataset	Num Sets	Elements per Set	Tokens per Element	Element Similarity
Yelp	160,016	6.37	5.95	JAC
GDELT	500,000	26.20	19.38	JAC
Enron	517,431	133.57	4.64	JAC
Flickr	500,000	8.04	9.17	NEDS
DBLP	500,000	13.01	5.54	NEDS
MIND	123,130	32.49	4.26	NEDS

To accelerate this process, we observe that it may be possible to terminate the verification of a pair (R, S) early, without fully computing its maximum weighted matching. The Kuhn-Munkres algorithm [11, 14], which is used for this purpose, computes the matching in steps, starting from an empty one, $G_M^0 = \emptyset$. At each step i, it increases the size of the current matching G_M^i by adding new vertices that were unmatched, until the matching is complete. To obtain an upper bound s_{max}^i, we assign each unmatched element $r \in R$ to each nearest neighbor in S, allowing multiple elements to be assigned to the same neighbor. A lower bound s_{min}^i is also obtained by greedily assigning unmatched elements of R to unmatched elements of S. If at any step $s_{max} < \theta_{RS}$, (R, S) can be pruned. Also, if $s_{min} \geq \theta_{RS}$, (R, S) can be directly added to the results (assuming that the exact similarity score is not required). This allows us to stop the verification at any intermediate step.

8 EVALUATION

Next, we present our experimental evaluation. We compare the execution time of our algorithms against the respective baselines for threshold and top-k join. We also investigate the pruning effectiveness of our filters.

8.1 Experimental Setup

Datasets. We have used six real-world datasets:

- Yelp: 160,016 sets extracted from the Yelp Open Dataset. Each set refers to a business. Its elements are the categories associated to it.
- GDELT: 500,000 randomly selected sets from January 2019 extracted from the GDELT Project. Each set refers to a news article. Its elements are the themes associated with it. Themes are hierarchical. Each theme is represented by a string concatenating all themes from it to the root of the hierarchy.
- Enron: 517,431 sets, each corresponding to an email message. The elements are the words contained in the message body.
- Flickr: 500,000 randomly selected images from the Flickr Creative Commons dataset. Each set corresponds to a photo. The elements are the tags associated to that photo.
- DBLP: 500,000 publications from the DBLP computer science bibliography. Each set refers to a publication. The elements are author names and words in the title.
- MIND: 123,130 articles from the MIcrosoft News Dataset. Each set corresponds to an article. The elements are the words in its abstract.

Following common practice, we derive tokens by splitting each word into q-grams (q=3) and replacing each q-gram with an integer identifier. The characteristics of the datasets are summarized in

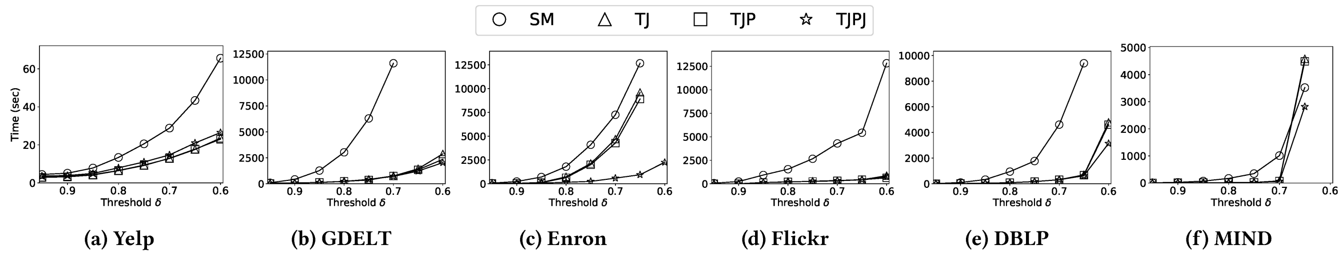


Figure 3: Execution time for varying threshold δ (dataset size $|\mathcal{D}|$ set to 40%).

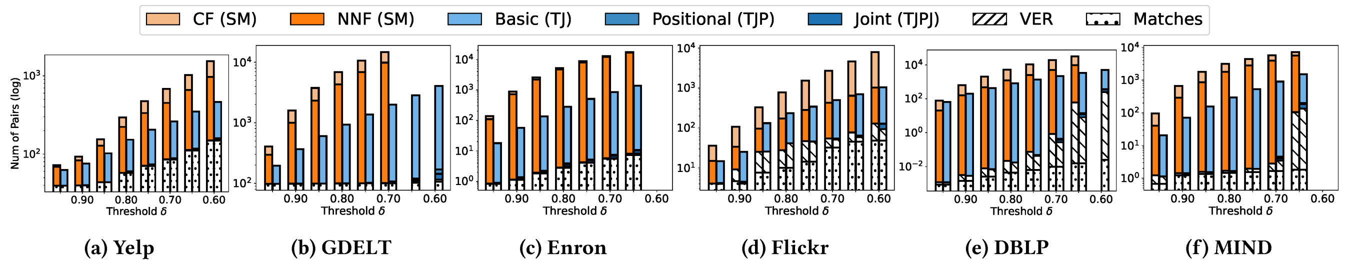


Figure 4: Pruning effectiveness of each filter for varying threshold δ (dataset size $|\mathcal{D}|$ set to 40%).

Table 1, which also indicates the element similarity applied in each case.

Algorithms. For threshold join, we compare four algorithms:

- SilkMoth [6], which is the current state of the art;
- TJ, which is our basic algorithm, using token utilities that are only relative to R;
- TJP, which additionally uses the positional filter; and
- TJPJ, which employs both the positional and the joint filter.

For top-k join, we compare three algorithms:

- SMK, which is a baseline derived by extending SilkMoth to the top-k setting;
- FJK, which is a baseline derived by adapting a top-k algorithm for traditional set similarity join to the fuzzy setting;
- TJK, which is our proposed, token-based algorithm.

All algorithms were implemented in Java, including SilkMoth which was also implemented by us. The experiments were executed on a server with AMD Ryzen Threadripper 3960X 24-Core processor and 256 GB RAM running Ubuntu 20.04, using a single thread and heap size of 70 GB. Our code and all datasets are available on GitHub.[2]

8.2 Results on Threshold Join

We first compare the execution time of TJ, TJP and TJPJ against SilkMoth, while varying the similarity threshold δ. The results are shown in Figure 3. Runs that did not complete in 5 hours were interrupted and are omitted from the plot.

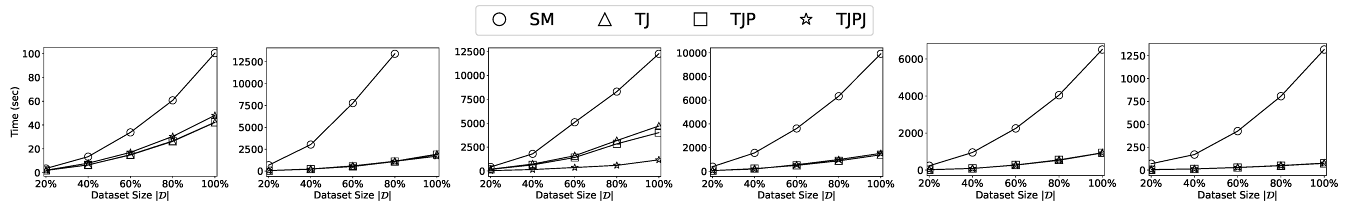
As we can see, TJPJ clearly outperforms SilkMoth in all cases, being an order of magnitude faster on average. In Yelp, which contains fewer and smaller sets as shown in Table 1, both methods exhibit low execution times, and their differences are smaller, with TJPJ being about 3 times faster for $\delta = 0.6$. In the other datasets, which are more challenging, TJPJ is up to 37 times faster than SilkMoth. The difference between the two methods becomes increasingly higher as the similarity threshold decreases, which means that many more candidates are generated and need to be processed. This shows that our proposed token-based filters are significantly more lightweight without sacrificing pruning effectiveness.

Regarding TJ and TJP, their execution time is usually close to that of TJPJ. This shows that the use of token utilities to filter candidates is very effective. Even the simpler filters employed by TJ and TJP are sufficient in most cases to achieve good performance. However, this does not hold in Enron and MIND, which contain large sets, and therefore verification can be very costly. In these cases, the lower pruning power of TJ and TJP has a high impact, making them much slower than TJPJ in Enron, and even slower than SilkMoth for $\delta = 0.65$ in MIND.

To study in more detail the impact of the various filters, we measure their pruning effectiveness, in terms of the number of pairs in the input and output of each stage. The results are shown in Figure 4. In these plots, each left bar corresponds to SilkMoth, which includes two filters (CF and NNF), while each right bar refers to our method, which involves three filters (Basic, Positional, and Joint). Each bar also includes the number of pairs that are verified (VER). In each bar segment, the upper level indicates the input number of pairs for that stage, while the bottom level indicates the output. We also show the number of final matches, which is the same for all algorithms. The log scale in the y-axis should be noted.

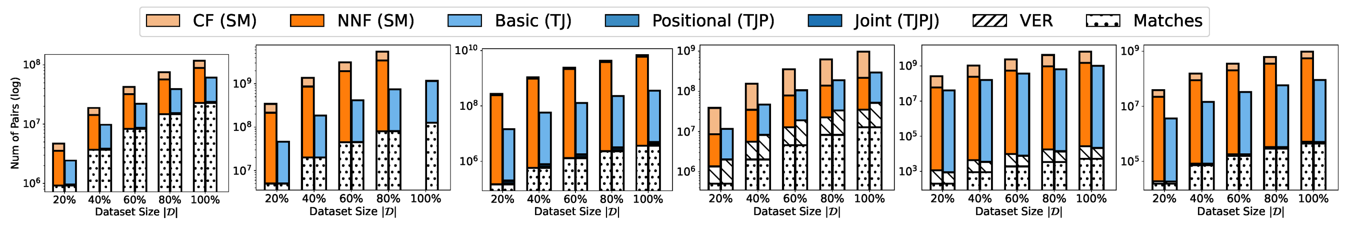
A first and important observation is that our refinement filters need to deal with a significantly lower number of candidates than those of SilkMoth. This can be seen by the clear difference between the total height of the right bars compared to the left ones. In SilkMoth, every candidate S that contains at least one signature

[2]https://github.com/alexZeakis/TokenJoin



(a) Yelp **(b) GDELT** **(c) Enron** **(d) Flickr** **(e) DBLP** **(f) MIND**

Figure 5: Execution time for varying dataset size $|\mathcal{D}|$ ($\delta = 0.80$).



(a) Yelp **(b) GDELT** **(c) Enron** **(d) Flickr** **(e) DBLP** **(f) MIND**

Figure 6: Pruning effectiveness of each filter for varying dataset size $|\mathcal{D}|$ ($\delta = 0.80$).

token of R has to go through refinement. Instead, in our methods, during candidate generation we also track the accumulated token utility of each candidate S. We then use it to immediately prune candidates at the end of this stage, before actual refinement starts. This shows the importance of the pre-refinement filter (described in Section 5.1). It is characteristic that, in most cases, SilkMoth needs to enter the stage of NNF in order to reach the same level of candidates that is the starting point for our refinement.

A second important observation is that the basic filter of `TJ` is able to prune roughly the same number of pairs as NNF, or even higher in some cases, such as in DBLP. The number of candidates that can be pruned by the positional or the joint filter, but not the basic, is relatively small, and thus not very noticeable in the plot in most cases. This justifies why the execution time of `TJ`, `TJP` and `TJPJ` does not exhibit large differences, as discussed previously. Usually, the overhead of the extra filters is comparable to the cost savings obtained from the additionally pruned candidates.

Yet, some differences among the filters can be observed for lower thresholds, e.g., for $\delta = 0.6$ in GDELT and $\delta = 0.65$ in MIND, which justify the respective differences in the execution time shown in Figure 3. Especially in the case of Enron, which contains large sets, even these small differences have a huge impact on the execution time, as shown in Figure 3.

In addition, Figures 5 and 6 show the execution time of the algorithms and the pruning effectiveness of the filters, respectively, when varying the dataset size $\mathcal{D}$. The main observations are consistent with the ones discussed above. Our token-based filtering algorithms `TJ`, `TJP` and `TJPJ` outperform SilkMoth in all cases, especially as the dataset size increases, which means that more candidate pairs exist. In GDELT, SilkMoth fails to complete the execution within the specified time limit of 5 hours when using the entire dataset. Instead, our methods successfully return results in all cases, exhibiting much better scalability.

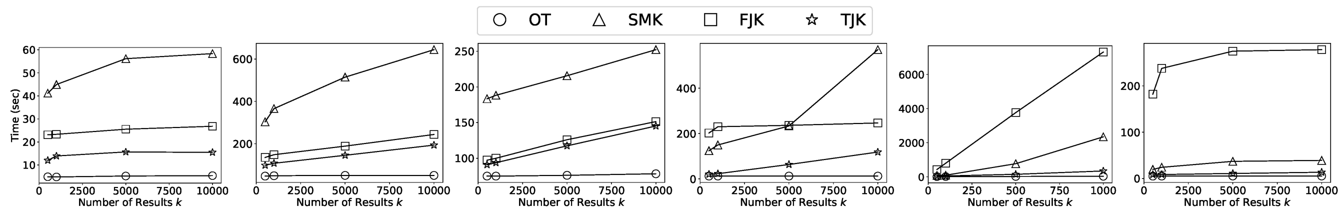
As before, this can be explained by comparing the pruning effectiveness of the filters. Due to the pre-refinement filter, our methods need to refine a much lower number of candidates compared to SilkMoth. Moreover, even with the basic filter of `TJ` we are able to reach comparable pruning capacity to the considerably more expensive NNF filter of SilkMoth.

Summary. Overall, our results can be summarized as follows:

- *Execution time.* `TJPJ` clearly outperforms SilkMoth in all the experiments, offering an average and a maximum speedup of around 9× and 37×, respectively. The execution time of `TJ` and `TJP` is close to that of `TJPJ` in most cases, with `TJPJ` offering an average speedup of around 1.65× and 1.57×, respectively. However, `TJPJ` exhibits higher efficiency and scalability when dealing with larger sets and lower thresholds, reaching a speedup of 10× in such cases.
- *Pruning effectiveness. Our pre-refinement filter reduces the number of candidates to be refined by around 80% on average compared to SilkMoth. In addition, our refinement filters offer comparable (and in some cases even higher) pruning power to those of SilkMoth, despite being significantly more lightweight. The extra filters employed by `TJPJ` offer significant benefits when dealing with large sets where verification cost is very high.*

8.3 Results on Top-k Join

Next, we evaluate the performance of our method for the top-k variant of fuzzy set similarity join. We compare the execution time of our algorithm `TJK` against the two baselines `SMK` and `FJK`. For all methods, we use the same threshold initialization process described in Section 6.2. For reference, we additionally include a method that assumes that the true threshold δ, i.e., the similarity score of the k-th best pair, is known a priori. This method, denoted by `Oracle`



(a) Yelp **(b) GDELT** **(c) Enron** **(d) Flickr** **(e) DBLP** **(f) MIND**

Figure 7: Execution time for varying k (dataset size $|\mathcal{D}|$ set to 100%).

Table 2: Average verification time (nanoseconds) per verified pair grouped by size.

Set Size	Ver-FULL	Ver-UB	Ver-ULB
Small	3	3	3
Medium	90	89	88
Large	114	110	25

Threshold (OT), applies the TJPJ algorithm to perform a threshold join using δ.

In this experiment, we set the dataset size to 100%. We vary the number of results k in the range [500, 1000, 5000, 10000], except for DBLP, where we vary k in [50, 100, 500, 1000]. The reason is that, as can be seen in Figure 6, there are significantly fewer matches (more than an order of magnitude difference) at the same similarity threshold in DBLP compared to the rest of the datasets. Moreover, we skip exact duplicates to avoid trivial matches.

The results are shown in Figure 7. TJK is the fastest algorithm, offering an average speedup of around 6× compared to its competitors, which reaches up to 25× in several cases, especially as k increases. Moreover, in DBLP and MIND, TJK is very close to OT, which shows that both threshold initialization and candidate prioritization are quite effective. Nevertheless, although SMK and FJK employ the same threshold initialization process, their execution time is higher. This indicates that candidate prioritization and filtering in these methods is less efficient. FJK performs better than SMK in Yelp, GDELT, Enron and Flickr, while SMK is faster in DBLP and MIND.

Like in the case of threshold join, the lower execution time of TJK compared to SMK is attributed to the performance benefits of token-based filtering compared to element-based filtering. FJK benefits from token filtering but its candidate prioritization is more naive compared to TJK, as it performs verifications more eagerly. This increases its execution time, especially in Flickr, DBLP and MIND, where the element similarity is based on edit distance, which is more expensive than Jaccard similarity.

8.4 Efficient Verification

In the results presented so far, we have been using the standard verification process for all methods, which fully computes the maximum weighted bipartite matching for each verified pair (R, S). In our final experiment, we examine the effect of applying early termination in the verification stage, as described in Section 7. Recall that this is based on maintaining an upper and lower bound for each pair during verification. These bounds allow us to more quickly identify false positives and true positives, respectively, without calculating the complete matching. For true positives, the assumptions is that only the matching pairs, and not their exact similarity score, are needed.

In this experiment, we measure the average verification time for all pairs that are verified by TJPJ across all datasets, for 40% of the entire dataset size and $\delta = 0.8$. Recall that the verification cost highly depends on the size of the verified sets. Thus, we group the verified pairs in three categories based on their size: (i) *small* – up to 10 elements per set; (ii) *medium* – up to 100 elements per set; and (iii) *large* – more than 100 elements per set.

We compare the following methods: (i) Ver-FULL: the standard verification, without early termination; (ii) Ver-UB: use of upper bound to accelerate verification for false positives; (iii) Ver-ULB: use of both upper and lower bounds to early identify both false and true positives. The results are shown in Table 2. Although early termination has a small impact for small and medium sets, its effect increases when dealing with large sets. For the latter, the use of upper bound offers a speedup of around 1.03×, while additionally using the lower bound achieves a speedup of around 4.5×.

9 CONCLUSIONS

In this work, we have addressed the set similarity join problem based on maximum weighted bipartite matching. In contrast to the state-of-the-art algorithm, which relies on element-based filtering, we have proposed a token-based filtering approach. We define the concept of token utility, and use it throughout the entire process, both generation and refinement, to filter candidates. This allows us to design filters that are significantly more lightweight and effective. We have also introduced a top-k algorithm, and we have accelerated verification through early termination.

Our experiments with six real-world datasets show that our proposed algorithm outperforms the state of the art by a wide margin in all cases, being an order of magnitude faster on average. In the future, we intend to investigate the design of parallel and distributed algorithms, as well as approximate ones, to further improve efficiency and scalability. We also plan to support elements with different weights.

ACKNOWLEDGMENTS

This work was partially funded by the EU projects SmartDataLake (H2020 – Grant No. 825041) and STELAR (Horizon Europe – Grant No. 101070122).

REFERENCES

[1] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. 2006. Efficient Exact Set-Similarity Joins. In *VLDB*. 918–929.

[2] Roberto J Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling up all pairs similarity search. In *WWW*. 131–140.

[3] Panagiotis Bouros, Shen Ge, and Nikos Mamoulis. 2012. Spatio-textual similarity joins. *Proceedings of the VLDB Endowment* 6, 1 (2012), 1–12.

[4] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A Primitive Operator for Similarity Joins in Data Cleaning. In *ICDE*. 5.

[5] Tobias Christiani, Rasmus Pagh, and Johan Sivertsen. 2018. Scalable and Robust Set Similarity Join. In *ICDE*. 1240–1243.

[6] Dong Deng, Albert Kim, Samuel Madden, and Michael Stonebraker. 2017. Silk-Moth: An Efficient Method for Finding Related Sets with Maximum Matching Constraints. *PVLDB* 10, 10 (2017), 1082–1093.

[7] Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser, and Johann-Christoph Freytag. 2018. Set Similarity Joins on MapReduce: An Experimental Survey. *PVLDB* 11, 10 (2018), 1110–1122.

[8] Zvi Galil. 1986. Efficient Algorithms for Finding Maximum Matching in Graphs. *ACM Comput. Surv.* 18, 1 (1986), 23–38.

[9] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. 2001. Approximate String Joins in a Database (Almost) for Free. In *VLDB*. 491–500.

[10] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. 2014. String Similarity Joins: An Experimental Evaluation. *PVLDB* 7, 8 (2014), 625–636.

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

[12] Wei Lu, Xiaoyong Du, Marios Hadjieleftheriou, and Beng Chin Ooi. 2014. Efficiently Supporting Edit Distance Based String Similarity Search Using B^+-Trees. *IEEE Trans. Knowl. Data Eng.* 26, 12 (2014), 2983–2996.

[13] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. 2016. An empirical evaluation of set similarity join techniques. *PVLDB* 9, 9 (2016), 636–647.

[14] James Munkres. 1957. Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics* 5, 1 (1957), 32–38.

[15] George Papadakis, Dimitrios Skoutas, Emmanouil Thanos, and Themis Palpanas. 2020. Blocking and Filtering Techniques for Entity Resolution: A Survey. *ACM Comput. Surv.* 53, 2 (2020), 31:1–31:42.

[16] Jianbin Qin, Wei Wang, Yifei Lu, Chuan Xiao, and Xuemin Lin. 2011. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *SIGMOD*. 1033–1044.

[17] Leonardo Andrade Ribeiro and Theo Härder. 2011. Generalizing prefix filtering to improve set similarity joins. *Information Systems* 36, 1 (2011), 62–78.

[18] Sunita Sarawagi and Alok Kirpal. 2004. Efficient set joins on similarity predicates. In *SIGMOD*. 743–754.

[19] Venu Satuluri and Srinivasan Parthasarathy. 2012. Bayesian Locality Sensitive Hashing for Fast Similarity Search. *Proc. VLDB Endow.* 5, 5 (2012), 430–441.

[20] Sebastian Wandelt, Dong Deng, Stefan Gerdjikov, Shashwat Mishra, Petar Mitankin, Manish Patil, Enrico Siragusa, Alexander Tiskin, Wei Wang, Jiaying Wang, and Ulf Leser. 2014. State-of-the-art in string similarity search and join. *SIGMOD Rec.* 43, 1 (2014), 64–76.

[21] Jiannan Wang, Guoliang Li, and Jianhua Feng. 2012. Can we beat the prefix filtering? An adaptive framework for similarity join and search. In *SIGMOD*. 85–96.

[22] Jiannan Wang, Guoliang Li, and Jianhua Feng. 2014. Extending string similarity join to tolerant fuzzy token matching. *TODS* 39, 1 (2014), 7:1–7:45.

[23] Jin Wang, Chunbin Lin, and Carlo Zaniolo. 2019. MF-Join: Efficient Fuzzy String Similarity Join with Multi-level Filtering. In *ICDE*. 386–397.

[24] Wei Wang, Jianbin Qin, Chuan Xiao, Xuemin Lin, and Heng Tao Shen. 2013. VChunkJoin: An Efficient Algorithm for Edit Similarity Joins. *IEEE Trans. Knowl. Data Eng.* 25, 8 (2013), 1916–1929.

[25] Xubo Wang, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. 2017. Leveraging Set Relations in Exact Set Similarity Join. *Proc. VLDB Endow.* 10, 9 (2017), 925–936.

[26] Chuan Xiao, Wei Wang, and Xuemin Lin. 2008. Ed-Join: an efficient algorithm for similarity joins with edit distance constraints. *Proc. VLDB Endow.* 1, 1 (2008), 933–944.

[27] Chuan Xiao, Wei Wang, Xuemin Lin, and Haichuan Shang. 2009. Top-k Set Similarity Joins. In *ICDE*. 916–927.

[28] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. 2011. Efficient similarity joins for near-duplicate detection. *TODS* 36, 3 (2011), 1–41.

[29] Minghe Yu, Guoliang Li, Dong Deng, and Jianhua Feng. 2016. String similarity search and join: a survey. *Frontiers Comput. Sci.* 10, 3 (2016), 399–417.

[30] Li Yujian and Liu Bo. 2007. A normalized Levenshtein distance metric. *IEEE Trans. Pattern Anal. Mach. Intell.* 29, 6 (2007), 1091–1095.

[31] Jiaqi Zhai, Yin Lou, and Johannes Gehrke. 2011. ATLAS: a probabilistic algorithm for high dimensional similarity search. In *SIGMOD*. 997–1008.

[32] Yong Zhang, Xiuxing Li, Jin Wang, Ying Zhang, Chunxiao Xing, and Xiaojie Yuan. 2017. An Efficient Framework for Exact Set Similarity Search Using Tree Structure Indexes. In *ICDE*. 759–770.

[33] Zhenjie Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, and Divesh Srivastava. 2010. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *SIGMOD*. 915–926.

[34] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In *SIGMOD*. 847–864.