Front. Comput. Sci., 2016, 10(3): 399–417
DOI 10.1007/s11704-015-5900-5

REVIEW ARTICLE

String similarity search and join: a survey

Minghe YU, Guoliang LI (✉), Dong DENG, Jianhua FENG

Department of Computer Science, Tsinghua University, Beijing 100084, China

Abstract String similarity search and join are two important operations in data cleaning and integration, which extend traditional exact search and exact join operations in databases by tolerating the errors and inconsistencies in the data. They have many real-world applications, such as spell checking, duplicate detection, entity resolution, and webpage clustering. Although these two problems have been extensively studied in the recent decade, there is no thorough survey. In this paper, we present a comprehensive survey on string similarity search and join. We first give the problem definitions and introduce widely-used similarity functions to quantify the similarity. We then present an extensive set of algorithms for string similarity search and join. We also discuss their variants, including approximate entity extraction, type-ahead search, and approximate substring matching. Finally, we provide some open datasets and summarize some research challenges and open problems.

Keywords string similarity, similarity search, similarity join, top-*k*

1 Introduction

Selection and join are two important operations in traditional databases to help users query the data. The underlying data, however, are rather dirty in real world, due to the typographical errors and data inconsistencies (e.g., the same entity has different representations). The exact selection and join operations cannot return answers if the data are not exactly matched. To address this problem, approximate selection and join (aka similarity search and join) are proposed to extend exact selection and exact join by tolerating errors and inconsistencies, and they have many real-world applications, including data cleaning, data integration and data clustering.

Given a set of objects and a query object, similarity search aims to find similar objects to the query. String similarity search has many real-world applications, such as data cleaning and spell checking. For example, in spell checking, given a set of words and a query word, string similarity search wants to find similar words to the query [1–9]. Many information systems utilize this feature to enhance the usability and provide user-friendly functionalities, e.g., Microsoft Word, Gmail, and Google search engine.

Given two sets of objects, similarity join aims to find all similar pairs from the two collections. String similarity join also has many real-world applications, such as data integration, entity resolution, near duplicate detection, and clustering [10–21]. For example, assume we want to build an academic publication search system, there are many database sources on publications, such as DBLP, DBLife, and CiteSeer. The entity may have inconsistencies (typographical errors or difference representations) between records in different sources. For instance, "entity resolution", "entity matching", "entity alignment", "deduplication", and "record linkage" should refer to the same entity, although they have different representations. To tolerate the inconsistencies, we need to utilize the similarity join operations to integrate the data from multiple data sources.

There are several challenges in string similarity join and search. The first is how to quantify the similarity between two objects. Similarity functions are proposed to evaluate the similarity between the data. There are three main types of

Received August 7, 2015; accepted October 9, 2015

E-mail: {yumh12,dd11}@mails.thu.edu.cn;
{liguoliang,fengjh}@tsinghua.edu.cn

similarity functions: 1) character-based similarity functions, 2) token-based similarity functions, and 3) hybrid similarity functions. The first evaluates the similarity by transforming an object to another based on character transformations. The second tokenizes each object as a set and utilizes the set similarity to quantify the similarity. The third hybrids the first and the second, which also models an object as a set of tokens and allows fuzzy matching between the tokens. We will provide a comprehensive survey on various similarity functions.

The second is how to efficiently support similarity join and search, and achieve high performance and scalability. A well-known method is to employ a filter-and-verification strategy which includes two steps. The filter step utilizes a lightweight filtering technique to prune larger numbers of dissimilar pairs with a little cost, and the verification step verifies the candidate pairs that are not pruned by the filter step. Most of existing algorithms focus on the filter step and the optimization goal is to make a tradeoff between filtering power and filtering cost, where the filtering power refers to the percentage of pruned pairs and filtering cost denotes the cost of use the filter to prune dissimilar pairs. Obviously, the larger the pruning power is, the larger numbers of pairs will be pruned; but the filtering cost will be higher. Thus it is important to make a tradeoff between them.

Although string similarity search and join have been extensively studied in decade, there is no thorough survey. In this paper, we present a comprehensive survey on string similarity search and join. We first give the problem definitions and then introduce similarity functions to quantify the similarity. Next, We present an extensive set of algorithms for string similarity search and join. We also discuss their variants, including approximate entity extraction, type-ahead search, and approximate substring matching. Finally, we provide some open datasets and summarize some research challenges and open problems.

The remainder of paper is organized as follows. In Section 2, we formally define the string similarity search and join problems and introduce their variants. We discuss the similarity search and join algorithms in Sections 3 and 4, respectively. The algorithms to their variants are introduced in Section 5. In Section 6, we provide several open datasets. Finally, we conclude our work and discuss research challenges and open problems in Section 7.

2 Problem formulation

In this section, we first introduce the similarity functions for quantifying the string similarity, and then formally define similarity search and join problems.

2.1 Similarity functions

Similarity functions are widely used to evaluate whether two strings are similar. Given two strings, we can use a similarity function to compute their similarity. If the similarity exceeds a given threshold, the two strings are said to be similar. The similarity functions can be broadly classified into three categories: token-based similarity, character-based similarity, and hybrid similarity.

• **Token-based similarity** The token-based similarity models each string as a set of tokens. These similarity functions measure the similarity of two strings based on the common tokens shared by their corresponding token sets. Consider two strings r and s. We also use them to denote their corresponding token sets if there is no ambiguity. Let $r \cap s$ denote the overlap of r and s, and $|r|$ denote the size of r.

The overlap similarity (OLP) takes the size of the overlap of their token sets as their similarity, i.e., $OLP(r, s) = |r \cap s|$. However, this function does not consider the sizes of the two token sets and similarity score is not normalized into [0,1]. To address these limitations, Jaccard similarity (JAC), Cosine similarity (COS) and Dice similarity ($DICE$) are proposed.

- Jaccard similarity: $JAC(r, s) = \dfrac{|r \cap s|}{|r \cup s|}$ $= \dfrac{|r \cap s|}{|r + |s| - |r \cap s|}$;
- Cosine similarity: $COS(r, s) = \dfrac{|r \cap s|}{\sqrt{|r| \cdot |s|}}$;
- Dice similarity: $DICE(r, s) = \dfrac{2|r \cap s|}{|r| + |s|}$.

For example, given two strings $r = \{frontier, computer, science\}$ and $s = \{computer, science\}$, their similarity scores based on above similarity functions are $OLP(r, s) = 2$, $JAC(r, s) = 2/3$, $COS(r, s) = 2/\sqrt{6}$, and $DICE(r, s) = 4/5$.

All the token-based similarities reply on the overlap, and thus to compute the token-based similarity, we can first compute the overlap size, and then the similarity can be easily calculated. To compute the overlap, we can first build a hash map for each token in a set, e.g., r, and then check whether each token in another set, e.g., s, appears in the hash map. If so, we increase the overlap size by 1. In this way, we can compute the overlap size with time complexity of $O(|r| + |s|)$. Obviously, the complexity of computing the token-based similarity is also $O(|r| + |s|)$.

• **Character-based similarity** The character-based similarity takes each string as a sequence of characters. It measures the similarity by counting the number of different characters in these two sequences.

The most representative character-based function is edit distance (ED), which calculates the minimum number of single-character edit operations (including insertion, deletion and substitution) when transforming one string to another. Thus two strings are said to be similar with respect to edit distance, if the minimum operation number is smaller than a give threshold.

To calculate edit distance for two strings, a simple method is to utilize the dynamic-programming algorithm. We can use a matrix to record the edit distances of prefixes of two strings, and each cell in this matrix is calculated based on minimum edit distance of their prefixes. Formally, let $r[1, i]$ denote the prefix of r with the first i characters, and $s[1, j]$ denote the prefix of s with the first j characters. Let T denote the matrix with $|r| + 1$ columns and $|s| + 1$ rows, and $T(i, j)$ is the edit distance of $r[1, i]$ and $s[1, j]$. To compute the edit distance between $r[1, i]$ and $s[1, j]$, we consider the transformations from $r[1, i]$ to $s[1, j]$. There are three cases:

Case 1 Insertion: We first transform $r[1, i]$ to $s[1, j - 1]$ and then insert the j-th character of s, i.e., $s[j]$. The number of edit operations is $T(i, j - 1) + 1$.

Case 2 Deletion: We first transform $r[1, i - 1]$ to $s[1, j]$ and then delete the i-th character of r, i.e., $r[i]$. The number of edit operations is $T(i - 1, j) + 1$.

Case 3 Match or substitution: We first transform $r[1, i-1]$ to $s[1, j - 1]$ and then match or substitute $r[i]$ with $s[j]$. If $r[i] = s[j]$, we use a match operation and the number of edit operations is $T(i-1, j-1)$. If $r[i] \neq s[j]$, we use a substitution operation and the number of edit operations is $T(i-1, j-1)+1$.

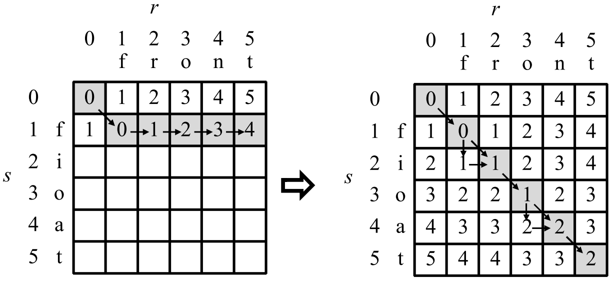
In this way, we can compute the matrix T as follows.

$$T(i, j) = \min \begin{cases} T(i, j + 1) + 1 & \text{Insertion;} \\ T(i - 1, j) + 1 & \text{Deletion;} \\ \begin{cases} T(i - 1, j - 1) & \text{Match} : r[i] = s[j], \\ T(i - 1, j - 1) + 1 & \text{Substitution} : r[i] \neq s[j], \end{cases} \end{cases} \tag{1}$$

where $T[0, j] = j$ and $T[i, 0] = i$.

Obviously $T(|s|, |r|)$ is the edit distance of r and s. The time complex for calculating their edit distance is $O(|s||r|)$. For instance, consider two strings $r = \{front\}$ and $s = \{float\}$, their edit distance is $ED(r, s) = 2$ as shown in Fig. 1.

Edit distance is a distance function. The smaller the edit distance is, the more similar the two strings are. To transform it to a similarity function, a character-based similarity function, edit similarity (EDS), is proposed to measure the character-based similarity. The edit similarity between two strings r and s is $EDS(r, s) = 1 - ED(r, s)/\max(|r|, |s|)$. For instance, consider two strings $r = \{front\}$ and $s = \{float\}$. Their edit similarity is $EDS(r, s) = 3/5$.



		0	1 f	2 r	3 o	4 n	5 t
0		0	1	2	3	4	5
1	f	1	0	1	2	3	4
2	i						
3	o						
4	a						
5	t						

		0	1 f	2 r	3 o	4 n	5 t
0		0	1	2	3	4	5
1	f	1	0	1	2	3	4
2	i	2	1	1	2	3	4
3	o	3	2	2	1	2	3
4	a	4	3	3	2	2	3
5	t	5	4	4	3	3	2

Fig. 1 Dynamic programming matrix for two strings

Hamming distance counts the number of mismatched characters in every position of two strings. For example, the hamming distance of “karolin” and “kathrin” is 3, because they have 3 different characters in positions 3, 4, 5[1]. Hamming distance is widely used in telecommunication to estimate errors by counting the number of flipped bit in a fixed-length binary word, so it is also called the signal distance. The time complexity of compute the hamming distance is $O(|s| + |r|)$.

Chaudhuri et al. [22] proposed a fuzzy edit-distance based similarity called generalized edit similarity (GES) which allows approximately matching in the transformations. GES takes each string as a sequence of tokens. For each token in one string, GES allows it approximately match a token in another string. In traditional edit distance, if two tokens are not matched, it uses a substitution with cost of 1. But in GES, it assigns a cost based on the edit distance and matches each token to the most similar token in another string. GES does not follow symmetry property. For example, consider two strings s_1=“wnba nba” and s_2=“nba”. As all tokens in s_1 are similar to “nba” in s_2, the similarity is $GES(s_1, s_2) = (0.75 + 1)/2 = 0.875$. But if we calculate $GES(s_2, s_1)$, since “nba” in s_2 is most similar with “nba” in s_1, $GES(s_2, s_1) = 1$.

Furthermore, they provide an approximation of generalized edit similarity (AGES) which can ignore the position of tokens.

- **Hybrid-based similarity** Wang et al. [23,24] proposed hybrid similarity functions. They also tokenize each string as a set of tokens. Different from the token-based similarity functions, they allow fuzzy matching between tokens. They use character-based similarity functions to quantify the similarity between tokens. Since the token-based similarities rely

[1] If two strings have different sizes, we can append some characters at the end of the short strings to make them have the same length

on the overlap, they also need to compute the fuzzy overlap. To this end, they use a bigraph to model two token sets, where the nodes are tokens. There is an edge between two nodes if their character-based similarity exceeds a given threshold and the edge weight is the similarity. Next, they find the maximal matching in the bigraph as the fuzzy overlap of the two token sets, where the matching is a set of edges such that any two edges do not share a common node in the set. Then they can use the fuzzy overlap to substitute the overlap, and thus can use the token-based similarity and define the hybrid similarity between two token sets.

- Fuzzy Overlap: $OLP(r, s) = |r\overline{\cap}s|$;
- Fuzzy Jaccard: $JAC(r, s) = \dfrac{|r\overline{\cap}s|}{|r + |s| - |r\overline{\cap}s|}$;
- Fuzzy Cosine: $COS(r, s) = \dfrac{|r\overline{\cap}s|}{\sqrt{|r| \cdot |s|}}$;
- Fuzzy Dice: $DICE(r, s) = \dfrac{2|r\overline{\cap}s|}{|r| + |s|}$,

where $|r\overline{\cap}s|$ is the weight of the maximal matching in the bigraph.

2.2 Threshold-based similarity search & join

Based on the similarity functions, we can formally define the string similarity search and join problems.

Definition 1 (string similarity search) Given a set of strings S, a query string Q, and a similarity function *Sim* and a similarity threshold τ, the string similarity search problem aims to find all the strings from S whose similarity scores to Q based on the similarity function exceeds a given threshold τ, i.e., $\{s|s \in S$ and $Sim(s, Q) \geqslant \tau\}$.

Definition 2 (string similarity join) Given two sets of strings R and S, a similarity function *Sim* and a threshold τ, string similarity join is to find a set of string pairs whose similarity scores are not smaller than τ, i.e., $\{(r, s)|r \in R, s \in S$ and $Sim(r, s) \geqslant \tau\}$.

Both of these two problems need a similarity threshold τ. For similarity measures, they find the results with similarity not smaller than the threshold τ; but for distance function, e.g., edit distance, they find the results with similarity not larger the threshold τ.

In practice, however, it is rather hard to get an appropriate threshold, because a large threshold returns a larger numbers of dissimilar results and a small threshold returns few and even empty result. To address this problem, the top-k similarity search and join are proposed.

2.3 Top-k similarity search & join

Definition 3 (top-k similarity search) Given a string set S, a query Q and a similarity function *Sim*, and an integer k, the top-k similarity search is to find k most similar strings from S that have the largest similarity to Q measured by *Sim*, i.e., $A \subseteq S$, and $\forall s \in A, s' \in S - A, Sim(s, Q) \leqslant Sim(s', Q)$.

Definition 4 (top-k similarity join) Given two string sets S and R and a similarity function *Sim* and an integer k, the top-k similarity join problem aims to find the k most similar string pairs form S and R which have the highest similarity score based on *Sim*, i.e., $A \subseteq S * R$, and $\forall(s, r) \in A, (s', r') \in S * R - A, Sim(s, r) \leqslant Sim(s', r')$.

2.4 Other variants

Since string similarity search and join have many real applications, there are some other variants to string similarity search and join.

- **Type-ahead search** Type-ahead search, also known as search-as-you-type, enables instant search that instantly returns results as a user types a query character by character. For example, when a user types “super” in IMDB[2], the system shows movies containing this keyword as prefix such as “supernatural” and “Batman v Superman: Dawn of Justice”. Obviously, this instant feedback can help the user to navigate the underlying data and easily get the answer. Different from “Autocompletion” [25–27] which treats the user’s query with multiple keywords as a single string, type-ahead search [28–30] can support multiple-keyword queries. For the single-*k*eyword queries, it treats the query as a partial keyword. Support the current query is w. The system returns strings that contain keywords with a prefix matching or similar to w. For the multiple-*k*eyword queries, the last keyword that the user is typing is taken as a prefix of a keyword, and the other keywords are taken as the complete keywords. In the exact search, the system needs to return record containing all the complete keywords and a keyword with a prefix matching the partial keyword. In the fuzzy search, the result needs to contain keywords similar to complete keywords and a keyword similar to the partial prefix.
- **Approximate entity extraction** Given a set of entities and a document, the entity extraction aims to find the substring from the documents corresponding to the predefined entities.

[2] http://www.imdb.com/

For example, suppose "Named entity recognition is a subtask of information extraction" is a document from Wikipedia, and "information extraction" and "entity recognition" are two entities. Entity extraction can help us find the two predefined entities. It has many real applications in the fields of information retrieval, molecular biology, bioinformatics, and natural language processing [31–33]. As the documents may contain errors and an entity may have multiple representations, there are many recent researches [34–38] focusing on approximate entity extraction, which aims to identify the substrings from the documents that are similar to the predefined entities.

• **Substring matching** Substring matching aims to find all the substrings from a given set of strings that are similar to a query. Different from the string similarity search problem, the results of substring matching need to contain substrings that are similar to the query, instead of the whole strings. In other words, no matter how many different tokens there are between a string and the query, this string could be an answer if one of its substrings is similar to the query. The challenge of substring matching problem is to quickly identify the result that contains a substring similar to the query. There are many algorithms [39–42] based on the efficient filtering-verification framework or dynamic programming method.

3 Algorithms for string similarity search

In this section, we discuss algorithms on string similarity search problem. We first introduce algorithms for threshold-based similarity search, and then present the top-k similarity search algorithms.

3.1 Threshold-based algorithms

A well-known technique to address the similarity search problem is count filtering. The basic idea is that if two strings are similar, they must have at least T common tokens or q-grams, where q-grams are substrings of a string. For example, consider $q = 2$. The 2-gram set of string "FCS" is $\{``FC", ``CS"\}$. It is easy to prove that, if two strings r and s are similar based on the edit distance, they should share at least $\max(|r|, |s|) - q + 1 - \tau * q$ q-grams, where τ denotes the threshold for edit distance [43].

Thus for character-based similarity, we can calculate T as

- Edit distance: $T = \max(|r|, |s|) - q + 1 - \tau * q$;
- Hamming distance: $T = \max(|r|, |s|) - q + 1 - \tau * q$;
- Edit similarity: $T = \max(|r|, |s|) - q + 1 - (\max(|r|, |s|) * (1 - \tau) * q$.

For token-based similarity, if two strings are similar, we can calculate T as follows.

- Jaccard similarity: $T = \dfrac{\tau}{1+\tau}(|r| + |s|)$;
- Cosine similarity: $T = \tau\sqrt{|r| \cdot |s|}$;
- Dice similarity: $T = \dfrac{\tau}{2}(|r| + |s|)$.

Then given a query Q, we need to calculate the threshold T. Since the threshold relies on both the query and the data string, we need to compute the threshold T that depends only on Q and can be used for all strings.

For character-based similarity functions, give a query Q, we calculate T as

- Edit distance: $T = |Q| - q + 1 - \tau * q$;
- Hamming distance: $T = |Q| - q + 1 - \tau * q$;
- Edit similarity: $T = |Q| - q + 1 - |Q| * (1 - \tau) * q$.

For token-based similarity, we can calculate T as follows.

- Jaccard similarity: $T = \dfrac{\tau}{1+\tau}(|Q| + l_{\min})$;
- Cosine similarity: $T = \tau\sqrt{|Q| * l_{\min}}$;
- Dice similarity: $T = \dfrac{\tau}{2}(|Q| + l_{\min})$,

where $l_{\min}$ is the minimal size for all strings in the data.

• **Indexing** To check whether a data sting has T common tokens or q-grams with the query Q, we need to build an effective index. To achieve this goal, we can build an inverted index, where the entries are tokens or q-grams, and each entry has an inverted list that keeps the strings containing the token or q-gram.

• **Algorithm framework** Given a query Q, we first generate its tokens or q-grams, and then retrieve the inverted lists of

Algorithm 1 Similarity search

```
Input: $S$: The set of strings; $Q$: query; $\tau$: Threshold
Output: $A$: Results
begin
    $A \leftarrow \phi$;
    Build inverted index for $S$;
    Generate the tokens or $q$-grams for $Q$;
    Calculate threshold $T$ for $Q$;
    for each token/q-gram $g$ do
        Retrieve inverted list of $q$;
    Compute candidate $c$ that appears at least $T$ times on the inverted lists;
    Verify $c$;
    if $c$ is similar to the query then
        $A$.add($c$);
end
```

these tokens or q-grams. Next, it identifies the strings that appear at least T times on these inverted lists (i.e., containing at least T common tokens with the query) and takes them as candidates. Finally, it needs to verify the candidates and removes the false positives. The pseudo code of this framework is shown in Algorithm 1.

There are several effective list-merge algorithms to identify the candidates that appear at least T times on the inverted lists of tokens or q-grams of the query. Algorithms 2–4 show the pseudo codes of three representative algorithms: ScanCount, MergeSkip and DevideSkip.

Algorithm 2 ScanCount(S,τ)

```
Input: $S$: The set of record ID list; $\tau$: Threshol
Output: $R$: Results
begin
    $C \leftarrow$ an array of $|S|$ counters and all elements = 0;
    for each list $l \in S$ containing the token in query do
        for each record ID $r \in l$ do
            $C[r]$++;
            if $C[r]=\tau$ then
                $R$.add($r$);
end
```

Algorithm 3 MergeSkip(S,τ)

```
Input: $S$: The set of record ID list; $\tau$: Threshold
Output: $R$: Results
begin
    Insert the frontier records of the list to a heap $H$;
    while $H \neq \phi$ do
        $t \leftarrow H$. top ; $n$=0;
        while $H$. top = $t$ do
            $H$. pop ; $n$++;
        if $n \geqslant \tau$ then
            $R$. add($t$);
            Push next record on each popped list to $H$;
        else
            Pop $\tau - n - 1$ smallest records from $H$;
            $t'$=$H$. top;
            for each record ID $s \in$ popped lists do
                $r \leftarrow$ the smallest record $\geqslant t'$ in $s$;
                $H$. push($r$);
end
```

• **ScanCount** It maintains an $|S|$-length array where each element in the array is initialized as 0. Then it scans the inverted list that contains the tokens of the query. For each record on these inverted list, ScanCount increases the count in the array corresponding to the record by one. Finally, the records with counts larger than the threshold are taken as the candidates.

Algorithm 4 DevideSkip(S,τ)

```
Input: $S$: The set of record ID list; $\tau$: Threshold
Output: $R$: Results
begin
    $L_{long} \leftarrow L$ longest list among $S$;
    $L_{short} \leftarrow$ the remaing shor list;
    $C \leftarrow$ MergeSkip ($L_{short}$, $\tau - l$);
    for each record ID $r \in C$ do
        $n$=0;
        for each inverted list$l \in L_{long}$ do
            if $r$ appears on $l$ then
                $n$++;
        if $n \geqslant \tau$ then
            $R$. add($r$);
end
```

• **MergeSkip** MergeSkip requires the string IDs on the inverted lists sorted, which is easy to implement when constructing the index. It maintains the first element of the inverted list lists as a heap. When finding similar strings for the query, MergeSkip pops the top element from the heap and increases the count of the string ID corresponding to the popped string. If the number is larger than a given threshold, this string can be consider as a candidate and it pushes next ID on each popped list into the heap. Otherwise, it updates the heap with new top elements of inverted lists after removing all the ID smaller than the current top one of the heap. It repeats these steps until the heap becomes empty.

• **DivideSkip** DivideSkip is the improved algorithm of MergeSkip. After generating strings' ID list, it sorts them based on their length and partitions them into two groups. One is a set of l longest lists and the other lists are assigned into the short list group. For the group with shorter list, DivideSkip directly uses MergeSkip to find strings appearing $T - l$ times as candidates, where T denotes the threshold of DivideSkip. Then for each record in the candidates, this algorithm checks whether it appears in the longer group and the total number of occurrences is larger than T. All the results satisfying these conditions are taken as the final candidates.

Comparing these three methods, ScanCount needs to process all the inverted list containing the tokens in the query. Therefore, for the inverted list with many records, it may not achieve high performance. MergeSkip can prune many irrelevant records by utilizing a heap to count the occurrence number of tokens for the query. The most efficient algorithm is DivideSkip which improves MergeSkip by avoiding scanning long inverted lists.

[3] http://fastss.csg.uzh.ch

• **FastSS**[3)] FastSS utilizes a neighborhood generation based method to support similarity search for edit distance. Given a threshold τ, for each string, it generates its neighborhoods by deleting i characters for $i \in [0, \tau]$. Thus suppose the string length is l. It has $\sum_{i=1}^{\tau} \binom{l}{i}$ neighborhoods. It is easy to prove that two strings are similar, they must have a same neighborhood. In this way, for each data string, FastSS indexes its neighborhoods using inverted lists. Then given a query, FastSS also generates its neighborhoods, retrieves the inverted lists of these neighborhoods and takes the strings on the inverted lists as the candidates. Finally, FastSS needs to verify the candidates.

• **V-Gram** Li et al. [44,45] proposed variable-length-grams to support similarity search. The basic idea is that the fix-length gram may be not efficient, as some grams may be very frequent and others are infrequent. To address this problem, we can judiciously select high-quality grams to avoid generating very frequent grams. They proposed effective techniques to generate variable length grams.

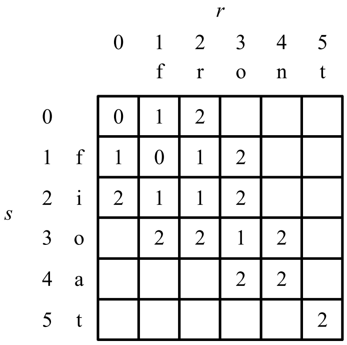
3.2 Top-k similarity search

Obviously, the threshold-based algorithms can be extended to support top-k similarity search problem by iteratively tuning the threshold τ. Initially, it estimates a threshold τ, and computes the results based on τ. If there are larger than k results, it can select k most similar results as the final answer. If there are smaller than k results, it needs to increase the threshold and recomputes the answer based on the new threshold. However this method is rather expensive, as it is rather hard to estimate an appropriate threshold.

To address this problem, there are some works studying new efficient algorithms for the top-k similarity search problem. An efficient way is to utilize a priori queue, which always calculates the "promising strings" that have the most probability to be in the final results. Obviously, it is rather challenging to identify the promising strings. To this end, there are some recent algorithms.

HS-Topk is a hierarchical framework to support top-k similarity search problem with edit distance [46]. The index structure they used is called "HS-Tree" which is a hierarchical segment tree index. It first groups the strings by length. Then it constructs a complete binary tree for each group of strings where the root is a dummy node. For each node in these binary trees, it partitions each string in its parent into two disjoint segments. The first segment is the prefix of the string with length $\lfloor len/2 \rfloor$, where len denotes the length of the string and the second segment is the suffix with length $\lceil len/2 \rceil$. Iteratively, this partition is terminated when one of the segment in this level has the length of 1. That means, the max level of an HS-Tree is $\lfloor \log_2 l \rfloor$, where l denotes the length of string with the maximal length. For example, consider a group of string {"*brother*", "*brothel*", "*broathe*"}. The first level nodes contain the segments "*bro*" and {"*ther*", "*thel*", "*athe*"}, respectively. For the second level, as one of segment "*b*" that is partitioned from "*bro*" is with length 1, the partition is terminated. To make this structure to support top-k similarity search, the basic idea is to first visit the promising strings with large probability to be similar to the query by pruning large numbers of dissimilar strings. To efficiently pruning unnecessary strings, two strategies are provided. One is greedy-match strategy that utilizes HS-tree structure to prune the strings with consecutive errors; the other one is a batch-pruning-based strategy which uses the largest edit distance of current k candidate results to compare with the lower bound of strings based on counting the number of matched segment when traversing the HS-tree to prune strings by edit distance. The details of the strategies is shown in Ref. [46].

Another study on top-k similarity search with edit-distance constraints is proposed by Deng et al. [47]. It improves the traditional dynamic-programming algorithm to calculate edit distance in order to avoid trying large numbers of edit-distance thresholds to select an appropriate one. Instead of calculating every entry in the matrix T which is utilized to compute the edit distance of two strings r and s with dynamic-programming method, this algorithm only computes part of entries of the matrix. To be specific, this algorithm initializes the entries with 0 edit distance, and then based on these, it computes the entries with 1 edit distance. Iteratively the entry $T[|r|][|s|]$ has been computed, and this is the edit distance of r and s. To avoid using threshold of edit distance, this algorithm constructs a trie structure in which each internal node contains a character and the strings are only stored in the leaf nodes. For instance, let us still consider the two strings "float" and "front" in the example of Section 2, the progressive methods on computing the edit distance are shown in Fig. 2. In Fig. 2, the computation terminates when $T[5][5]$ has been calculated. Obviously, this method is much more efficient than the traditional method as it avoids to compute many irrelevant cells in the matrix. To utilize the dynamic-programming method described before on the trie structure, it first finds all the nodes in the path that can be constructed as the prefix of the query for initialization. In other words, it initializes the entries with no edit distance. Then it traverses from these nodes by increasing the number of edit distance until k leaf nodes have been accessed and these k strings in these leaves are treated as results. Moreover, it presents an efficient method to reduce the number of nodes in each iteration for further improving the performance of calculating edit distance.



		0	1	2	3	4	5
			f	r	o	n	t
0		0	1	2			
1	f	1	0	1	2		
2	i	2	1	1	2		
3	o		2	2	1	2	
4	a				2	2	
5	t						2

Fig. 2 Computing edit distance with progressive dynamic programming

AppGram [48] studies the problem of KNN string similarity search with edit-distance constraints. It proposes a filter-and-verification approach by utilizing approximate q-gram matchings. To avoid visiting many dissimilar strings and verifying frequently, in the filter step, AppGram employs the CA strategy [49] and f-queue strategy which buffers the frequency of the approximate q-grams to support candidate selection. With these two strategies, AppGram uses a count filtering to pre-prune strings, and processes them in parallel based on the distance. For the strings with small distances, it uses a queue to prune, and others are used by two filtering strategies. Finally, it utilizes a max heap to maintain top-k similar strings with the query. With this pipeline method, AppGram can significantly reduce the overhead cost of employing multiple filtering techniques.

3.3 Algorithm summary

In this section, we summarize the algorithms discussed above and compare their advantages and disadvantages in Table 1 [44–48,50].

4 Algorithms for string similarity join

A straightforward method to string similarity join takes each data string in a set as a query, and uses the query to find similar strings in another set. However this method may not be efficient, as it cannot share the computations on the first set. To address this problem, many join algorithms have been proposed [51–53].

4.1 Filter-verification framework

A naive method to enumerate every pair from the two sets is expensive. To settle this problem, a filter-and-verification framework is proposed. In the filter step, it uses a lightweight filtering technique to identify a set of candidate pairs and prune lots of dissimilar pairs. In the verification step, it verifies every candidate pair and remove the false positives. Since the verification step is trivial, most of existing algorithms focus on the filter step.

• **Filtering** Specifically the filter step aims to design efficient filter algorithms for pruning. To balance pruning power and filtering cost, the algorithms should be “light-weight" for pruning. A common solution is to utilize the signature-based technique in the filter step: we generate the signature for each string and if two strings are similar, they must share at least one signature. Using this technique, we can utilize the inverted list to check whether two strings share a common signature. Therefore, when finding similar strings for a given string, we use the signatures in this query to find corresponding inverted lists, and the strings on these inverted lists are taken as candidates of this string. The pairs that have no common signatures will be pruned.

Table 1 Comparison on similarity search algorithms

Algorithm	Technique	Advantage	Disadvantage	Query	Function
FastSS[3)]	Inverted list, deletion	Efficient for short strings and small thresholds	High space complexity	Threshold	ED
V-Gram [44,45]	Variable length q-gram	High pruning power	High indexing overhead	Threshold	ED
HS-Topk [46]	Binary tree, greedy algorithm	Support various queries	Inefficient filter step for large thresholds	Threshold top-k	ED
DevideSkip [50]	Inverted list, heap	Support various functions easy to implement	Cannot support top-k	Threshold	JAC, COS, DICE, ED
TopkSearch [47]	Dynamic programming	Efficient pruning technique	Inefficient for long strings	Top-k	ED
AppGram [48]	Approximate grams, queue, heap	Efficient pruning strategy	High space complexity	Top-k	ED

• **Verification** In the verification step, a naive method is to directly compute the similarity of all the candidates. To avoid unnecessary calculation, some algorithms employ effective techniques and structures such as max heap, prior queue and dynamic programming to improve the verification. For instance, consider we use the dynamic programming technique as shown in Section 2 to find results whose edit distance are smaller than a given threshold τ. Given two strings r and s, a matrix T with $|r|+1$ rows and $|s|+1$ columns is used to compute their edit distance, and the edit distance of this two strings is the value of $T[|r|][|s|]$. Obviously, the other elements in T whose values are larger than τ do not need to be calculated further, as their length difference is already larger than the threshold. Thus for each row or column, only $2\tau+1$ values are required to compute. Therefore, the complexity can be improved to $O(\tau(|s|+|r|))$.

• **Algorithm framework** Algorithm 5 shows an example of signature-based filter-verification framework of RS-Join which processes string similarity join for two different string sets. In the initialization, we generate the signature for each strings in R and construct a set of inverted list I_R for signatures in R (Line 3). Then for each string s in another set S, we generate its signature set S_s (Line 5). For the signature in S_s, we find its corresponding inverted list in I_R and the strings on the inverted list are candidates of s. Next, we verify whether the strings in this list are similar with string s (Line 6–9). If the string r satisfies $Sim(r,s) > \tau$, we add this string pair into result set (Line 9).

Algorithm 5 Similarity join

```
Input: $S$,$R$: Two string sets ; $\tau$: Threshold
Output: $R$: Results
1  begin
2      $R \leftarrow \phi$;
3      $I_R \leftarrow$ inverted list set of $R$;
4      for each string $s \in S$ do
5          $S_s \leftarrow$ signature set of $s$;
6          for each signature $sig \in S_s$ do
7              for each string $r \in I_R[sig]$ do
8                  if $Sim(r,s) > \tau$ then
9                      $R$.add(< $s, r$ >);
10 end
```

4.2 Length filtering

Besides using count filtering to prune dissimilar strings, there are also some other filtering techniques can be used in the filter step.

Length filtering is a filtering technique that utilizes the length of strings for pruning. For example, the basic idea of GramCount [54] is that if two strings are similar, the length difference of them cannot be large. Thus, we can partition strings into several groups that the strings in the same group have the same length. The pruning condition of length filtering for strings from different groups is designed based on different similarity functions. Given two strings r and s, if they are similar strings, they should satisfy:

- Jaccard similarity: $\tau|s| \leqslant |r| \leqslant \frac{|s|}{\tau}$;
- Cosine similarity: $\tau^2|s| \leqslant |r| \leqslant \frac{|s|}{\tau^2}$;
- Dice similarity: $\frac{\tau}{2-\tau}|s| \leqslant |r| \leqslant \frac{2-\tau}{\tau}|s|$;
- Edit distance: $|s|-\tau \leqslant |r| \leqslant |s|+\tau$,

where τ denotes the threshold in each similarity function.

Therefore, the strings from the groups with lengths not satisfying these in-equations cannot be similar strings, and we can safely prune them.

4.3 Prefix filtering

The basic idea of prefix filtering [55] is to utilize a substring to evaluate their similarity. If the substrings are not similar enough, we can prune them. Algorithm 6 shows the framework of prefix filtering. First of all, it sorts all the distinct tokens from the string set based on a global order such as the alphabetical order, document frequency (DF) order or inverse document frequency (IDF) order, and reorders tokens in each string with this global order (Lines 3 and 4). After this initialization, we compare the prefixes of each string for two datasets (Lines 5–10). If there is an intersection between them, these two strings could be a result (Line 10). The prefix is selected based on similarity functions. With the functions shown in Section 2, consider a string s and a similarity function threshold τ, the length of prefix for each function can be computed as follows:

- Overlap similarity: $p = |s| - \tau + 1$;
- Jaccard similarity: $p = \lfloor (1-\tau) * |s| \rfloor + 1$;
- Cosine similarity: $p = \lfloor (1-\tau^2) * |s| \rfloor + 1$;
- Dice similarity: $p = \lfloor (1-\frac{\tau}{2-\tau}) * |s| \rfloor + 1$;
- Edit distance: $p = q * \tau + 1$.

There are many recent studies on string similarity based on the prefix filtering [56,61], and their goals are to shorten the prefix. We can classify these studies based on the similarity functions they support.

Algorithm 6 Prefix filtering

```
Input: $S, R$: Two string sets ; $\tau$: Threshold
Output: $R$: Results
begin
    $R \leftarrow \phi$;
    Build global ordering on all tokens;
    Sort tokens in each string with the global ordering;
    for each string $s \in S$ do
        $s_p \leftarrow s$.prefix;
        for each string $r \in R$ do
            $r_p \leftarrow r$.prefix;
            if $s_p \cap r_p \neq \phi$ then
                $R$.add($< s, r >$);
end
```

4.3.1 Algorithms for Jaccard, Consine and Dice

PPJoin [59] extends the prefix filtering technique to support token-based similarity join and shortens the prefix filtering by estimating the low bound of the union size and the upper bound of intersection set of two sets. In PPJoin, different string pairs have different thresholds of common token number (upper bound) which are generated based on the similarity threshold and the sizes of two sets. For example, given two sets r and s, and a threshold τ, we take Jaccard similarity (JAC) as an example to measure string similarity. That means if the two strings are similar, they should satisfy $JAC(r, s) = \dfrac{|r \cap s|}{|r \cup s|} \geqslant \tau$. So their upper bound is $\dfrac{\tau}{\tau + 1} \cdot (|r| + |s|)$, and we can use this threshold to implement prefix filtering on r and s. For the other similarity function such as Cosine similarity and Dice similarity, they can use similar regulations to generate thresholds and implement prefix filtering. They also use some other techniques to do early termination.

To estimate a tighter upper bound, PPJoin utilizes other filtering in the suffix of strings. It first chooses the middle token in the suffix of one string and uses it to do a binary search on the other string. With this position, it can get a new tighter upper bound. For instance, consider the suffixes in Fig. 3 and "?" denotes the unknown token. After getting the middle token Q from suffix r_s, it locates the position of Q in the other suffix s_s. As all the tokens are sorted, the tokens only on the same side may be the same. Therefore, these two suffixes can only share at most $2 + 1 + 3 = 6$ tokens. If we do not have the suffix filtering, we suppose all the tokens of r_s are in the suffix s_s, which means they share seven common tokens. Obviously, suffix filtering can help PPJoin to get tighter upper bound to improve pruning power.

$$\mathrm{r_s}\text{: } \{\underbrace{?, ?, ?}_{\mathrm{r_l}}, \mathrm{Q}, \underbrace{?, ?, ?}_{\mathrm{r_r}}\}$$
$$\mathrm{s_s}\text{: } \{\underbrace{?, ?}_{\mathrm{s_l}}, \mathrm{Q}, \underbrace{?, ?, ?, ?, ?}_{\mathrm{s_r}}\}$$

Fig. 3 Suffix filtering example

4.3.2 Algorithms for edit distance

- **ED-Join** Xiao et al. proposed an algorithm called Ed-Join [58] to shorten the prefix length for edit distance. In the filter step, Ed-Join employs two optimized filtering techniques to reduce the number of candidates obtained from analyzing the locations and contents of dissimilar string pairs. The location-based filtering can remove unnecessary q-grams to reduce the length of prefix. The content-based filtering is based on contents which utilizes a probing window to detect clustered mismatched substrings. For example, consider the two strings $s = c_1, c_2, \underline{c_3, c_4}, c_5, c_6$ and $t = c_1, c_2, \underline{e_1, e_2}, c_5, c_6$. Suppose $q = 2$ and the edit distance threshold $\tau = 1$. Suppose the probing window covers the underlined substrings. Since there are already two different characters, the two strings cannot be similar and Ed-Join can prune this string pair.
- **QChunk** QChunk [56,62] is an algorithm which supports edit distance for string similarity join. In this algorithm, they extract q-grams and q-chunks from data and query strings. Q-chunks are the set of q-length substrings which start from $1 + i * q$ positions $(0 \leqslant i \leqslant (|s| - 1)/q)$ for a string s. In other word, q-chunks are the disjoint q-grams of s. To make the last q-chunk has exactly q characters, we can use some spatial character such as \$ to fill the vacancy. For example, the q-chunk set of "FCS" is {"FC", "$S\$$"}. Let g_q and c_q denote the set of q-grams and q-chunks, respectively. QChunk algorithm proves that if two strings r and s are similar, that is, $ED(r, s) \leqslant \tau$, they should satisfy two properties:

  1) $|g_q(r) \cap c_q(s)| \geqslant \lceil \dfrac{|s|}{q} \rceil - \tau$, and

  2) $|c_q(r) \cap g_q(s)| \geqslant \lceil \dfrac{|r|}{q} \rceil - \tau$.

  Based on these two properties, QChunk provides an algorithm with two phases to prune dissimilar string pairs and implements string similarity join. It first generates the candidates with two strategies. The first strategy is to generate candidates by indexing q-grams of r and utilizing q-chunks of s to match the indexed q-grams, and the other is to index q-chunks of r and use q-grams of s to generate candidates. These two strategies are applied into prefix filtering by QChunk for string similarity join. The second phase is to use an effective technique to further reduce the number of candidates. The detail of QChunk can be found in Ref. [56].
- **Adapt prefix filtering** Wang et al. [63] improved the pre-

fix filtering by selecting more tokens into the prefix. Recall the prefix filtering, it selects the first p tokens as the prefix, and if two strings are similar, their prefixes share at least one common signature. If we select one more token into the prefix, then if two strings are similar, their prefixes share at least 2 common tokens. Similarly, if we select c more tokens into the prefix, then if two strings are similar, their prefixes share at least $c+1$ common tokens. Obviously, a large c reduces the number of candidates, but it has high filtering cost. Thus it is important to select an appropriate c. However it is rather challenging to select s as it is hard to accurately estimate the candidate number for a given c. To address this problem, Wang et al. proposed effective indexes and techniques to select an appropriate c.

• **Pivotal prefix** Deng et al. [64] proposed the pivotal prefix to shorten the prefix length. The basic idea is that for two strings, we can select different length of prefixes like QChunk. If we can shorten the prefix length, we can improve the performance. They studied how to reduce the prefix length with dynamic-programming algorithm and use an alignment filter to prune strings with consecutive errors, so that their algorithm can prune large numbers of dissimilar strings. The details of this algorithm is shown in Ref. [64].

4.4 Verification algorithms

It is efficient to verify a candidate for token-based similarity, but it is expensive for edit distance as discussed in Section 1. To address this problem, Li et al. [65,66] proposed progressive methods to verify the candidate for edit distance. The basic idea is that we can compute the exact edit distance of prefixes and then estimate the edit distance for suffixes using length filtering (or other filtering techniques). If the estimated edit distance is larger than the threshold, the candidate cannot be similar and we can prune the candidate.

Li et al. proposed a prefix tree to support string similarity join and search on multi-attribute data [67]. In the filter step, to holistically prune dissimilar results, they respectively propose a cost-based method and a budget-based method with high-quality prefix trees for similarity search and join problems. In the verification step, they design a hybrid verification algorithm to improve verification performance. It first sorts the filters of each attribute by their filtering performance. Then for each candidate generated by the prefix trees, this algorithm uses these filters to verify whether it could be a result, and updates the expected cost. If it can pass all these ordered filters, this candidate is added into the result set. Repeatedly, this algorithm can generate the final results. In addition, this algorithm can be extended to support other filtering techniques, including count filtering, length filtering, content filtering, and prefix filtering.

4.5 Other threshold-based algorithms

In this subsection, we introduce some other algorithms utilizing other indexing structures to support threshold-based similarity join.

• **M-tree** M-tree [68] is a classic tree-based index structure for metric-space similarity. In an m-tree, the distance between two different level nodes represents their similarity, and nodes with the same parent are divided by minimum "overlap". In addition, the strings are only stored in the leaf nodes and the internal nodes are used for pruning. The most important part of m-tree is the internal node. In an internal node N, it stores a landmark string N_m and a covering radius N_r. These two parameters represent all the children in the subtree rooted by N which have a distance to N_m no larger than N_r. Utilizing this structure, we can utilize the triangle inequality when traversing an m-tree to find similar strings for the query. Given a string query Q, a threshold τ and a function $d(*,*)$ for computing the distance between two strings, when we visit an internal node i, if $d(i_m, Q) > i_r + \tau$, node i can be pruned. Otherwise, we access its children. Iteratively, we can find all similar strings for the query. Different M-trees have different pruning power, so it is important to select a high-quality M-tree. There are several algorithms to improve it for similarity search problem [69,70].

• **Trie-Join** The trie structure is widely applied in string similarity problems [36,71,72]. Consider a token-based similarity. Before constructing a trie structure, it first fixes all the tokens from the dataset in a global order (e.g., the alphabetical order), and then uses this order to reorder tokens in each string. Finally, it constructs a trie structure. In the trie structure, each node stores a token (except the root), and each string from dataset corresponds to a unique path from the root to a leaf node. Strings with a common prefix share the same ancestors. For the character-based similarity, it has the same process to build a tree structure and the only difference is that the tree nodes are characters.

Trie-join [71,72] utilizes the trie structure to solve string similarity join problem. First it provides a definition called "active node". A node n in the trie is called an active node of a string s if their edit distance is not larger than the given threshold τ. In other words, all the descendants of n will be similar with s and all the other nodes cannot be similar to s. Therefore, the basic idea of Trie-join is if an active node

r of s is a leaf node, $< r, s >$ can be a similar string pair. To efficiently find string pairs based on this idea, Trie-Join incrementally constructs the trie structure and constructs the active node set for a node utilizing that of its parent, which can avoid repeated calculation when computing the active nodes among sibling nodes. To further improve the algorithm, Trie-join presents a partition-based method, which partitions strings into two parts and the threshold τ can reduce to $\tau/2$. With the two tries of these two parts of strings, it can efficiently find join results for large edit-distance thresholds.

- **Pass-Join** Pass-Join [65,66] utilizes the pigeon-hole principle to settle similarity join problem with edit-distance similarity. The basic idea is that given two strings r and s and an edit distance threshold τ, it splits r into $\tau + 1$ segments such that s is similar with r only if one of segments of r is a substring of s. For example, consider two strings "orange" and "apple" and the threshold is 2. We can split the first string as {"or", "an", "ge"}. As there is no segment of "orange" matching any substrings of "apple", these two strings cannot be similar. Based on this theory, Pass-Join creates a set of inverted indices for the $\tau + 1$ segments. Then for each string, it selects some of its substrings and utilizes them to find candidate pairs using the inverted indices. Finally, it verifies the candidate to generate results. To select substrings with the minimum size, Pass-Join develops efficient technique that the number of selected substrings can be reduced to $\lfloor (r^2 - \Delta^2)/2 \rfloor + \tau + 1$, where $\Delta = \big||r| - |s|\big|$ denotes the length difference between r and s.

Pass-Join can also support other similarity functions. For the token-based similarity functions, we can transform the threshold to the lower bound of common tokens between two strings and utilize this bound to generate the threshold. Using the functions described in Section 2, the thresholds for Jaccard, Cosine and Dice Similarity are $\tau = \lfloor l * (1 - \alpha)/\alpha \rfloor$, $\tau = \lfloor l * (1 - \alpha^2)/\alpha^2 \rfloor$, and $\tau = \lfloor l * [2(1 - \alpha)]/\alpha \rfloor$, respectively, where α denotes the original threshold in each function.

- **PARTENUM** Arasu et al. presented a partition-enumeration-based algorithm for set similarity join [73]. In this algorithm, it combines partition-based and enumeration-based method to solve the problem. For the partition-based method, each vector is divided into $\tau + k$ pieces, where τ is a threshold and k is a constant. The two vectors having a hamming distance no larger than τ must have at least k common pieces. Based on the partition-based method, the scheme of the enumeration-based method is that for each vector v, it picks k partitions in every possible way, and for each selection, it generates a signature by partitioning v into these k pieces. With the two methods, for a given vector v, PARTENUM partitions it into $(\tau + 1)/2$ segments and generates a set of signatures of each segment based on enumeration scheme with a new threshold $k = 1$. Therefore, in the PARTENUM algorithm, the partition-based method reduces generating signatures for vectors of smaller dimensions; for a smaller threshold, the enumeration-based method makes the number of signatures generated become more tractable for a smaller threshold.

4.6 Top-k similarity join

Xiao et al. [74] extended the prefix filtering to support top-k similarity join. It also first sorts the tokens in all strings in a global order. Then for each token set, it assigns each token with a weight which is the largest possible similarity of other sets to this set, in the case that they do not share the tokens before this token. For example, consider "FCS, Journal, Computer Science". The weights of the four tokens are 1, 0.75, 0.5, 0.25. If another set shares "Journal" with this set but without sharing "FCS", the maximum similarity is 0.75. In this way, it first accesses the first token and uses the weight as an upper bound. Then it always first visits the tokens with largest upper bounds. It uses a priority queue to store the current top-k candidates. If the similarity of the top-k candidates is larger than the largest upper bound, the algorithm can terminate.

- **B^{ed}-Tree** B^{ed}-tree is proposed to support both top-k and threshold-based similarity join problem with a B^+-tree structure [75]. It first transforms strings to integer values with some mapping functions which can support to construct a B^+-tree. When comparing two trees to find similar string pairs, for the two internal nodes accessed, the children node pairs which contain the strings with edit distance smaller than the current threshold can be further accessed. Iteratively, it terminates when visiting the leaf nodes, and verifies the strings on these nodes to check whether they could be the top-k results. Based on this general theory of using B^+-tree, to support similarity join problem, B^{ed}-tree uses several mapping functions to transform strings to integer values including dictionary order, gram counting order and gram location order. The transformations based on these three orders are described in Ref. [75].

4.7 Parallel similarity join

In some applications, the datasets are rather large, e.g., clustering web pages in search engines, so we need to use parallel techniques. To this end, there are some studies on similarity joins with map-reduce [76–80]. Most of them utilize the prefix to generate the key-value pairs. Deng et al. [76] proposed Mass-Join and utilized the segments to generate the key-value pairs. As the number of segments is smaller than that of grams, Mass-Join has better performance.

4.8 Using prefix filtering to improve similarity search

The prefix filtering technique can also be used to support the similarity search problem with a minor change. The difference between similarity search and join is that for join, we are given a threshold; but for similarity search, each query has its own threshold and we need to use an index to support all the thresholds. To address this problem, Wang et al. proposed a delta-index-based method [63]. They group the strings based on their lengths. In the same group, they build a delta index: for the first token, they build an inverted index and keep the maximal weight like the top-*k* join method; similarly they build an inverted index for other tokens. It is easy to prove that the sum of these delta indexes has the same size of the traditional inverted index on all tokens. Then given a query, they utilize the indexes with the similarity larger than the threshold to compute the answers.

4.9 Algorithm summary

To further illustrate similarity join algorithms, we summarize their techniques and compare their advantages and disadvantages in this section. Table 2 shows the details of summary and comparison.

5 Other variants

In this section, we introduce the algorithms for other variants for string similarity problems.

5.1 Type-ahead search

Li et al. [26,81,82] and Chaudhuri et al. [27] studied the type-ahead search problems. They utilize the trie structure to index the strings. Then given a query, they find the trie nodes that are similar to the query, called active nodes. The leaf descendants of the active nodes are the answers of the query. They study how to efficiently and incrementally identify the active nodes. Xiao et al. [83] recently proposed to use suffix to index the trie structure to improve the performance, at the cost of involving large indexes.

Li et al. [81] studied how to return top-*k* answers. They rank an answer by combing the edit distance to the query and the weight of each record. For the two basic list-accessing methods: random access and sort access, they develop a forward-list-based method and a heap-based method with list-materialization techniques for supporting these two access methods, respectively. The forward-list-based method utilizes a trie of keywords extracted from all records, an inverted list for the correspondence of records and keywords, and a forward index of records and keywords in it with its weight, to gradually narrow the candidates of results down. To support sort access, it uses a sorted list for complete keywords. For the partial keywords, it uses a max heap to order the records from inverted list of leaf nodes which has an ancestor path of this partial keywords. Then they can merge these two parts to get top-*k* results with the maximal weight. To further improve the performance, this algorithm utilizes a materialized list to reduce the number of lists in the max heap, and decreases the cost of push/pop operations on the heap. Moreover, they improve the efficiency of sort access by using a list-pruning technique for fuzzy type-ahead search.

Li et al. [84] also provide an algorithm that utilizes SQL for type-ahead search. This algorithm can support both single and multiple keyword queries. The algorithm maintains three tables to support type-ahead search. The first one is an inverted list of records and their keywords. The second table stores the sorted keywords in the alphabet order, and each keyword is assigned with an ID. Then it generates all the prefixes of each keyword, and construct, a new table to store prefixes and their ranges of keywords containing them. Utilizing

Table 2 Comparison on similarity join algorithms

Algorithm	Technique	Advantage	Disadvantage	Query	Function
GramCount [54]	Length filter	The first gram based work	High complexity	Threshold	JAC, COS, DICE, ED
QChunk [56]	Disjoint grams	High pruning power	Inefficient for long strings	Threshold	ED
ED-Join [58]	Content filter	Shorten the prefix	Cannot support JAC	Threshold	ED
PPJoin [59]	Prefix & suffix filter	Efficient filter technique	Cannot support ED	Threshold	JAC, COS, DICE
Pass Join [65,66]	Partition	High pruning power	–	Threshold	JAC, COS, DICE, ED
M-tree [68]	Index tree, cluster	Metric space	Inefficient than others	Threshold	JAC, COS, DICE, ED
Trie-Join [71,72]	Trie	Efficient for short strings	Expensive for long strings	Threshold	JAC, COS, DICE, ED
Topk-Join [74]	Prefix filter priori queue	Efficient for short strings	High space complexity	Top-*k*	JAC, COS, DICE, ED
B^{ed}-Join [75]	B^{+}-tree	Small index	Poor filter power	Top-*k*	ED
PartEnum [73]	Partition	efficient for small threshold	High time complexity	Threshold	JAC, COS, DICE, ED

these three tables, when a user is typing a query, the algorithm, for each keyword, utilizes the prefix table to find the range of keywords and uses the inverted list to get the records containing these keywords.

For the fuzzy search, it proposes a new neighborhood-generation-based method. The basic idea is that two strings are similar only if they have common neighbors obtained by deleting characters from the strings. With a word-level incremental method that uses the previously computed result for searching next keyword, this algorithm can efficiently support multiple-*k*eyword queries.

5.2 Approximate entity extraction algorithm

Wang et al. [35] proposed a neighborhood based method to support entity extraction. They first generate the neighborhoods for each entity and use the inverted lists to index the neighborhoods. Then they scan the document and utilize the neighborhoods to identify the candidates. They also discuss how to reduce the neighborhood number.

Faerie [23,38,85,86] is a unified framework for supporting multiple similarity functions for approximate dictionary-based entity extraction. Their basic idea is that many substrings in the document have overlap and it can avoid redundant computation by sharing the calculation across the overlaps. In this unified framework, it considers each entity and document as a set of tokens where the token is a q-gram of a string. With these token sets, it transforms the similarity/dissimilarity as the overlap similarity. Therefore, consider two strings r and s. The conditions for making these two string similar with transformed similarity functions are:

- Jaccard similarity: $|r \cap s| \geqslant \lceil (|r| + |s|) * \frac{\tau}{1+\tau} \rceil$;
- Cosine similarity: $|r \cap s| \geqslant \sqrt{|r||s|} * \tau$;
- Dice similarity: $|r \cap s| \geqslant \lceil (|r| + |s|) * \frac{\tau}{2} \rceil$;
- Edit distance: $|r \cap s| \geqslant \max(|r|, |s|) - \tau * q$;
- Edit similarity: $|r \cap s| \geqslant \lceil \max(|r|, |s|) + q - 1 - \max(|r|, |s|) * (1-\tau) * q \rceil$,

where τ denotes the original threshold in each other similarity function.

With these transformed functions, Faerie presents a heap-based filtering algorithm to utilize the shared computation across overlaps, which constructs a single heap on top of inverted lists of tokens in the document and scans every inverted list only once. The entity and the substrings of the document with the occurrence number no smaller than the lower bound of the overlap similarity that are calculated by a binary-search-based technique, could be a candidate. Eventually, it can generate results by verifying the candidates.

Deng et al. [36] develop a trie-based method to solve approximate entity extraction problem. In this algorithm, it first partitions each entity into different disjoint segments and constructs a trie structure for these segments. Therefore, for the substrings in the document, it can use this trie to find approximate matching entities based on the fact that if a substring of document approximately matches this entity, it must contain at least one segment of the entity [87]. It develops a search-and-extension method for identifying answers, which fist generates the candidate entities when traversing the trie, and then verifies them to check whether they are results. To avoid involving large numbers of segments, it uses a length-based pruning and even-partition weight as an upper bound to improve performance. Different from Faerie which needs to involve large index size for the gram-based index structure, segments in this algorithm are disjoint, so it has less space complex than Faerie.

5.3 Substring matching algorithm

Kim et al. [39] present an algorithm for top-k approximate substring matching to find k strings containing substrings that are most similar to the query for edit distance. In this algorithm, it develops a dynamic programming method to evaluate the lower bound of substrings' similarity, improves the method with skipping irrelevant strings, and efficiently locates the substrings with a q-gram inverted list. With the lower bound lb, it generates common positional q-gram list for each strings by reading the posting lists of q-grams in query string sequentially in increasing order of string IDs, and calculates first k strings' similarity as current results. Then, for the other strings, if the lower bound of similarity between a string and the query is smaller than the kth smallest string in the results, it computes the actual similarity value to verify whether this is a result. When the edit distance of the kth string of results becomes at most lb, it updates lb and selects a distinct q-gram set with size lb in which each q-gram has no overlap, so that it can skip computing the edit distance of query and strings which has no q-gram in this set. Therefore, whenever the edit distance of the kth smallest string in the result decreases, it constructs a new q-gram set to skip computation. Repeatedly, after all the strings in the dataset have been computed or skipped, it can generate the final results.

Ge et al. [41] studied the approximate substring matching over uncertain strings. As uncertainty usually incurs considerable large index, it generates a q-gram index which guarantees that the increased index due to uncertainty can be small and tunable, since q is a system parameter and is usually small [88]. For this index, it proposes a multilevel filtering technique based on measuring signature distance. It uses a table to store signatures. After scanning this tree and generating candidates, it verifies the candidates based on the upper and lower bounds that are calculated by their algorithm to get answers. Based on the q-gram index and verification technique, this algorithm can significantly reduce the cost.

6 Open datasets

To compare the string similarity search and join algorithms, we need to use appropriate datasets. We introduce some open datasets that were widely used in previous work. The details of these datasets are shown in Tables 3 and 4.

- **Querylog**[4] is a set of query log strings and the average length is 18.9.
- **Word**[5] is a set of real English words with 8.7 average length.
- **Geome**[6] is a dataset of human gene sequences.
- **DBLP**[7] is a real dataset for publication, and consists of title, authors and its provenance.
- **PubMed**[8] is a title set in which each title is obtained from medical publications.
- **Trec**[9] is a set of documents from the well-*k*nown benchmark in information retrieval.
- **Enron**[10] includes a set of emails with titles and bodies.
- **Wiki**[11] includes a set of English wikipedia webpages.

The first three datasets can be used for character-based similarity, and the last five datasets can be used for token-based similarity.

Table 3 Datasets for character-based similarity

Dataset	Cardinality	Avg Len	Max Len	Size/M
Word	122 823	8.7	29	1.2
Querylog	500 000	18.9	500	9.7
Genome	250 000	100	100	25

Table 4 Datasets for token-based similarity

Dataset	Cardinality	Avg Tok	Max Tok	Size
DBLP	1 088 728	10.73	43	7.5G
PubMed	504 881	5.48	26	4.4G
Trec	347 949	75	273	103M
Enron	245 567	135	3 162	129M
Wiki	4 000 000	213	36 907	3.2G

Jiang et al. provided a good experimental study on string similarity joins [89]. There was also a competition on string similarity search join and search organized by EDBT 2013 [90,91]. Many open datasets and experimental results were provided in these studies.

7 Conclusion and open problems

In this paper, we provide a comprehensive survey on string similarity search and join. We formalize the problems of string similarity search and join and other variants. For the string similarity search and join problems, we introduce the filtering-and-verification framework. For similarity search, we introduce the list-merge algorithms. For similarity join, we introduce the prefix filtering technique. We also discuss other effective techniques. For the other variants including type-ahead search, approximate entity extraction and approximate substring matching, we also discuss recent studies to these problems. Finally, some open datasets are provided.

There are also many open problems.

1) New similarity measures

Although edit distance and Jaccard are widely used, they may not be power enough in real applications, especially in big data era. Therefore, it is very important to learn new similarity measures based on some given matching examples.

2) Crowdsourced similarity join and search

In many applications, similarity measures may not work well, and it is rather hard to identify similar pairs because we need to utilize human recognition techniques to judge whether two strings are similar. Alternatively, we can ask the human to determine whether two strings are similar. Thus crowdsourced similarity join and search is a new research direction.

3) Using knowledge base to improve the similarity search

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

[5] http://dbgroup.cs.tsinghua.edu.cn/ligl/simjoin/

[6] http://www.1000genomes.org/

[7] http://dblp.uni-trier.de/xml/

[8] http://www.ncbi.nlm.nih.gov/pubmed

[9] http://trec.nist.gov/data/t9_filtering.html

[10] http://www.cs.cmu.edu/ enron/

[11] http://dumps.wikimedia.org/

and join

Existing studies only consider the textual similarity but do not consider the semantics behind the data. Obviously, the semantics is rather important in evaluating the similarity. Thus we can utilize the knowledge base to quantify the similarity, and it is rather challenging to define knowledge-aware similarity and devise efficient algorithms to support this new similarity.

4) Similarity join and search on multiple attribute data

Most of existing algorithms focus on single attribute data. However in practice, the data have multiple attributes, e.g., structured data. It is important to support similarity search on multiple attribute data.

5) Similarity search and join systems

Most of existing methods employ a standalone way to study the search and join algorithms, but do not push the techniques into RDBMS. If we can deploy the techniques into RDBMS, it can benefit many applications using RDBMS.

Acknowledgements This work was partly supported by the National Grand Fundamental Research 973 Program of China (2015CB358700), the National Natural Science Foundation of China (Grant Nos. 61422205, 61472198), Beijing Higher Education Young Elite Teacher Project(YETP0105), Tsinghua-Tencent Joint Laboratory for Internet Innovation Technology, "NExT Research Center", Singapore (WBS:R-252-300-001-490), Huawei, Shenzhou, FDCT/116/2013/A3, MYRG105(Y1-L3)-FST13-GZ, National High-Tech R&D (863) Program of China (2012AA012600), and the Chinese Special Project of Science and Technology (2013zx01039-002-002).

References

1. Zhang C J, Chen L, Tong Y, Liu Z. Cleaning uncertain data with a noisy crowd. In: Proceedings of the 31st IEEE International Conference on Data Engineering. 2015, 6–17
2. Papotti P, Naumann F, Kruse S. Estimating data integration and cleaning effort. In: Proceedings of International Conference on Extending Database Technology. 2015, 61–72
3. Chu X, Morcos J, Ilyas I F, Ouzzani M, Papotti P, Tang N, Ye Y. KATARA: a data cleaning system powered by knowledge bases and crowdsourcing. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015, 1247–1261
4. Verma P, Kesswani N. Web usage mining framework for data cleaning and IP address identification. 2014, arXiv: 1408.5460v1
5. Maccio V J, Chiang F, Down D G. Models for distributed, large scale data cleaning. In: Proceedings of Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining. 2014, 369–380
6. Almeida R, Oliveira P, Braga L, Barroso J. Ontologies for reusing data cleaning knowledge. In: Proceedings of International Catholic Stewardship Council. 2012, 238–241
7. Fan J, Li G, Zhou L, Chen S, Hu J. SEAL: spatio-textual similarity search. Proceedings of the VLDB Endowment, 2012, 5(9): 824–835
8. Yu M, Li G, Wang T, Feng J, Gong Z. Efficient filtering algorithms for location-aware publish/subscribe. IEEE Transactions on Knowledge and Data Engineering, 2015, 27(4): 950–963
9. Li G, Ooi B C, Feng J, Wang J, Zhou L. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In: Proceedings of ACM SIGMOD International Conference on Management of Data. 2008, 903–914
10. Badgeley M A, Sealfon S C, Chikina M D. Hybrid Bayesian-rank integration approach improves the predictive power of genomic dataset aggregation. Bioinformatics, 2015, 31(2): 209–215
11. Lui T, Tsui N, Chan L W, Wong C, Siu P, Yung B Y M. DECODE: an integrated differential co-expression and differential expression analysis of gene expression data. BMC Bioinformatics, 2015, 16: 182
12. Arfaoui N, Akaichi J. Automating schema integration technique case study: generating data warehouse schema from data mart schemas. Communications in Computer and Information Science, 2015, 521: 200–209
13. Nastase V, Fahrni A. Coarse-grained cross-lingual alignment of comparable texts with topic models and encyclopedic knowledge. 2014, arXiv: 1411.7820v1
14. Srikantaiah K C, Suraj M, Venugopal K R, Patnaik L.M. Similarity based dynamic web data extraction and integration system from search engine result pages for web content mining. ACEEE International Journal on Information Technology, 2013, 3(1): 42–49
15. Cevahir A. Scalable textual similarity search on large document collections through random indexing and K-means clustering. In: Proceedings of Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining. 2014, 231–238
16. Yin J, Wang J. A dirichlet multinomial mixture model-based approach for short text clustering. In: Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2014, 233–242
17. Dai Z, Sun A, Liu X. Crest: cluster-based representation enrichment for short text classification. In: Proceedings of Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining. 2013, 256–267
18. SureshReddy G, Rajinikanth T V, Rao A A. Design and analysis of novel similarity measure for clustering and classification of high dimensional text documents. In: Proceedings of the 15th International Conference on Computer Systems and Technologies. 2014, 194–201
19. Liu S, Li G, Feng J. A prefix-filter based method for spatio-textual similarity join. IEEE Transactions on Knowledge and Data Engineering, 2014, 26(10): 2354–2367
20. Wang J, Li G, Kraska T, Franklin M J, Feng J. Leveraging transitive relations for crowdsourced joins. In: Proceedings of ACM SIGMOD International Conference on Management of Data. 2013, 229–240
21. Wang J, Li G, Yu J X, Feng J. Entity matching: how similar is similar. Proceedings of the VLDB Endowment, 2011, 4(10): 622–633
22. Chaudhuri S, Ganjam K, Ganti V, Motwani R. Robust and efficient fuzzy match for online data cleaning. In: Proceedings of ACM SIGMOD international conference on Management of data. 2003, 313–324
23. Wang J, Li G, Feng J. Fast-join: an efficient method for fuzzy token matching based string similarity join. In: Proceedings of the 27th IEEE International Conference on Data Engineering. 2011, 458–469
24. Wang J, Li G, Feng J. Extending string similarity join to tolerant fuzzy token matching. ACM Transactions on Database Systems, 2014, 39(1):

7

25. Nandi A, Jagadish H V. Effective phrase prediction. In: Proceedings of the 33rd International Conference on Very Large Databases, 2007, 219–230

26. Ji S, Li G, Li C, Feng J. Efficient interactive fuzzy keyword search. In: Proceedings of the 18th International Conference on World Wide Web. 2009, 371–380

27. Chaudhuri S, Kaushik R. Extending autocompletion to tolerate errors. In: Proceedings of ACM SIGMOD International Conference on Management of Data. 2009, 707–718

28. Zheng Y, Bao Z, Shou L, Tung A K. MESA: a map service to support fuzzy type-ahead search over geo-textual data. Proceedings of the VLDB Endowment, 2014, 7(13): 1545–1548

29. Li G, Ji S, Li C, Feng J. Efficient type-ahead search on relational data: a TASTIER approach. In: Proceedings of ACM SIGMOD International Conference on Management of Data. 2009, 695–706

30. Kavila S D, Ravva R, Bandaru R. Fuzzy type-ahead keyword search in RDF data. In: Proceedings of International Conference on Frontiers of Intelligent Computing: Theory and Applications. 2013, 67–73

31. Chandel A, Nagesh P C, Sarawagi S. Efficient batch top-*k* search for dictionary-based entity recognition. In: Proceedings of the 22nd IEEE International Conference on Data Engineering. 2006, 28

32. Cowan B, Zethelius S, Luk B, Baras T, Ukarde P, Zhang D. Named entity recognition in travel-related search queries. In: Proceedings of Association for the Advancement of Artificial Intelligence Conference, 2015, 3935–3941

33. Tang Z, Jiang L, Yang L, Li K, Li K. CRFs based parallel biomedical named entity recognition algorithm employing MapReduce framework. Cluster Computing, 2015, 18(2): 493–505

34. Lu W, Fung G P C, Du X, Zhou X, Chen L, Deng K. Approximate entity extraction in temporal databases. World Wide Web, 2011, 14(2): 157–186

35. Wang W, Xiao C, Lin X, Zhang C. Efficient approximate entity extraction with edit distance constraints. In: Proceedings of ACM SIGMOD International Conference on Management of Data. 2009, 759–770

36. Deng D, Li G, Feng J. An efficient trie-based method for approximate entity extraction with edit-distance constraints. In: Proceedings of the 28th IEEE International Conference on Data Engineering. 2012, 762–773

37. Nakajima D, Mitsui Y, Samejima M, Akiyoshi M. An information extraction method from different structural web sites by word distances between a user instantiated label and similar entity. In: Proceedings of International Conference on Data Mining Workshops, 2011, 1177–1182

38. Deng D, Li G, Feng J, Duan Y, Gong Z. A unified framework for approximate dictionary-based entity extraction. The International Journal on Very Large Data Bases, 2015, 24(1): 143–167

39. Kim Y, Shim K. Efficient top-*k* algorithms for approximate substring matching. In: Proceedings of ACM SIGMOD International Conference on Management of Data. 2013, 385–396

40. Tang N, Sidirourgos L, Boncz P A. Space-economical partial gram indices for exact substring matching. In: Proceedings of the 18th ACM International Conference on Information and Knowledge Management. 2009, 285–294

41. Ge T, Li Z. Approximate substring matching over uncertain strings. Proceedings of the VLDB Endowment, 2011, 4(11): 772–782

42. Warren R H, Tompa F W. Multi-column substring matching for database schema translation. In: Proceedings of the 32nd International Conference on Very Large Databases. 2006, 331–342

43. Jokinen P, Ukkonen E. Two algorithms for approximate string matching in static texts. In: Proceedings of the 16th International Symposium on Mathematical Foundations of Computer Science. 1991, 240–248

44. Li C, Wang B, Yang X. VGRAM: improving performance of approximate queries on string collections using variable-length grams. In: Proceedings of the 33rd International Conference on Very Large Data Bases. 2007, 303–314

45. Yang X, Wang B, Li C. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In: Proceedings of ACM SIGMOD International Conference on Management of Data. 2008, 353–364

46. Wang J, Li G, Deng D, Zhang Y, Feng J. Two birds with one stone: an efficient hierarchical framework for top-*k* and threshold-based string similarity search. In: Proceedings of International Conference on Data Engineering. 2015, 519–530

47. Deng D, Li G, Feng J, Li W S. Top-*k* string similarity search with edit distance constraints. In: Proceedings of the 29th IEEE International Conference on Data Engineering. 2013, 925–936

48. Wang X, Ding X, Tung A K H., Zhang Z. Efficient and effective KNN sequence search with approximate n-grams. Proceedings of the VLDB Endowment, 2013, 7(1): 1–12

49. Fagin R, Lotem A, Naor M. Optimal aggregation algorithms for middleware. Journal of Computer and System Sciences, 2003, 66(4): 614–656

50. Li C, Lu J, Lu Y. Efficient merging and filtering algorithms for approximate string searches. In: Proceedings of the 24th IEEE International Conference on Data Engineering. 2008, 257–266

51. Siragusa E, Weese D, Reinert K. Scalable string similarity search/join with approximate seeds and multiple backtracking. In: Proceedings of EDBT/ICDT Joint Conference. 2013, 370–374

52. Liu X, Li G, Feng J, Zhou L. Effective indices for efficient approximate string search and similarity join. In: Proceedings of the 9th IEEE International Conference on Web-Age Information Management. 2008, 127–134

53. Cui J, Meng D, Chen Z. Leveraging deletion neighborhoods and trie for efficient string similarity search and join. Lecture Notes in Computer Science, 2014, 8870: 1–13

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

55. Chaudhuri S, Ganti V, Kaushik R. A primitive operator for similarity joins in data cleaning. In: Proceedings of the 22nd IEEE International Conference on Data Engineering. 2006, 5

56. Qin J, Wang W, Lu Y, Xiao C, Lin X. Efficient exact edit similarity query processing with the asymmetric signature scheme. In: Proceedings of ACM SIGMOD International Conference on Management of Data. 2011, 1033–1044

57. Rong C, Lu W, Wang X, Du X, Chen Y, Tung A K H. Efficient and scalable processing of string similarity join. IEEE Transactions on Knowledge and Data Engineering, 2013, 25(10): 2217–2230

58. Xiao C, Wang W, Lin X. Ed-Join: an efficient algorithm for similarity joins with edit distance constraints. Proceedings of the VLDB Endowment, 2008, 1(1): 933–944

59. Xiao C, Wang W, Lin X, Yu J X. Efficient similarity joins for near duplicate detection. In: Proceedings of International World Wide Web Conference. 2008, 131–140

60. Xiao C, Wang W, Lin X, Yu J X, Wang G. Efficient similarity joins for near-duplicate detection. ACM Transactions on Database Systems, 2011, 36(3): 15

61. Wang W, Qin J, Xiao C, Lin X, Shen H T. VChunkJoin: an efficient algorithm for edit similarity joins. IEEE Transactions on Knowledge and Data Engineering, 2013, 25(8): 1916–1929

62. Qin J, Wang W, Xiao C, Lu Y, Lin X, Wang H. Asymmetric signature schemes for efficient exact edit similarity query processing. ACM Transactions on Database Systems, 2013, 38(3): 16

63. Wang J, Li G, Feng J. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In: Proceedings of ACM SIGMOD International Conference on Management of Data. 2012, 85–96

64. Deng D, Li G, Feng J. A pivotal prefix based filtering algorithm for string similarity search. In: Proceedings of ACM SIGMOD International Conference on Management of Data. 2014, 673–684

65. Li G, Deng D, Wang J, Feng J. PASS-JOIN: a partition-based method for similarity joins. Proceedings of the VLDB Endowment, 2011, 5(3): 253–264

66. Li G, Deng D, Feng J. A partition-based method for string similarity joins with edit-distance constraints. ACM Transactions on Database Systems, 2013, 38(2): 9

67. Li G, He J, Deng D, Li J. Efficient similarity join and search on multi-attribute data. In: Proceedings of ACM SIGMOD International Conference on Management of Data. 2015, 1137–1151

68. Ciaccia P, Patella M, Zezula P. M-tree: an efficient access method for similarity search in metric spaces. In: Proceedings of International Conference on Very Large Databases. 1997, 426–435

69. Aßfalg J, Borgwardt K M, Kriegel H P. 3D String: a feature string kernel for 3D object classification on voxelized data. In: Proceedings of the 15th ACM International Conference on Information and Knowledge Management. 2006, 198–207

70. Bartolini I, Ciaccia P, Patella M. String matching with metric trees using an approximate distance. In: Proceedings of the 9th International Symposium on String Processing and Information Retrieval. 2002, 271–283

71. Wang J, Feng J, Li G. Trie-join: efficient trie-based string similarity joins with edit-distance constraints. Proceedings of the VLDB Endowment, 20103, (1–2): 1219–1230

72. Feng J, Wang J, Li G. Trie-join: a trie-based method for efficient string similarity joins. The International Journal on Very Large Data Bases, 2012, 21(4): 437–461

73. Arasu A, Ganti V, Kaushik R. Efficient exact set-similarity joins. In: Proceedings of the 32nd International Conference on Very Large Data Bases. 2006, 918–929

74. Xiao C, Wang W, Lin X, Shang H. Top-*k* set similarity joins. In: Proceedings of the 25th IEEE International Conference on Data Engineering. 2009, 916–927

75. Zhang Z, Hadjieleftheriou M, Ooi B C, Srivastava D. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In: Proceedings of ACM SIGMOD International Conference on Management of Data. 2010, 915–926

76. Deng D, Li G, Hao S, Wang J, Feng J. Massjoin: a MapReduce-based method for scalable string similarity joins. In: Proceedings of the 30th IEEE International Conference on Data Engineering. 2014, 340–351

77. Afrati F N, Sarma A D, Menestrina D, Parameswaran A G, Ullman J D. Fuzzy joins using MapReduce. In: Proceedings of the 28th IEEE International Conference on Data Engineering. 2012, 498–509

78. Vernica R, Carey M J, Li C. Efficient parallel set-similarity joins using MapReduce. In: Proceedings of ACM SIGMOD International Conference on Management of Data. 2010, 495–506

79. Metwally A, Faloutsos C. V-SMART-join: a scalable MapReduce framework for all-pair similarity joins of multisets and vectors. Proceedings of the VLDB Endowment, 2012, 5(8): 704–715

80. Deng D, Jiang Y, Li G, Li J, Yu C. Scalable column concept determination for web tables using large knowledge bases. Proceedings of the VLDB Endowment, 2013, 6(13): 1606–1617

81. Li G, Wang J, Li C, Feng J. Supporting efficient top-*k* queries in type-ahead search. In: Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval. 2012, 355–364

82. Li G, Ji S, Li C, Feng J. Efficient fuzzy full-text type-ahead search. The International Journal on Very Large Data Bases, 2011, 20(4): 617–640

83. Xiao C, Qin J, Wang W, Ishikawa Y, Tsuda K, Sadakane K. Efficient error-tolerant query autocompletion. Proceedings of the VLDB Endowment, 2013, 6(6): 373–384

84. Li G, Feng J, Li C. Supporting search-as-you-type using SQL in databases. IEEE Transactions on Knowledge and Data Engineering, 2013, 25(2): 461–475

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

86. Li G, Hu J, Feng J, Tan K. Effective location identification from microblogs. In: Proceedings of the 30th IEEE International Conference on Data Engineering. 2014, 880–891

87. Ukkonen E. Approximate string matching with q-grams and maximal matches. Theoretical Computer Science, 1992, 92(1): 191–211

88. Navarro G, Baeza-Yates R A, Sutinen E, Tarhio J. Indexing Methods for Approximate String Matching. IEEE Data Engineering Bulletin, 2001, 24(4): 19–27

89. Jiang Y, Li G, Feng J, Li W. String similarity joins: an experimental evaluation. Proceedings of the VLDB Endowment, 2014, 7(8): 625–636

90. Jiang Y, Deng D, Wang J, Li G, Feng J. Efficient parallel partition-based algorithms for similarity search and join with edit distance constraints. In: Proceedings of the Joint EDBT/ICDT 2013 Workshops. 2013, 341–348

91. Wandelt S, Deng D, Gerdjikov S, Mishra S, Mitankin P, Patil M, Siragusa E, Tiskin A,Wang W, Wang J, Leser U. State-of-the-art in string similarity search and join. ACM SIGMOD Record, 2014, 43(1): 64–76

Minghe Yu is currently a PhD candidate in the Department of Computer Science, Tsinghua University, China. She obtained her Bachelor degree in Department of Computer Science, Northeast University, China. Her research interests include data integration and spatio-textual data query.

Guoliang Li is an associate professor in the Department of Computer Science, Tsinghua University, China. He received his PhD in computer Science from Tsinghua University in 2009. He is an Awardee of the NSFC Excellent Young Scholars Program in 2014. His research interests include data cleaning and integration, spatial databases, and crowdsourcing.

Dong Deng is a PhD candidate in the Department of Computer Science, Tsinghua University, China. He obtained his Bachelor degree in Department of Computer Science, Beihang University, China. His research interests include data cleaning and integration, and database usability.

Jianhua Feng received his BS, MS and PhD degrees in computer science from Tsinghua University, China. He is currently working as a professor in Department of Computer Science, Tsinghua University. His main research interests include large-scale data management and analysis.