Two Birds With One Stone: An Efficient Hierarchical Framework for Top-k and Threshold-based String Similarity Search

Jin Wang Guoliang Li Dong Deng Yong Zhang Jianhua Feng
Department of Computer Science and Technology, TNList, Tsinghua University, Beijing 100084, China.
{wangjin12,dd11}@mails.tsinghua.edu.cn; {liguoliang,zhangyong05,fengjh}@tsinghua.edu.cn

***Abstract*—String similarity search is a fundamental operation in data cleaning and integration. It has two variants, threshold-based string similarity search and top-k string similarity search. Existing algorithms are efficient either for the former or the latter; most of them can't support both two variants. To address this limitation, we propose a unified framework. We first recursively partition strings into disjoint segments and build a hierarchical segment tree index (HS-Tree) on top of the segments. Then we utilize the HS-Tree to support similarity search. For threshold-based search, we identify appropriate tree nodes based on the threshold to answer the query and devise an efficient algorithm (HS-Search). For top-k search, we identify promising strings with large possibility to be similar to the query, utilize these strings to estimate an upper bound which is used to prune dissimilar strings, and propose an algorithm (HS-Topk). We also develop effective pruning techniques to further improve the performance. Experimental results on real-world datasets show our method achieves high performance on the two problems and significantly outperforms state-of-the-art algorithms.**

I. Introduction

As an important operation in data cleaning and integration, string similarity search has attracted significant attention from the database community. It has a widespread real applications such as web search, spell checking, and DNA sequence discovery in bio-informatics [1], [11]. Given a set of strings and a query, string similarity search aims to find all the strings from the string set that are similar to the query. There are many metrics to quantify the similarity between strings, such as Jaccard, Cosine and edit distance. Among them edit distance is one of the most widely-used metrics to tolerate typographical errors [5], [16], [10], [14]. In this paper we focus on using edit distance to quantify string similarity.

There are two variants of the string similarity search problem. The first is threshold-based string similarity search, which requires users to input a threshold and identifies all the strings from the string set whose edit distances to the query is within the given threshold. The second is top-k string similarity search, which finds k strings from the string set that have the smallest edit distances to the query.

Existing studies on the threshold-based similarity search problem [12], [11], [20], [16] employ a filter-verification framework. In the filter step, they utilize the threshold to devise effective filters in order to prune large numbers of dissimilar strings and generate a set of candidates. In the verification step, they verify the candidates by computing their real edit distances to the query. These threshold-based algorithms are rather expensive for top-k search, because to support top-k search, they have to enumerate the threshold incrementally, execute the search operation for each threshold, and thus involve plenty of unnecessary computations. On the other hand, existing studies on top-k similarity search [26], [27], [5] are inefficient for threshold-based search because they cannot make full use of the given threshold to do pruning. To summarize, existing methods are efficient either for threshold-based search or for top-k search and most of them can't efficiently support both of the two problems. Thus it calls for a unified framework to efficiently support the two variants of string similarity search.

To address this problem, we propose a unified framework which can efficiently support these two variants. We first recursively partition data strings in the string set into disjoint segments and build a hierarchical segment tree index (HS-Tree) on top of the segments. Then we utilize the HS-Tree to answer a threshold-based query or top-k query. For threshold-based similarity search, based on the pigeonhole principle, if a data string is similar to the query, the data string must have enough segments matching some substrings of the query. We can utilize the HS-Tree to identify such strings and devise an efficient algorithm for threshold-based similarity search. For top-k similarity search, we first access the promising data strings that have large possibility to be similar to the query (e.g., the strings sharing largest number of common segments with the query). Based on the promising strings, we can accurately estimate an upper bound of the edit distances of top-k answers to the query and utilize the bound to prune dissimilar strings. We utilize the HS-Tree to identify the promising data strings and devise an efficient algorithm for top-k similarity search. Experimental results on real datasets show our method achieves high performance on both of the two problems and significantly outperforms state-of-the-art algorithms.

In this paper, we make the following contributions.

- We propose a hierarchical segment index HS-Tree which can be utilized to support both threshold-based similarity search and top-k similarity search.
- We devise the HS-Search algorithm based on the HS-Tree index to facilitate threshold-based similarity search. We propose the HS-Topk algorithm based on HS-Tree index to support top-k similarity search.
- We develop batch-based and greedy-match-based pruning strategies to prune dissimilar strings.
- We have conducted an extensive set of experiments. Experimental results show our method achieves high performance on both threshold-based similarity search and top-k similarity search and significantly outperforms state-of-the-art algorithms.

The rest of the paper is organized as follows. We formulate our problem and review related works in Section II. We intro-

duce our hierarchical index in Section III. The HS-Search algorithm is proposed to support threshold-based similarity search in Section IV and the HS-Topk algorithm is presented to support top-k similarity search in Section V.Experimental results are reported in Section VI. We conclude in Section VII.

II. Preliminaries

In this section, we first formulate the problem in Section II-A and then review related works in Section II-B.

A. Problem Definition

Given two strings r and s, the edit distance between r and s, denoted as $\mathsf{ED}(r, s)$, is the minimum number of edit operations (including substitution, insertion, and deletion) needed to transform r to s. There are two variants in string similarity search. The first identifies the strings from a string set whose edit distances to the query are not larger than a given threshold. The second finds top-k strings with the smallest edit distances to the query. Next we formulate the two problems.

Definition 1 (Threshold-based Similarity Search): Given a string set $\mathcal{S}$, a query q, and a threshold τ, threshold-based similarity search finds all strings $s \in \mathcal{S}$ such that $\mathsf{ED}(s, q) \leq \tau$.

Definition 2 (Top-k Similarity Search): Given a string set $\mathcal{S}$, a query q, and an integer k, top-k similarity search finds a subset $\mathcal{R} \subseteq \mathcal{S}$, such that $|\mathcal{R}| = k$ and for any $r \in \mathcal{R}$ and $s \in \mathcal{S} - \mathcal{R}$, $\mathsf{ED}(r, q) \leq \mathsf{ED}(s, q)$.

Example 1: Consider the string set in Table I. Suppose the query q="`brothor`", $\tau = 1$ and $k = 2$. The threshold-based similarity search returns {"`brother`"} since the edit distance between "`brother`" and q="`brothor`" is 1 and the edit distances between other strings and q are larger than 1. The top-2 similarity search returns $\mathcal{R}$={"`brother`", "`brothel`"}, because the edit distances between the two strings and q are respectively 1 and 2, and the edit distances for other strings to the query are not smaller than 2.

TABLE I. A string collection

ID	String	Length
s_1	brother	7
s_2	brothel	7
s_3	broathe	7
s_4	breathes	8
s_5	swingable	9
s_6	deduction	9
s_7	abna levina	11
s_8	christopher swenson	19

B. Related Works

1) Threshold-based Similarity Search: There are many similarity search algorithms [12], [20], [16], [27], [4]. Most of existing studies employ a filter-verification framework to address the string similarity search problem, and many effective filters have been devised to prune dissimilar strings. Sarawagi et al. [17] proposed count filter based on n-grams. Li et al. [11] extended the count filter and developed several list-merge algorithms to improve the performance. Wang et al. [20] proposed an adaptive prefix filtering framework to improve the performance. Zhang et al. [27] proposed B^{ed}-tree which utilized B^+-tree structure to index strings. Li et al. [12] used variable length grams as signatures to support string similarity search. Qin et al. [16] devised an asymmetry signature. Hadjieleftheriou et al. [7] proposed a hash-based method to estimate the number of results.

These methods have two limitations. First, the n-gram-based signature has lower pruning than our segment-based signature, because to avoid missing results, n cannot be large and a small n has limited pruning power. Second, although we can extend such methods to support top-k similarity search by enumerating different thresholds, they are rather expensive as they require to perform the search algorithms many times.

2) Top-k Similarity Search: There have already been some studies on top-k similarity search. Yang et al. [26] proposed an n-gram based method to support top-k similarity search. It dynamically tuned the length of grams according to different thresholds. However, it needed to build duplicate indexes for each n and led to low efficiency. Deng et al. [5] proposed a range-based algorithm by grouping specific entries to avoid duplicated computations in the dynamic-programming matrix when the edit distance is computed. This algorithm used a trie index to share the common prefixes of strings. However for long strings, there are fewer common prefix and the pruning power will be limited. B^{ed}-tree [27] can also be used to support top-k similarity search. However this method utilized n-grams to group "similar" strings together, which needed to enumerate many different thresholds and thus led to low efficiency. Wang et al. [22] designed a novel filter-and-refine pipeline approach that utilized approximate n-gram matchings to compute top-k results.

Compared with these algorithms, our method has two advantages. First, we can utilize the HS-Tree to identify promising strings so as to estimate a tighter upper bound and use the bound to eliminate dissimilar strings. Thus our method achieves higher performance on top-k similarity search. Second, our method can also efficiently support the threshold-based search, because our method can select appropriate nodes in the HS-Tree based on the given threshold and utilize these nodes to efficiently identify the answer.

3) Similarity Join: There are many studies on string similarity join. Given two string sets, similarity join finds all similar string pairs [1], [14], [19], [24], [25], [2], [6]. An experimental survey is made in [9]. Bayardo et al. [1] proposed the prefix filter based method for similarity join. Xiao et al. proposed the position filter [25] and mismatch filter [24] to improve the prefix filter. Wang et al. [19] proposed a trie-based method to support similarity join. Li et al. [14] proposed the segment filter with efficient substring selection and verification methods to perform similarity join. Although Adapt [20] and QChunk [16] extended join techniques to support search, they perform much worse than our method (see Section VI).

In this paper, we extend the segment-based filter in [14] to support similarity search. Different from PassJoin which requires to first specify a threshold and then build the index based on the threshold, we can build an HS-Tree index in an offline step and utilize the index to answer similarity search queries with arbitrary thresholds. For top-k similarity search, since it is rather hard to predefine an appropriate threshold, PassJoin has to enumerate the threshold and thus achieves low performance. Our HS-Tree addresses the limitations of PassJoin and can efficiently support top-k and threshold-based similarity search.

4) Other Related Works: Some previous studies focus on approximate entity extraction, which gives a dictionary of entities and a document, finds all substrings in the document that are similar to some entities. Wang et al. [21] proposed

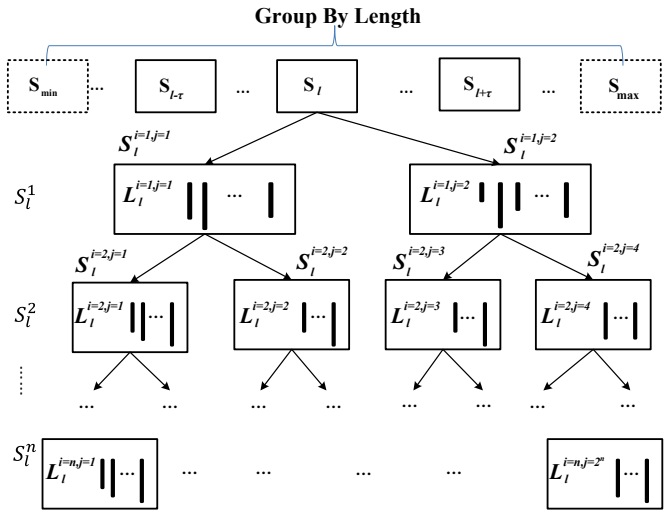


Fig. 1. The HS-Tree Index.

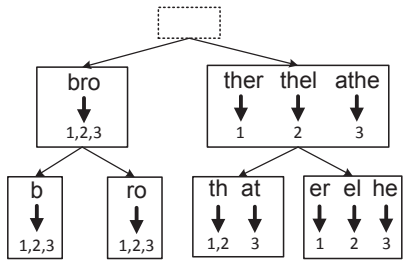


Fig. 2. An HS-Tree example for $\mathcal{S}_7$.

a neighborhood deletion based method to solve this problem. Li et al. [13] proposed a unified framework to support approximate entity extraction under various similarity functions. Recently Kim et al. [10] proposed efficient algorithms to solve the problem of substring top-k similarity search, which finds the top-k approximate substrings matching with a given query, which is different from our top-k similarity search problem. Another related topic is query autocompletion. Ji et al. [8] proposed a trie-based structure to support fuzzy prefix search and Li et al. [15] improved it by removing unnecessary nodes. Chaudhuri et al. [3] proposed a similar solution. Xiao et al. [23] extended the neighborhood deletion method to improve the performance of query autocompletion.

III. The Hierarchical Segment Index

In this section, we introduce a hierarchical segment tree (HS-Tree) to index the data strings. We first group the strings by length and let $\mathcal{S}_l$ denote the group of strings with length l. For each group $\mathcal{S}_l$, we build a complete binary tree, where the root is a dummy node. We partition each string $s \in \mathcal{S}_l$ into two disjoint segments where the first segment is the prefix of s with length $\lfloor \frac{|s|}{2} \rfloor$ and the second segment is the suffix of s with length $\lceil \frac{|s|}{2} \rceil$, where $|s|$ is the length of s. Let $\mathcal{S}_l^{1,1}$ and $\mathcal{S}_l^{1,2}$ respectively denote the set of the first segments and that of the second segments for the strings in $\mathcal{S}_l$. Based on these two sets, we generate two children of the root, i.e., two nodes in the first level, $n_l^{i=1,j=1}$ and $n_l^{i=1,j=2}$, where i denotes the level and j denotes the sibling. (The level of the root is 0.) For each tree node $n_l^{i,j}$, we build an inverted index, where entries are segments in $\mathcal{S}_l^{i,j}$ and each segment with segment id j is associated with an inverted list $\mathcal{L}_l^{i,j}$ which is a list of strings that contain the segment.

Algorithm 1: HS-Tree Construction ($\mathcal{S}$)

Input: $\mathcal{S}$: The string set
Output: The HS-Tree index

1 **begin**
2 Group strings in $\mathcal{S}$ by length;
3 **for** $l = l_{min}$ *to* l_{max} **do**
4 Calculate the maximum level $L = \lfloor \log_2 l \rfloor$;
5 Generate sets $\mathcal{S}_l^{1,1}$, $\mathcal{S}_l^{1,2}$, nodes $n_l^{1,1}$, $n_l^{1,2}$, indexes $\mathcal{L}_l^{1,1}$, $\mathcal{L}_l^{1,2}$;
6 **for** $i = 2$ *to* n **do**
7 **for** $j = 1$ *to* 2^i **do**
8 Generate sets $\mathcal{S}_l^{i,2j-1}$, $\mathcal{S}_l^{i,2j}$, nodes $n_l^{i,2j-1}$, $n_l^{i,2j}$, indexes $\mathcal{L}_l^{i,2j-1}$, $\mathcal{L}_l^{i,2j}$;
9 **end**

Next we recursively construct the tree. For each node $n_l^{i,j}$, we partition each segment in $\mathcal{S}_l^{i,j}$ into two segments (using the same partition method above) and let $\mathcal{S}_l^{i+1,2j-1}$ and $\mathcal{S}_l^{i+1,2j}$ respectively denote the set of the first segments and that of the second segments. Then node $n_l^{i,j}$ has two children $n_l^{i+1,2j-1}$ and $n_l^{i+1,2j}$ with respect to $\mathcal{S}_l^{i+1,2j-1}$ and $\mathcal{S}_l^{i+1,2j}$. We also build the inverted indexes $\mathcal{L}_l^{i+1,2j-1}$ and $\mathcal{L}_l^{i+1,2j}$. The procedure is terminated if there exists a segment in the level with length of 1. In other words, the maximum level is $\lfloor \log_2 l \rfloor$. Figure 1 illustrates the index structure.

Algorithm 1 shows the algorithm to build the HS-Tree index. It first groups strings by length (line 2). For each group $\mathcal{S}_l$, it builds a hierarchical index with L levels, where $L = \lfloor \log_2 l \rfloor$ (line 4). In level i, it iteratively partitions the segments in level $i-1$ into 2 segments and constructs invert indexes $\mathcal{L}_l^{i,j}$ for the jth segment in level i (lines 6-8).

Example 2: Consider the string set in Table I. We first group them by length and then iteratively build HS-Tree for each length l. Take group $\mathcal{S}_7$ in Figure 2 as an example. The group length is 7. The maximal level is $L = \lfloor \log_2 7 \rfloor = 2$. Consider string s_1 ="`brother`". In level 1, s_1 is partitioned into two segments {"`bro`","`ther`"}. Then in level 2, these two segments are iteratively partitioned into 2 segments {"`b`","`ro`"}, and {"`th`","`er`"}. Similarly, we can iteratively partition strings s_2 and s_3 to build the index.

Space Complexity: We analyze the space complexity of the HS-Tree. Suppose l_{min} is the minimum string length and l_{max} is the maximum string length. For each group $\mathcal{S}_l$, the HS-Tree index includes segments and the inverted index. $\mathcal{S}_l$ contains $\lfloor \log l \rfloor$ levels. In the ith level, there are 2^i segments. Thus each string is partitioned into at most $\sum_{j=1}^{\log l} 2^j = \mathcal{O}(l)$ segments. Obviously each string is contained in at most $\mathcal{O}(l)$ inverted lists, and thus the space complexity of the HS-Tree is $\mathcal{O}(\sum_{l=l_{min}}^{l=l_{max}} l * |\mathcal{S}_l|)$, which means the total number of characters in all input strings.

Discussion: For ease of presentation, in the HS-Tree, we assume each node has two children and our method can be easily extended to support the case that each node has more than two children, like B-tree. In addition, we focus on in-memory setting and our method can be extended to support disk-based setting (like B-tree). Due to space constraints, we leave it as a future work.

IV. Threshold-based Similarity Search

In this section, we devise an efficient algorithm HS-Search to efficiently answer threshold-based similarity search queries using the HS-Tree index (see Section IV-A). We first introduce a filter-verification framework and then develop novel techniques to improve both the filter step (see Section IV-B) and the verification step (see Section IV-C).

A. The HS-Search Algorithm

Consider a query q with a threshold τ. Based on the length filter: two strings cannot be similar if their length difference is larger than τ, we only need to access the HS-Tree with lengths between $|q| - \tau$ and $|q| + \tau$. Consider the HS-Tree with length $l \in [|q| - \tau, |q| + \tau]$. In the ith level, the strings are partitioned into 2^i segments. For $2^i \geq \tau + 1$, any string s in $\mathcal{S}_l$ cannot be similar to q if q has no substring matching a segment of s based on the pigeonhole principle. Moreover, if s is similar to q, q must contain at least $2^i - \tau$ segments of s, as formalized in Lemma 1. On the contrary, if $2^i < \tau + 1$, any string in $\mathcal{S}_l$ may be similar to q even if q has no substring matching a segment of the string.

Lemma 1: Given a string s with 2^i segments, a query q and a threshold τ, if s is similar to q, q must contain at least $2^i - \tau$ segments of s.

Example 3: Consider a query q={"swaingbe"}, the data string s_5 = {swingable}, s_6 = {deduction}, and $\tau = 2$. In level 2, the 4 segments of s_5 are {"sw","in","ga","ble"}, and q has $2^2 - 2 = 2$ common substrings "sw" and "in" with s_5, thus s_5 is a candidate for query q and $\tau = 2$. On the contrary, the 4 segments of s_6 = {deduction} are {"de","du","ct","ion" }. And s_6 has no matched segments with q. So we can safely prune s_6.

Generally, consider the level $i \geq \log_2(\tau+1)$. If a string has less than $2^i - \tau$ segments that match substrings of the query, we can prune it. In other words, we only need to check the candidate strings which share at least $2^i - \tau$ common segments with q. To facilitate the checking, we can utilize the inverted index on each node to identify the candidates, which will be discussed in detail later. As there are many level i such that $i \geq \log_2(\tau + 1)$, we can use any of such levels to identify candidates. Obviously the deeper the level is, the shorter the segment is, and thus the lower pruning power is. Thus we use the minimal level $\lceil \log_2(\tau + 1) \rceil$ with longest segments.

Formally, consider a query q and group $\mathcal{S}_l$. We first locate the $i = \lceil \log_2(\tau + 1) \rceil^{th}$ level. For each node $n_l^{i,j}$ $(1 \leq j \leq 2^i)$, we compute the length of segments in this node $\mathtt{Len}_l^{i,j}$. (It is worth noting that the segments in each node have the same length.) To check whether q has a substring matching a segment in node $n_l^{i,j}$, we only need to enumerate the set of substrings of q with length $\mathtt{Len}_l^{i,j}$, denote as $\mathcal{W}(q, \mathcal{L}_l^{i,j})$. We will discuss how to reduce the size of $\mathcal{W}(q, \mathcal{L}_l^{i,j})$ in Section IV-B. Next we find the strings which have at least $2^i - \tau$ segments matching the query. To this end, for each substring in $\mathcal{W}(q, \mathcal{L}_l^{i,j})$, we identify the substring from the inverted index of the node and retrieve the inverted list of the substring. Next we compute the strings that appear more than $2^i - \tau$ times on the invited lists. Such strings will be regarded as candidates and then we will verify them and get the results.

Algorithm 2: HS-Search $(\mathcal{S}, q, \tau)$

Input: $\mathcal{S}$: The string set
q: The query string
τ: The given edit-distance threshold
Output: $\mathcal{R} = \{(s \in \mathcal{S}) \mid \mathrm{ED}(s, q) \leq \tau\}$

```
1  begin
2      Calculate the maximum level i = ⌈log_2 (τ + 1)⌉;
3      for l = |q| − τ to |q| + τ do
4          for j = 1 to 2^i do
5              Generate substrings set W(q, L_l^{i,j}) ;
6              for w ∈ W(q, L_l^{i,j}) do
7                  for s ∈ L_l^{i,j}[w] do
8                      N_i(s, q) = N_i(s, q) + 1;
9                      if N_i(s, q) ≥ 2^i − τ then
10                         if ED(s, q) ≤ τ then
11                             R = R ∪ {s};
12 end
```

Based on Lemma 1, we devise the HS-Search algorithm to support threshold-based string similarity search and the pseudo-code is shown in Algorithm 2. HS-Search first calculates the level $i = \lceil \log_2(\tau + 1) \rceil$. Then HS-Search utilizes length filter to reduce the number of visited HS-Tree: for a query string q and threshold τ, only groups $\mathcal{S}_l(|q| - \tau \leq l \leq |q| + \tau)$ are visited. Next HS-Search generates the set of substrings of q, $\mathcal{W}(q, \mathcal{L}_l^{i,j})$, for j = 1 to 2^i (line 5). According to Lemma 1, if a string s is similar to q, s should appear at least $2^i - \tau$ times in the inverted lists $\mathcal{L}_l^{i,j}[w]$ for $1 \leq j \leq 2^i$ and $w \in \mathcal{W}(q, \mathcal{L}_l^{i,j})$. To this end, HS-Search checks whether $w \in \mathcal{W}(q, \mathcal{L}_l^{i,j})$ is in $\mathcal{L}_l^{i,j}$. If so, for any string s in the inverted list $\mathcal{L}_l^{i,j}[w]$, s and q shares a common segment w and we increase $\mathcal{N}_i(s, q)$ by 1, where $\mathcal{N}_i(s, q)$ denotes the number of matched segments between s and q in level i(line 8). If $\mathcal{N}_i(s, q)$ exceeds $2^i - \tau$, s is a candidate and HS-Search verifies candidate s (line 11). To improve the performance of HS-Search, we design efficient techniques to reduce the substring-set size $\mathcal{W}(q, \mathcal{L}_l^{i,j})$ in Section IV-B and improve the verification cost in Section IV-C.

Example 4: Consider the string set in Table I. Suppose we have built the HS-Tree index as shown in Figure 2. Assume the query string is q ="brethor" and the threshold is $\tau = 2$. First, $i = \lceil \log_2(\tau + 1) \rceil = 2$. In level 2 there are 4 segments. If $\mathrm{ED}(s, q) \leq 2$, s and q must have at least $2^2 - 2 = 2$ common segments. As $|q| = 7$ and $l_{min} = 7$, we only need to visit the strings with length between 7 and 9 for $\tau = 2$. First we check $\mathcal{L}_7^{2,1}$, which contains segments {"b"}. String q has a matched substring "b" in $\mathcal{L}_7^{2,1}$. As strings s_1, s_2 and s_3 in the inverted list of "b" share a common segment with q, we increase $\mathcal{N}_2(s_1, q)$, $\mathcal{N}_2(s_2, q)$, $\mathcal{N}_2(s_3, q)$ by 1. Then we check $\mathcal{L}_7^{2,2}$, which contains segments {"ro"}, string q has no matched substring. Similarly we check $\mathcal{L}_7^{2,3}$ and $\mathcal{L}_7^{2,4}$. As q has a matched substring "th" in $\mathcal{L}_7^{2,3}$ with invited list of $\{s_1, s_2\}$, we increase $\mathcal{N}_2(s_1, q)$ and $\mathcal{N}_2(s_2, q)$ by 1. Now strings s_1 and s_2 have two matched segments with q, so we verify them and get $\mathrm{ED}(q, s_1) = 2$ and $\mathrm{ED}(q, s_1) = 3$. We put s_1 into the result set. As $\mathcal{N}_2(s_3, q) = 1 < 2$, we can safely prune s_3. Then we perform similar procedure on $\mathcal{L}_8^{i,j}$ and $\mathcal{L}_9^{i,j}$, no more answers are found and the algorithm terminates.

Complexity: We analyze the time complexity of Algorithm 2. It consists of two parts: the filtering time and the verification time. Before performing Algorithm 2, we need to group strings by length, which is $\mathcal{O}(\sum_{l=l_{min}}^{l_{max}} |\mathcal{S}_l|)$. This can be regarded as offline time and not included in the search time. The time to generate set $\mathcal{W}(q, \mathcal{L}_l^{i,j})$ is $\mathcal{O}(\tau)$ as discussed in Section IV-B. The total time of selecting substrings for $l \in [|q|-\tau, |q|+\tau$ and $j \in [1, 2^i]$ is $\mathcal{O}(l\tau^2)$. The filtering cost is to visit the inverted lists which is $\sum_{l\in[|q|-\tau,|q|+\tau]} |\mathcal{L}_l^{i,j}[w \in \mathcal{W}(q, \mathcal{L}_l^{i,j})]|$. The verification cost of q and s for threshold τ is $\mathcal{O}(\tau * \min(|q|, |s|))$ [14], and we will further improve the verification cost in Section IV-C.

B. Improving The Filtering Step

To find the candidate of a given query q, we need to first generate the set of substrings $\mathcal{W}(q, \mathcal{L}_l^{i,j})$ and then count how many common segments between strings in $\mathcal{S}_l$ and q. To reach such goal efficiently, we reduce the filtering cost through two directions: (1) reduce the size of $\mathcal{W}(q, \mathcal{L}_l^{i,j})$ for $1 \le j \le 2^i$; and (2) remove invalid segment matchings.

Reduce $\mathcal{W}(q, \mathcal{L}_l^{i,j})$. It is obvious smaller $\mathcal{W}(q, \mathcal{L}_l^{n,j})$ will lead to higher performance. Based on the position filter, a segment in s cannot be matched with the substrings of q with large position difference. For example, for s_5="swingable", query q="blending" and threshold $\tau = 2$. The segments of s in the second level is correspondingly {"sw","in","ga","ble"}. The substring "ble" cannot be matched with the fourth segment because their position difference is larger than τ. The position filter could be strengthened by considering the length difference of s and q, denoted as Δ. Thus suppose the start position of segment $w \in \mathcal{L}_l^{i,j}$ is $\texttt{Pos}_l^{i,j}$ and its length is $\texttt{Len}_l^{i,j}$. We can easily get the lower bound of start positions of substrings in q, denoted as $\mathsf{LB} = \max(1, \texttt{Pos}_l^{i,j} - \lfloor \tau - \Delta \rfloor)$, and the upper bound, denoted as $\mathsf{UB} = \min(|q| - \texttt{Len}_l^{i,j} + 1, \texttt{Pos}_l^{i,j} + \lfloor \tau + \Delta \rfloor)$. We only need to check the substrings starting within the range $[\mathsf{LB}, \mathsf{UB}]$. Moreover, by looking both from the left-side and right-side perspective [14], we can further reduce the value of LB and UB to $\mathsf{LB} = \max(1, \texttt{Pos}_l^{i,j} - (j-1), \texttt{Pos}_l^{i,j} + \Delta - (\tau + 1 - j))$ and $\mathsf{UB} = \min(|s| - \texttt{Len}_l^{i,j} + 1, \texttt{Pos}_l^{i,j} + (j-1), \texttt{Pos}_l^{i,j} + \Delta + (\tau + 1 - j))$. Thus $\mathcal{W}(q, \mathcal{L}_l^{i,j}) = \{q[\texttt{Pos}_l^{i,j}, \texttt{Len}_l^{i,j}]\}$ where $q[\texttt{Pos}_l^{i,j}, \texttt{Len}_l^{i,j}]$ is the substring of q with start position $\texttt{Pos}_l^{i,j} \in [\mathsf{LB}, \mathsf{UB}]$ and length $\texttt{Len}_l^{i,j}$. The correctness is stated in Lemma 2.

Lemma 2: Given a query string q and a threshold τ, using the set $\mathcal{W}(q, \mathcal{L}_l^{i,j}) = \{q[\texttt{Pos}_l^{i,j} \in [\mathsf{LB}, \mathsf{UB}], \texttt{Len}_l^{i,j}]\}$ to find matching candidates, our method will not miss any result.

To identify string s with $\mathcal{N}_i(s, q) \ge 2^i - \tau$, we use the list-merge algorithm [11] to improve the performance which utilizes a heap to efficiently identify the candidates without accessing every strings on the inverted lists.

Remove Invalid Matchings. The above method identifies the candidates by simply counting the number of matched common segments. However it is worth noting that the substrings of q matching with different segments may conflict with each other, where *conflict* means that the two matched substrings overlap. This is because the segments of the data strings are disjoint and the matched substrings of q should also be disjoint. For example, if q="acompany", s="accomplish" and threshold $\tau = 2$. If the substring "ac" of q matches the first segment "ac", the substring "com" cannot match the second segment, because "ac" and "com" are overlapped in q but they are disjoint in s. If we do not eliminate such conflict matching, it will involve false positives. For example, we get $\mathcal{N}_2(s, q) = 2$ in the segment count step, but string s has only one common segment with string q. This false positive will result in larger candidate size and extra verification cost.

Algorithm 3: SEGCOUNT()

Input: $\mathcal{M}_i(s, q)$: Matched segments of s, q in level i.
Output: Number of matched segments without conflict.

```
1 begin
2     D[1] = 1;
3     for j = 2 to N_i(s, q) do
4         D[j] = max_{1<t≤j-1}{γ(j, t) · D[t]} + 1;
5     return D[N_i(s, q)];
6 end
```

To solve this problem, we design a dynamic-programming algorithm to calculate the maximum number of matched segments while eliminating the conflict between matched segments (removing overlapped matching). Let $D[j]$ denote the maximum number of matched segments without conflict among the first j segments. To calculate $D[j]$, we need to find the last matched segment t without conflict for $t < j$, and compute the number of matched segments without conflict using this segment. Then we consider whether the current matched segment conflicts with the last matched segment. We use a function $\gamma(j, t)$ to judge whether two matches j and t conflict: $\gamma(j, t) = 1$ if there is no conflict; $\gamma(j, t) = 0$ otherwise. Then we can get the following recursion formula:

$$D[j] = \begin{cases} 1, & j = 1 \\ max_{1<t\le j-1}\{\gamma(j,t) \cdot D[t]\} + 1, & \text{otherwise} \end{cases} \quad (1)$$

Then we devise a dynamic-programming algorithm based on the above formula as shown in Algorithm 3. The algorithm takes as input the set of matched segments between s and q in level i, denoted as $\mathcal{M}_i(s, q)$, which can be easily gotten when computing $\mathcal{N}_i(s, q)$. The algorithm outputs the number of matched segments without conflict based on Equation 1. The time complexity of this algorithm is $\mathcal{O}(x^2)$, where $x = \mathcal{N}_i(s, q)$. The maximum value of x is 2^i in level i, but in practical cases the number of matched segments is far smaller than 2^i with the help of efficient substring selection methods. Thus the cost of this algorithm is negligible. We can integrate this observation into Algorithm 2 (replace line 8 of Algorithm 2 with Algorithm 3) to enhance the pruning power.

Example 5: Suppose string s ="are accommodate to", q ="were acomofortable" and $\tau = 5$. In the segment matching step, we search in level 3 and get $\mathcal{M}_3(s, q) =$ { "ac","com","mo" } and $\mathcal{N}_3(s, q) = 3 \ge 2^3 - 5$. However, when we perform Algorithm 3 on $\mathcal{M}_3(s, q)$, we get $D[1] = 1$, $D[2] = 1$ and $D[3] = 2$. So there are 2 rather than 3 matched segments. As $2 < 2^3 - 5 = 3$, we can safely prune s.

C. Improving The Verification Step

In our HS-Search algorithm, after generating the candidate strings, we verify whether their real edit distances to the

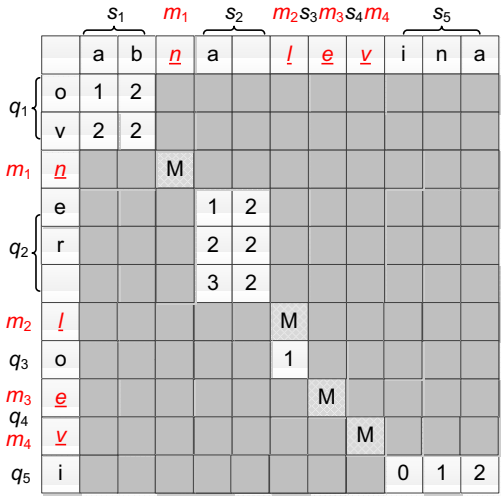


Fig. 3. The MultiExtension Verification ($y = 4$)

query are within the threshold τ. In this section we devise novel techniques to improve the verification step.

To compute the edit distance between two strings q and s, a naive method is to use the dynamic-programming algorithm. Given two strings q and s, it utilizes a matrix C with $|q|+1$ rows and $|s|+1$ columns where $C[i][j]$ is the edit distance between the substring $q[1,i]$ and $s[1,j]$ [18]. Actually, if we only want to check whether the edit distance between two strings is within a given threshold τ, we can further reduce the complexity by only computing the $C[i][j]$ values for $|i-j| \leq \tau$, with the cost of $(2\tau+1)\cdot\min(|q|,|s|)$. A length-aware method has been proposed to improve the time complexity to $\tau\cdot\min(|q|,|s|)$ [14]. These algorithms can also do early termination when the values in a row are all larger than τ to further improve the time complexity.

For our HS-Search algorithm, we can utilize the set of matched segments in $\mathcal{M}_i(s,q)$ to avoid unnecessary computations. As we can see from Section IV-A, given a candidate string s, $y = \mathcal{N}_i(s,q)$. We align q and s based on these matched segments and partition them into $2\cdot y+1$ parts including y matched segments and $y+1$ unmatched segments. We only need to verify whether $\mathsf{ED}(q,s) \leq \tau$ in this alignment. Suppose the set of matched segments is $\mathcal{M}_i(s,q) = \{m_1, m_2, \cdots, m_y\}$, and the set of unmatched segments of query q (string s) is $Q = \{q_1, q_2, \cdots, q_{y+1}\}$ ($S = \{s_1, s_2, \cdots, s_{y+1}\}$), where $q_j(s_j)$ is the substring between m_i and m_i+1 for $j \in [1,y]$ and $q_{y+1}(s_{y+1})$ is the substring after p_y (s_y). If two matched segments m_j and m_{j+1} are consecutive, $s_j(q_j)$ =“”. We denote the total edit distance as $\mathsf{TED} = \sum_{i=1}^{y+1}\mathsf{ED}(s_i,q_i)$. If TED is larger than τ, s is not similar to q in this alignment and we can discard s; otherwise we add s into the result set. We call this method as SingleThreshold.

We can further improve the performance of SingleThreshold by assigning each $\mathsf{ED}(q_j,s_j)$ with a tighter threshold bound. As we can see, the part s_j is between the segments m_j and m_{j+1}. Let p_j denote the order of m_j among the 2^i segments in s. For string s, s_j consists of $p_{j+1}-p_j-1$ segments (for $j=1$, the value is p_1-1 and for $j=y+1$, the value is 2^i-p_y). For a given threshold τ, if y is exactly $2^i-\tau$, the τ edit operations must be distributed in each segment according to the pigeon hole principle. In this case, if we find

Algorithm 4: MultiExtension $(s,q,\tau,\mathcal{M}_i(s,q))$

Input: s,q: the strings to be verified
τ: the given threshold
$\mathcal{M}_i(s,q)$: matched segments of s,q in level i.
Output: The verification result

```
begin
    Generate sets S and Q based on M_i(s,q);
    Generate thresholds based on M_i(s,q);
    for j = 1 to N_i(s,q) + 1 do
        Calculate ED(q_j, s_j) with length-aware method;
        if ED(q_j, s_j) > τ_j then return;
    Add s into the result set;
end
```

two consecutive errors in one segment (or in other words, more than τ_j errors appear in part j), we can safely terminate the verification step. The threshold τ_j of each part j is calculated as follows:

$$\tau_j = \begin{cases} p_1 - 1, & j = 1 \\ 2^i - p_y, & j = y+1 \\ p_{j+1} - p_j - 1, & \text{otherwise} \end{cases} \quad (2)$$

We propose an early termination technique (Lemma 3).

Lemma 3: Consider two strings s and q with exactly $y = 2^i - \tau$ common segments. If $\mathsf{ED}(q_j,s_j) > \tau_j$, $\mathsf{ED}(q,s) > \tau$.

Based on Lemma 3, we can devise the verification algorithm MultiExtension as shown in Algorithm 4. In this algorithm, we can use the length-aware method to efficiently verify $\mathsf{ED}(q_j,s_j)$. Next we walk through the two verification algorithms SingleThreshold and MultiExtension using the following example as shown in Figure 3.

Example 6: Consider two strings s_7=“abna levina” in Table II and q=“ovner loevi”, and the threshold is 5. We perform HS-Search on level 3 with 8 segments. The segments of s on level 3 are respectively {“a”,“b”,“n”,“a”,“l”, “ev”,“i”,“na”}. As we can see from Figure 3, s and q have matched segments m_1 = “n”,m_2 = “l”,m_3 = “ev”. $p_1 = 3$, $p_2 = 5$, $p_3 = 6$. Then we can generate the set of S={s_1=“ab”,s_2=“a”,s_3=“”,s_4=“ina”} and Q={q_1=“ov”,q_2=“er”,q_3=“o”,q_4=“i”}. For MultiExtension, the thresholds of each part are respectively 2,1,0,2. Then we calculate $\mathsf{ED}(s_1,q_1) = 2 \leq 2$, $\mathsf{ED}(s_2,q_2) = 2 > 1$, then MultiExtension terminates and discards s. But SingleThreshold will continue to calculate $\mathsf{ED}(s_3,q_3) = 1$, $\mathsf{ED}(s_4,q_4) = 2$ and $\mathsf{TED} = 2+2+1+2 = 7 > 6$, then it discards s. Thus MultiExtension outperforms SingleThreshold.

V. Top-k Similarity Search

In this section, we study the top-k similarity search problem. Different from the threshold-based similarity search, the top-k similarity search has no fixed threshold. Although we can extend the threshold-based method to find top-k answers by enumerating thresholds incrementally until k results found, this algorithm is rather expensive because it executes multiple (unnecessary) search operations for each threshold and involves many duplicated computations. To address this issue, we devise an efficient algorithm HS-Topk to support top-k similarity search using our HS-Tree index. The basic idea is to first access the promising strings with large possibility

to be similar to the query so as to prune large numbers of dissimilar strings effectively. To this end, we first propose a batch-pruning-based method in Section V-A and then present an effective pruning strategy in Section V-B. Finally we devise our HS-Topk algorithm in Section V-C.

A. The Batch-Pruning-Based Method

We maintain a priority queue $\mathcal{Q}$ to keep the current k promising results. Let $\mathsf{UB}_{\mathcal{Q}}$ denote the largest edit distance between the strings in $\mathcal{Q}$ to the query, i.e., $\mathsf{UB}_{\mathcal{Q}} = \max\{\mathsf{ED}(s \in \mathcal{Q}, q)\}$. Obviously $\mathsf{UB}_{\mathcal{Q}}$ is an upper bound of the edit distances of top-k results to the query. In other words, we can prune a string if its edit distance to the query is larger than $\mathsf{UB}_{\mathcal{Q}}$. Next we discuss how to utilize the queue to find top-k answers.

Given a query q, we still access the HS-Tree in a top-down manner. Consider the ith level of the HS-Tree with length l. For each node $n_l^{i,j}$, we generate the substrings $\mathcal{W}(q, \mathcal{L}_l^{i,j})$ for $1 \leq j \leq 2^i$ and identify the corresponding inverted lists $\mathcal{L}_l^{i,j}$. We group the strings in the inverted lists based on the number of substrings they contain in $\mathcal{W}(q, \mathcal{L}_l^{i,j})$. Let $\mathcal{B}_x$ denote the group of strings containing x substrings. As each string contains at most 2^i segments, there are at most 2^i groups. For strings in $\mathcal{B}_x$, they share x common segments with the query. If $2^i - 1 < \mathsf{UB}_{\mathcal{Q}}$, for $x \in [1, 2^i - 1]$ all strings in $\mathcal{B}_x$ can be regarded as candidates. Otherwise, there are $2^i - x$ mismatch segments for strings in $\mathcal{B}_x$. As a mismatch segment leads to at least 1 edit error, and thus the lower bound of the edit distances of strings in $\mathcal{B}_x$ to query q is $\mathsf{LB}_{\mathcal{B}_x} = 2^i - x$. Obviously if $\mathsf{LB}_{\mathcal{B}_x} \geq \mathsf{UB}_{\mathcal{Q}}$, we can prune the strings in $\mathcal{B}_x$ based on Lemma 4. In other words, we only need to visit the groups such that $x \geq 2^i - \mathsf{UB}_{\mathcal{Q}}$. On the other hand, the larger x is, the strings in $\mathcal{B}_x$ have larger possibility in the top-k answers. Thus, we want to first access the strings in groups with larger x. These two observations motivate us devise a batch-pruning-based method.

Lemma 4: If $\mathsf{LB}_{\mathcal{B}_x} \geq \mathsf{UB}_{\mathcal{Q}}$, strings in $\mathcal{B}_x$ can be pruned.

For level i, if $2^{i-1} \geq \mathsf{UB}_{\mathcal{Q}} + 1$, we can terminate because we have found all top-k answers within threshold $\mathsf{UB}_{\mathcal{Q}}$ using the nodes in the first $i-1$ levels; otherwise, we retrieve the inverted lists of substrings in $\mathcal{W}(q, \mathcal{L}_l^{i,j})$ from the ith level and identify the substrings with $\mathcal{N}_i(s, q) \geq 2^i - \mathsf{UB}_{\mathcal{Q}}$ from these lists. Next we group the strings into $\mathcal{B}_x$ ($x \in [2^i - \mathsf{UB}_{\mathcal{Q}}, 2^i]$) based on the number of matched segments. Then, we visit the groups based on the number x in descending order. For each string $s \in \mathcal{B}_x$, we compute the real edit distance between s and q. If $\mathsf{ED}(s, q) < \mathsf{UB}_{\mathcal{Q}}$, we update the priority queue $\mathcal{Q}$ and $\mathsf{UB}_{\mathcal{Q}}$ using s. Iteratively, we can correctly find the top-k answers.

Obviously this batch-pruning-based method not only reduces the filtering cost, because we only need to do segment counting once for a level i while we need to perform $2^i - 1$ times for each threshold using the threshold-based method, but also the verification time, because we can use a tighter bound $\mathsf{UB}_{\mathcal{Q}}$ to do verification.

Example 7: Consider a top-2 query q = "`breahers`" on the dataset in Table I. Suppose string s_4 = "`breathes`" is already in $\mathcal{Q}$ as it has a common segment "`brea`" in level 1 with q. $\mathsf{ED}(q, s_4) = 2$. $\mathsf{UB}_{\mathcal{Q}} = \infty$. Then, consider

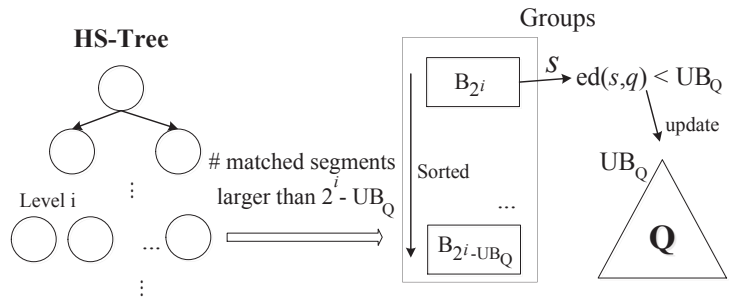


Fig. 4. The Batch-Pruning-Based Method

the HS-Tree for strings with length 7 ($\mathcal{S}_7$) in Figure 2. We start from level 1. As there is no matched segment, we move to level 2. There are two matched segments "`br`, `er`". Their inverted lists are $\{s_1\}$ and $\{s_1, s_2, s_3\}$ where s_1 = "`brother`", s_2 = "`brothel`", s_3 = "`breathe`". we have $\mathcal{N}_2(s_1, q) = 2, \mathcal{N}_2(s_2, q) = 1$ and $\mathcal{N}_2(s_3, q) = 1$. We put s_1 into $\mathcal{B}_2$ and verify $\mathcal{B}_2$. As $\mathsf{ED}(q, s_1)$=3 $< \mathsf{UB}_{\mathcal{Q}}$, we add s_1 into $\mathcal{Q}$ and update $\mathsf{UB}_{\mathcal{Q}} = 3$. As $2^2 \geq \mathsf{UB}_{\mathcal{Q}} + 1$, the algorithm is terminated and strings in $\mathcal{B}_1 = \{s_2, s_3\}$ are pruned.

B. The Greedy-Match Strategy

The batch-pruning-based method can effectively prune strings without enough common segments. If each mismatch segment only contains one edit error, this method is very effective as it can effectively estimate the lower bound. However, if one mismatch segment involves more than one consecutive errors, the estimation is not accurate, and this method fails to filter such candidates. For example, consider query q ="`broader`" on the dataset in Table I. For $\mathsf{UB}_{\mathcal{Q}} = 1$, string s_1 ="`brother`" and s_2 ="`brothel`" can pass the segment filter as they share a common segment "`bro`", but it is obvious that their edit distances to q are larger than 1 as the second segment contains 3 errors. To address this issue, we devise a *greedy-match strategy* to prune strings with consecutive errors by utilizing our hierarchical index structure.

Consider a string s in level i with $\mathcal{N}_i(s, q) \geq 2^i - \mathsf{UB}_{\mathcal{Q}}$. In this case, we cannot prune s. Instead of directly verifying string s, we go to the next level $i+1$ and estimate a tighter bound by counting the number of matched segments in level $i+1$ (i.e., $\mathcal{N}_{i+1}(s, q)$). If the number is smaller than $2^{i+1} - \mathsf{UB}_{\mathcal{Q}}$, we can prune string s based on Lemma 4. If the string is not pruned in level $i+1$. We check the level $i+2$. Iteratively, if the string is still not pruned in the leaf level, we will compute the real edit distance based on the method in Section IV-C. It is worth noting that the larger the level is, the shorter a segment is, and the higher probability that those dissimilar strings with consecutive errors can be pruned.

Next we discuss how to efficiently compute the number of matched segments between s and q. A naive method enumerates each segment of s and checks whether it appears as a substring of q. This method should enumerate many segments. Alternatively, we propose an effective method. Based on the characteristics of the HS-Tree, if the jth segment in level i matches a substring of q, the $2 * j - 1^{th}$ and $2 * j^{th}$ segments must match two substrings of q in level $i+1$. Thus we do not need to check them again. Thus, we only need to check the mismatch segments in level i.

Algorithm 5: GREEDYMATCH (s, q, i, τ)

Input: s, q: the strings to be verified
i: level; τ: the current threshold
$\mathcal{M}_i(s, q)$: matched segments of s, q in level i.
Output: True or False

```
1  begin
2      for r = i + 1 to n do
3          for segments w ∈ M_{r-1}(s, q) do
4              put w's two subsegments into M_r(s, q);
5          for j = 1 to 2^r do
6              if segment j ∉ M_r(s, q) then
7                  check the segment j;
8                  if find a matched substring then
9                      put segment j into M_r(s, q);
10         if N_r(s, q) < 2^r − τ then
11             return False;
12     return True;
13 end
```

The pseudo-code of the greedy-match strategy is shown in Algorithm 5. It gets the set of matched segments $\mathcal{M}_i(s, q)$ in current level i and then use $\mathcal{M}_i(s, q)$ to generate $\mathcal{M}_{i+1}(s, q)$. This iteratively-matching procedure for different levels will not involve heavy filtering cost, because segment j in level i corresponds to segments $2*j-1$ and $2*j$ in level $i+1$ and such matched segments can be passed down to lower levels. Besides, as we have generated the substrings $\mathcal{W}(q, \mathcal{L}^{r,j}_{|s|})$ for each level r $(1 \leq r \leq n)$, when we look for a matched substring for segment j, we just check $\mathcal{W}(q, \mathcal{L}^{r,j}_{|s|})$ and do not need to scan inverted lists any more.

Example 8: Consider s_8="`christopher swenson`" and query q="`atrmstophbwcmrense`". The length of s_8 is 19, so there are 4 levels. Suppose the current threshold $\mathsf{UB}_{\mathcal{Q}}$ is 3. It requires $2^2 - 3 = 1$ matched segments in level 2, $2^3 - 3 = 5$ segments in level 3 and $2^4 - 3 = 13$ segments in level 4. We find a matched segments "`stoph`" in level 2. Instead of verification, here we continue to look for another $5 - 1 * 2 = 3$ matched segment in level 3. We find a matched segment "`en`" in level 3. As there are totally 3 matched segments in level 3, which is smaller than the required number of matched segment 5, s_8 will be pruned and we do not need to compute its real edit distance to q.

C. The HS-Topk Algorithm

We combine the batch-pruning-based method and greedy-match strategy together and devise a top-k similarity search algorithm HS-Topk. The pseudo-code is shown in Algorithm 6. It first initializes the queue $\mathcal{Q}$ and sets the threshold $\mathsf{UB}_{\mathcal{Q}} = \infty$ (line 2). Then it searches the HS-Tree from the root (line 3). If the value of $\mathsf{UB}_{\mathcal{Q}}$ is no larger than the minimum threshold supported by current level ($2^{i-1} \geq \mathsf{UB}_{\mathcal{Q}} + 1$), the algorithm terminates and we can safely prune remainder strings (line 4). For each level, we only visit HS-Tree with lengths between $|q| - \mathsf{UB}_{\mathcal{Q}}$ and $|q| + \mathsf{UB}_{\mathcal{Q}}$ based on length filtering (line 5). For each HS-Tree, it identifies the matched segments, groups strings with different numbers of matched segments into different groups, and visits the group sorted by the number

Algorithm 6: HS-Topk $(\mathcal{S}, q, k)$

Input: q: the query string; k: the size of result set
$\mathcal{S}$: The string set
Output: $\mathcal{R}$: the top-k answer

```
1  begin
2      Initialize queue Q and UB_Q;
3      for i = 1 to L do
4          if 2^{i-1} ≥ UB_Q + 1 then return;
5          for l ∈ [|q| − UB_Q, |q| + UB_Q] do
6              Identify strings with occurrence number
               larger than 2^i − UB_Q and group them to B_x;
7              for x = 2^i to 2^i − UB_Q do
8                  for each string s ∈ B_x do
9                      if GREEDYMATCH(s, q, i, 2^i − UB_Q)
                       then
10                         Verify ED (s,q);
11                         if ED(s, q) < UB_Q then
12                             Update Q and UB_Q;
13 end
```

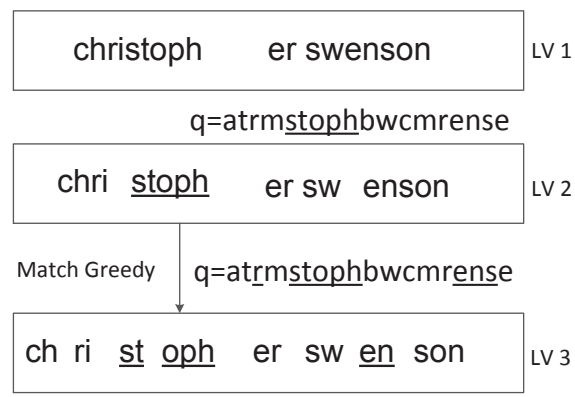


Fig. 5. An Example for Greedy-Match Strategy

in ascending order (line 6). For each string in the current group $\mathcal{B}_x$, we perform the greedy-match strategy (line 9). If the string passes the filter, we verify the candidate using threshold $\mathsf{UB}_{\mathcal{Q}}$ based on the techniques in Section IV-C (line 10). If $\mathsf{ED}(s, q) < \mathsf{UB}_{\mathcal{Q}}$, we use s to update $\mathcal{Q}$ and $\mathsf{UB}_{\mathcal{Q}}$ (line 12).

VI. EXPERIMENTAL STUDY

In this section, we conduct an extensive set of experiments and our experimental goal is to evaluate the efficiency of our algorithms and compare with state-of-the-art methods.

A. Experiment Setup

We used three publicly available real datasets in our experiments: DBLP Author, DBLP publication records[1] and Query Log[2], which are widely used in previous studies [9]. The details of data sets are shown in Table II. Author contains short strings, Query Log contains medium-length strings, and DBLP contains long strings.

We compared our algorithms with state-of-the-art methods. For threshold-based similarity search, we compared our HS-

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

TABLE II. DATASETS

Datasets	#	Avg Len	Max Len	Min Len
Author	612,781	15	46	6
Query Log	464,189	45	522	30
DBLP	1,385,925	105	1626	1

Search algorithm with Adapt [20], QChunk[16] and B^{ed}-tree[27]. Although there are some other similarity search algorithms, e.g., Flamingo[11] and VChunk [12], it has been shown in the previous studies that both Adapt and QChunk outperformed them [20], [16]. So we only compared HS-Search with Adapt and QChunk. For top-k similarity search, we compared our HS-Topk algorithm with Range [5], B^{ed}-tree and AQ[26]. We obtained the source code of Adapt, Range, B^{ed}-tree from the authors and implemented QChunk and AQ by ourselves [9]. All the algorithms were implemented in C++ and compiled using GCC 4.8.2 with -O3 flag. All the experiments were run on a Ubuntu server machine with two Intel Xeon E5420 CPUs (8 cores, 2.5GHz) and 32GB memory.

B. Evaluation on Threshold-based Search

1) Evaluating Different Verification Algorithms: We first evaluated the verification methods. We implemented three methods Length-aware, SingleThreshold and MultiExtension. Length-aware is the length-aware method [14]; SingleThreshold is the method that has only one threshold τ; MultiExtension is our method that has separate thresholds for each matched part based on Lemma 3. All the three methods were implemented with early-termination techniques. Figure 6 shows the results by varying edit-distance thresholds on the three datasets. We can observe that SingleThreshold involves less verification time than Length-aware because it can avoid duplicated computations on already matched segments and divided the two strings into different parts. For each part, MultiExtension has a different threshold and will terminate as soon as the edit distance of one part is larger than the given threshold of that part. Thus comparing with SingleThreshold, MultiExtension will terminate earlier.For example, on the Query Log dataset, for $\tau = 15$, Length-aware took 55 milliseconds on average, and SingleThreshold decreased the time to 39 milliseconds,while MultiExtension further reduced it to 24 milliseconds.

2) Filter Time vs Verification Time: Next we evaluated the cost of the filter step and the verification step and the result is show in Figure 7. We can see that our segment filter has great filtering power for small thresholds, and a large number of dissimilar strings can be pruned in the filter step, so the verification time is further reduced. For large thresholds e.g., $\tau = 15$ and $\tau = 20$ in Figure 7(c), the verification time is dominant in the overall time. This is because when the threshold becomes large, we need to do segment matching in lower levels and the segments would be much shorter. It is obvious shorter segments have more chance to be matched, so the number of candidates is larger than that of small thresholds.

3) Comparison with state-of-the-art methods: We compared our HS-Search algorithm with state-of-the-art algorithms Adapt,QChunk and B^{ed}-tree by varying different edit-distance thresholds on the three datasets Author, Query Log and DBLP. Figure 8 shows the results. We can see that HS-Search achieved the best performance on all the datasets and outperformed existing algorithms by 3 to 20 times. For example, on the Query Log dataset for $\tau = 10$, HS-Search took 6 milliseconds. And the average search time for Adapt, QChunk and B^{ed}-tree were 32, 151 and 104 milliseconds respectively. Among all the baselines, the overall performance of B^{ed}-tree was the worst because it had poor filtering power to prune dissimilar strings. Adapt had better performance than QChunk because Adapt took advantage of the adaptive prefix length to reduce a large number of candidates.

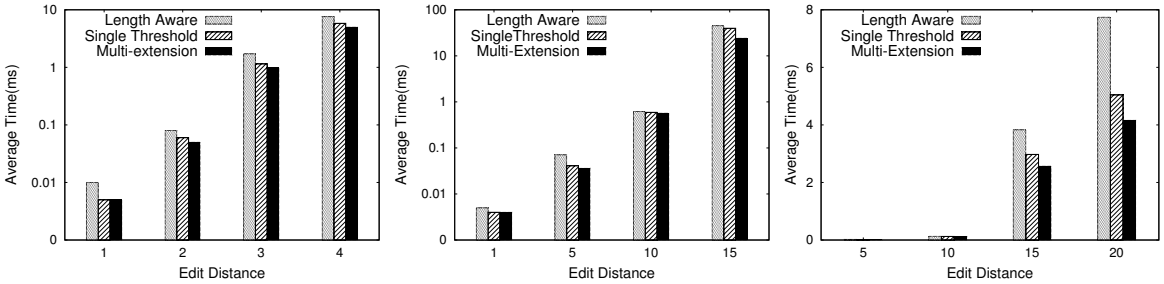
TABLE III. THRESHOLD-BASED SIMILARITY SEARCH: INDEX

Dataset	Method	Index Size(MB)	Index Construction Time(s)
Author	HS-Search	43.5	1.38
	Adapt	56	8.59
	QChunk	62	1.36
	B^{ed}-tree	32	5.0
Query Log	HS-Search	157	5.3
	Adapt	983	22.8
	QChunk	172	10.7
	B^{ed}-tree	129	86.0
DBLP	HS-Search	904	45.5
	Adapt	4194	121.9
	QChunk	425	70.5
	B^{ed}-tree	347	86.0

Our method achieved the best performance for the following reasons. First, existing algorithms are based on n-grams, and our method is based on segments which have much stronger filtering power than gram-based methods (as segments are longer than n-grams). Moreover, the segments are selected across the string and not restricted to the prefix. Thus, our method always generates the least number of candidates. Second, comparing with other similarity search algorithms we also design efficient verification mechanism. In this way, we can take advantage of the results of filter step and avoid redundant computations. Third, since previous algorithms are based on n-grams, they need to tune the parameter n for different datasets even for different thresholds on the same dataset to achieve the best performance. Our HS-Search algorithm does not need any parameter tuning. So the utility of HS-Search is much better than the existing algorithms.

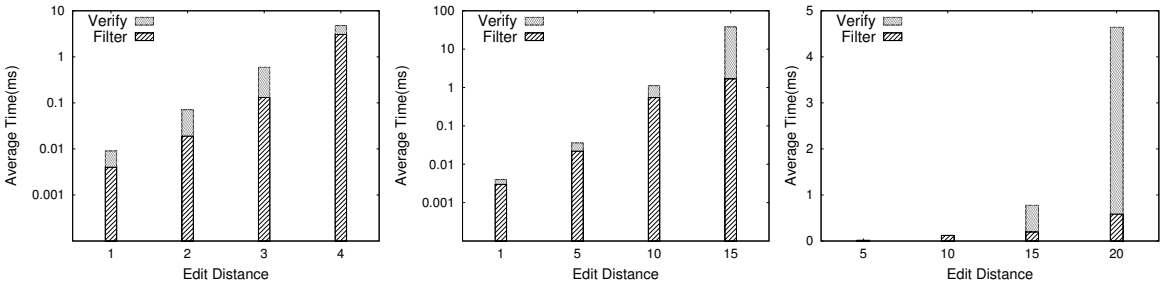
In addition, we compared the index construction time and index size and the result is shown in Table III. We have the following observations. First, HS-Search had the least index time because it can iteratively divide the segments in different levels and do not need to build a large inverted index like n-gram based methods. Second, the index size of HS-Tree is nearly the same with state-of-the-art n-gram-based methods, because HS-Tree partitions each string with length l into disjoint segments in each level with totally $1+2+...+2^{\log l} = \mathcal{O}(l)$ segments; and n-gram based methods generate $ln + 1$ grams. Thus they generate similar number of n-grams/segments and thus the index sizes are also similar. In addition, in the inverted lists, for a segment, HS-Tree only needs to maintain the string containing it but n-gram-based methods also need to store the position of n-gram, so our index size can be smaller than state-of-the-art methods. Third, B^{ed}-tree organizes "similar" strings into a B-tree node based on specific orders and does not need to maintain inverted lists, thus its index size is smaller.

4) Scalability: We evaluated the scalability of HS-Search. We varied the number of strings in each dataset and tested the average search time. Figure 9 shows the result on the three datasets. We can see that as the size of a dataset increased, our method scaled very well for different edit-distance thresholds and achieved near linear scalability. For example, on the DBLP data set, when the threshold was 20, the average search time for 400,000 strings, 500,000 strings and 600,000 strings were respectively 20, 27, and 33 milliseconds.



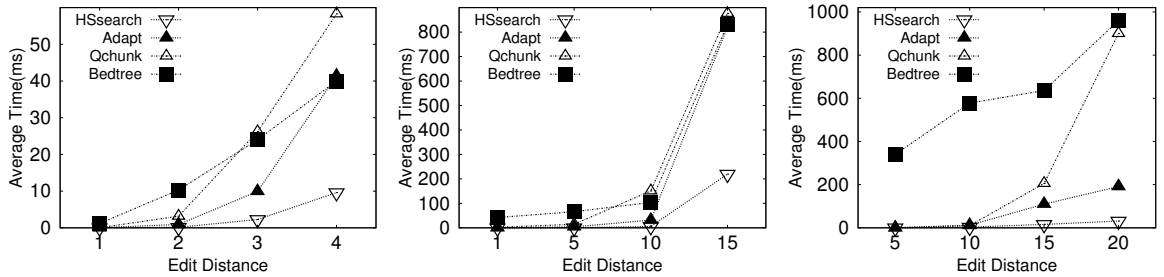
(a) Author(Avg Len = 15) (b) Log(Avg Len = 45) (c) DBLP(Avg Len = 105)

Fig. 6. Threshold-based Similarity Search: Evaluation on Different Verfication Algorithms.



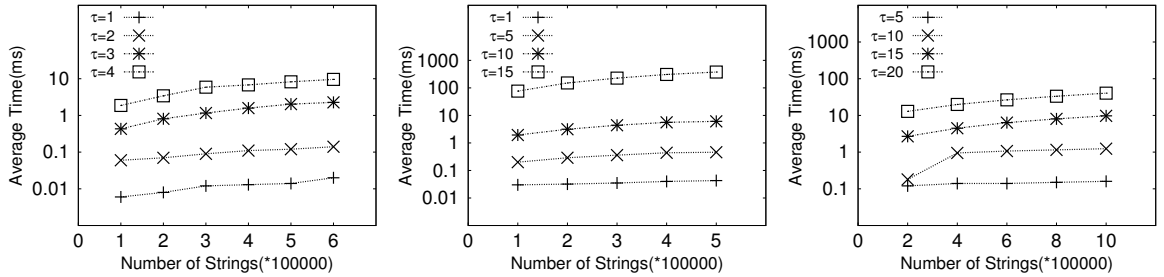
(a) Author(Avg Len = 15) (b) Log(Avg Len = 45) (c) DBLP(Avg Len = 105)

Fig. 7. Threshold-based Similarity Search: Filter Cost vs. Verification Cost.



(a) Author(Avg Len = 15) (b) Log(Avg Len = 45) (c) DBLP(Avg Len = 105)

Fig. 8. Threshold-based Similarity Search: Comparison with State-of-the-art Methods.



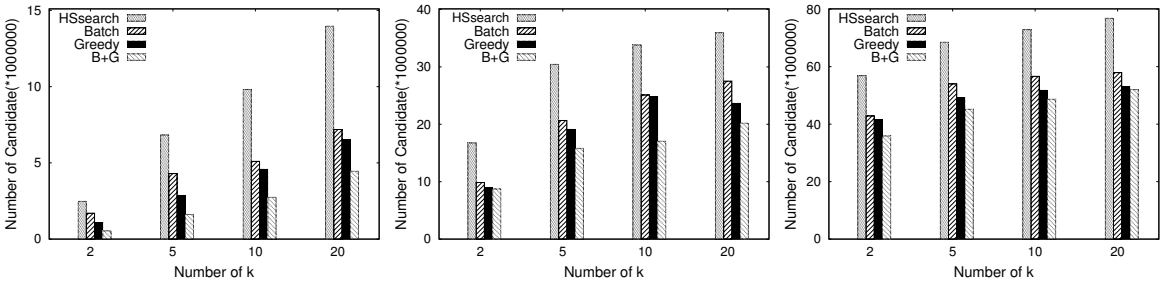
(a) Author(Avg Len = 15) (b) Query Log(Avg Len = 45) (c) DBLP(Avg Len = 105)

Fig. 9. Threshold-based Similarity Search: Scalability.

C. Evaluation on Top-k Similarity Search

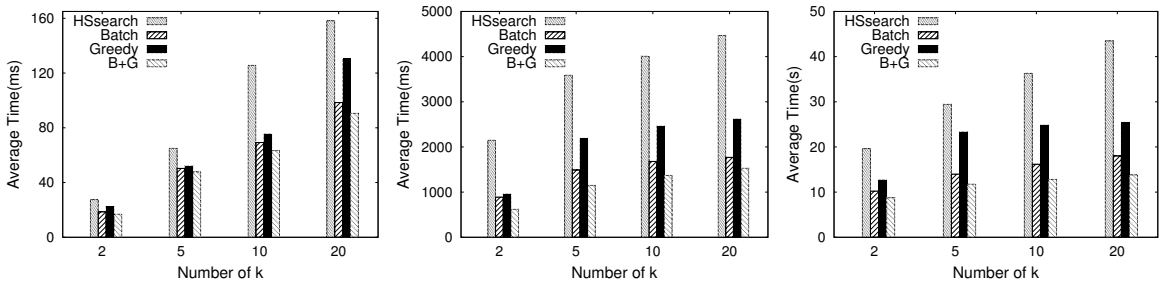
1) Evaluating Filtering Techniques: We first evaluated the efficiency of our filter techniques. We implemented four methods: HS-Search, Batch, Greedy and B+G. HS-Search extends the threshold-based algorithm by increasing the threshold by 1 each time and executing the algorithm multiple times; Batch only implements the batch-pruning-based method; Greedy only implements the greedy-match strategy; and B+G implements both batch-pruning-based method and greedy-match strategy. We evaluated the candidate number of each method to judge the filtering power. The result is shown in Figure 10. It is clear Batch can prune dissimilar strings in batch and thus reduce the number of candidates. As Greedy can find consecutive errors within a long segment, the number of candidates can also be reduced. We had significant filtering power by combining these two filters together. For example, on Author dataset with $k = 10$, HS-Search involved about 9.8 million candidates, Batch involved about 5 million candidates while Greedy involved 4.5 million candidates. Finally, B+G reduced the number to 2.7 million. This result shows the effectiveness of pruning techniques.

Then we evaluated the average search time. As shown in Figure 11, the average search time of Batch is much better than that of HS-Search because Batch can dynamically update the



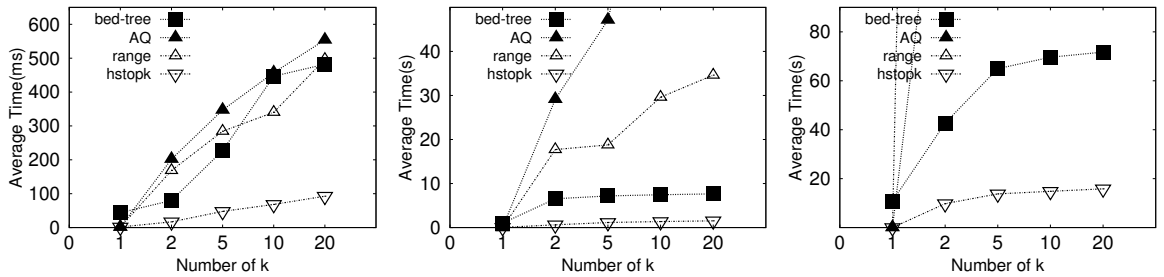
(a) Author(Avg Len = 15) (b) Query Log(Avg Len = 45) (c) DBLP (Avg Len = 105)

Fig. 10. Top-k Similarity Search: Candidate Number of Different Filters.



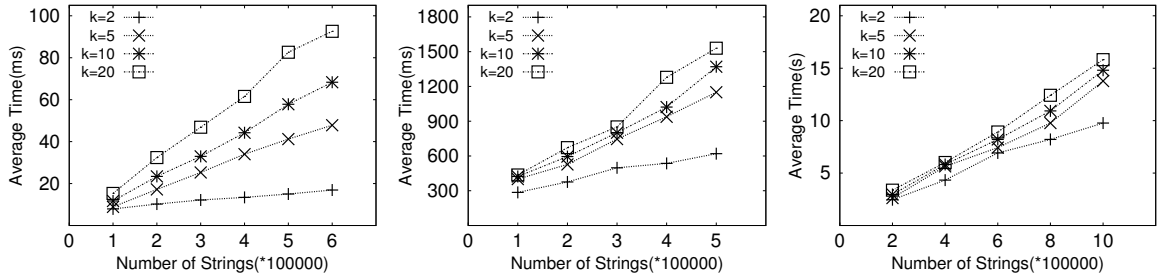
(a) Author(Avg Len = 15) (b) Query Log(Avg Len = 45) (c) DBLP (Avg Len = 105)

Fig. 11. Top-k Similarity Search: Average Time of Different Filters.



(a) Author(Avg Len = 15) (b) Query Log(Avg Len = 45) (c) DBLP (Avg Len = 105)

Fig. 12. Top-k Similarity Search: Comparison with State-of-the-art Methods.



(a) Author(Avg Len = 15) (b) Query Log(Avg Len = 45) (c) DBLP(Avg Len = 105)

Fig. 13. Top-k Similarity Search: Scalability.

threshold and prune strings in batch. Greedy was also better than HS-Search because it can reduce the candidate number by going to lower tree levels. However Greedy also involved relatively heavy filtering cost, so Greedy did not perform so well. By combining the two techniques, B+G can strengthen the filter power as well as reduce the filter cost and thus achieved the best performance.

2) Comparison with state-of-the-art methods: We compared our HS-Topk algorithm with state-of-the-art methods AQ, B^{ed}-tree and Range. We evaluated the performance on the same three datasets. For each experiment, we randomly selected 100 queries from the dataset and reported the average search time. All the algorithms were in-memory, including B^{ed}-tree. The results are shown in Figure 12.

We have the following observations. First, our HS-Topk algorithm outperformed all the existing methods. Second, on the Author dataset, Range outperformed B^{ed}-tree and AQ. This is because Range took advantage of the trie-based index [5]. If a large number of strings shared prefixes, Range had strong filtering power. Our method outperformed Range by nearly an order of magnitude because we can identify promising strings to estimate an upper bound and utilize the upper bound to prune large numbers of candidates for each threshold. Moreover, our batch-based-pruning and greedy-match techniques can also improve the performance.

TABLE IV. TOP-k SEARCH: INDEX

Dataset	Method	Index Size(MB)	Index Construction Time(s)
Author	HS-Tree	43.5	1.38
	Range	146	12.5
	AQ	316	6.4
Query Log	HS-Tree	157	5.3
	Range	1600	17
	AQ	224	78.3
DBLP	HS-Tree	904	45.5
	Range	4480	102.3
	AQ	1824	167.2

For instance, for $k = 10$, Range took 340 milliseconds on average, but our HS-Topk only took 41 milliseconds. Third, on the DBLP dataset with long strings, our method significantly outperformed other methods. We can see from Figure 12(c) that AQ and Range cannot finish within 10 hours. For instance, Range took more than 40,000 seconds to run the 100 queries when $k = 10$. This is because in datasets with long strings, the length of common prefixes is relatively short and there would be a large number of long, single branches in the trie index, which brings both space and computational overhead. Fourth,$\mathrm{B^{ed}}$-tree had relatively well performance on each dataset because it can group strings within a threshold together in one node and dynamically updated the threshold for pruning. But for small values of k, $\mathrm{B^{ed}}$-tree performed the worst because it involved many dissimilar strings in one node.

Table IV shows the index size and index time of each algorithm. We can see that HS-Tree involves both less space and time overhead than Range and AQ. Moreover, our method also outperformed them in query efficiency.

3) Scalability: We evaluated the scalability of HS-Topk. We varied the size of each datasets and tested the average query time for our HS-Topk algorithm. As shown in Figure 13, our method scaled very well with different k values and can support large-scale data. For example, on the Query Log dataset for k = 20, our method took 436 ms for 100,000 strings, and time increased to 672 ms for 200,000 strings and 1279 ms for 400,000 strings.

VII. CONCLUSION

In this paper, we have studied the problem of string similarity search. We proposed a hierarchical segment index to support both threshold-based similarity search and top-k similarity search. We devised an efficient algorithm HS-Search which utilized the segments to support threshold-based search queries. We extended this technique to support top-k similarity search and developed the HS-Topk algorithm with efficient filters which can further improve the performance. Experimental results show that our method significantly outperforms state-of-the-art algorithms on both threshold-based and top-k similarity search problems.

ACKNOWLEDGEMENT

This work was partly supported by the 973 Program of China (2015CB358700 and 2011CB302302), the NSF of China (61373024 and 61422205), Chinese Special Project of Science and Technology (2013zx01039-002-002), Beijing Higher Education Young Elite Teacher Project (YETP0105), Tencent, Huawei, the "NExT Research Center" funded by MDA, Singapore (WBS:R-252-300-001-490), and FDCT/106/2012/A3.

REFERENCES

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

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

[3] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *SIGMOD Conference*, pages 707–718, 2009.

[4] D. Deng, G. Li, and J. Feng. A pivotal prefix based filtering algorithm for string similarity search. In *SIGMOD Conference*, pages 673–684, 2014.

[5] D. Deng, G. Li, J. Feng, and W.-S. Li. Top-k string similarity search with edit-distance constraints. In *ICDE*, pages 925–936, 2013.

[6] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In *ICDE*, pages 340–351, 2014.

[7] M. Hadjieleftheriou, X. Yu, N. Koudas, and D. Srivastava. Hashed samples: selectivity estimators for set similarity selection queries. *PVLDB*, 1(1):201–212, 2008.

[8] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW*, 2009.

[9] Y. Jiang, G. Li, and J. Feng. String similarity joins: An experimental evaluation. *PVLDB*, 2014.

[10] Y. Kim and K. Shim. Efficient top-k algorithms for approximate substring matching. In *SIGMOD Conference*, pages 385–396, 2013.

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

[12] C. Li, B. Wang, and X. Yang. Vgram: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, pages 303–314, 2007.

[13] G. Li, D. Deng, and J. Feng. Faerie: efficient filtering algorithms for approximate dictionary-based entity extraction. In *SIGMOD Conference*, pages 529–540, 2011.

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

[15] G. Li, S. Ji, C. Li, and J. Feng. Efficient fuzzy full-text type-ahead search. *VLDB J.*, 20(4):617–640, 2011.

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

[17] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, pages 743–754, 2004.

[18] P. H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *J. Algorithms*, 1(4):359–373, 1980.

[19] J. Wang, G. Li, and J. Feng. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *PVLDB*, 3(1):1219–1230, 2010.

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

[21] W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit distance constraints. In *SIGMOD Conference*, 2009.

[22] X. Wang, X. Ding, A. K. H. Tung, and Z. Zhang. Efficient and effective knn sequence search with approximate n-grams. In *PVLDB*, volume 7, pages 1–12, 2014.

[23] C. Xiao, J. Qin, W. Wang, Y. Ishikawa, K. Tsuda, and K. Sadakane. Efficient error-tolerant query autocompletion. *PVLDB*, 6(6):373–384, 2013.

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

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

[26] Z. Yang, J. Yu, and M. Kitsuregawa. Fast algorithms for top-k approximate string matching. In *AAAI*, 2010.

[27] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *SIGMOD Conference*, pages 915–926, 2010.