minIL: A Simple and Small Index for String Similarity Search with Edit Distance

Zhong Yang[1], Bolong Zheng[1], Xianzhi Wang[2], Guohui Li[1], Xiaofang Zhou[3]
[1]Huazhong University of Science and Technology, Wuhan, China
Email: {zhongyang90, bolongzheng, guohuili}@hust.edu.cn
[2]University of Technology Sydney, Sydney, Australia
Email: xianzhi.wang@uts.edu.au
[3]The Hong Kong University of Science and Technology, Hong Kong, China
Email: zxf@cse.ust.hk

***Abstract*—The string similarity search is core functionality in a range of applications, including data cleaning, near-duplicate object detection, and data integration. We study the problem of threshold similarity search with the edit distance, where given a set of strings, a threshold k, and a query string q, we aim to find all strings in the set whose edit distances to q are no larger than k. Extensive studies have been proposed for the threshold similarity search problem with the edit distance. However, they suffer from a huge space consumption issue when achieving only an acceptable efficiency, especially for long strings. In this paper, we propose a simple yet small index, called minIL, to eliminate this issue. First, we adopt a minhash family to capture pivot characters and to construct sketch representations for strings. Second, we develop a multi-level inverted index to search sketches with a low space consumption. Finally, we apply a novel learned index technique on top of the index that further improves the query efficiency. Extensive experiments on real-world datasets offer insight into the performance of our method and show that it substantially reduces the index size, and is capable of outperforming the baseline approaches.**

***Index Terms*—threshold string similarity search, edit distance, invertd index, minhash**

I. Introduction

String similarity search, as one of the essential operating in data processing, has been concentrated and studied extensively in recent years [13], [15], [18], [19], [22], [24], [25]. Given a set of strings $\mathcal{S}$ and a query string q, we aim to find all strings in $\mathcal{S}$ that satisfy a query criteria under a similarity measure. Various string similarity measures are used, such as the cosine similarity, the jaccard similarity, and the overlap similarity. Among these similarity measures, the edit distance has a key advantage that it preserves the character ordering and captures a best alignment of two strings, which is crucial for applications, such as spell checking, plagiarism checking, speech recognition and protein/DNA sequences detection. For example, in the source tracking of COVID-19, the string similarity search with the edit distance is applied to find gene sequences similar to the virus in the genetic database. In this paper, we focus on the problem of threshold-based string similarity search under the edit distance that requires the distances between the results and the query string are within a given threshold.

Existing studies [12], [13], [15], [19], [24], [28] on threshold-based similarity search with the edit distance always suffer from either a low pruning rate issue or a huge space consumption issue. As is well known, the time cost of edit distance computation is $O(n^2)$ with a string length n. Therefore, the query efficiency is low when a large number of candidates need to be verified due to a low pruning rate issue, and the performance becomes even worse for long strings. Although approximate approaches [4], [5], [25], [27] guarantee the query efficiency on long strings, they still have a huge space consumption.

Given two strings s and q, the edit distance between s and q is the minimum number of edit operations needed to transform s to q. After conducting a preliminary experiment on existing datasets, we observe that the distribution of the characters to be edited in a string s is close to a uniform distribution with high probability, especially when the string is long. For example, the distribution of spell mistakes in an article, and the distribution of the mutated bases in a gene sequence. Intuitively, if we assume that the characters to be edited in strings are uniformly distributed, the probability that a randomly selected character in a string s needs to be edited is $\frac{k}{n}$, where k is the number of characters need to be edited and n is the string length. Accordingly, the probability that the characters do not need to be edited is $1 - \frac{k}{n}$. The probability is fairly high when $\frac{k}{n}$ is small, i.e., the edit distance between string s and q is small. The probabilities remains the same when the character is randomly selected from a certain interval of strings. Therefore, if we randomly select characters from multiple intervals independently to construct sketch representations for strings, the sketches of similar strings are likely to be similar. In other words, if the candidate sketches are similar to the query sketch, the candidate strings are similar to the query string with high possibility, and the results found with sketch strings have a high accuracy.

Therefore, we propose a novel approximate method that enables to find quickly the candidate strings whose characters need to be edited are uniformly distributed. The method fetches a series of pivot characters to construct a sketch representation for each string, and then computes the number of different pivots of the sketch representations between the

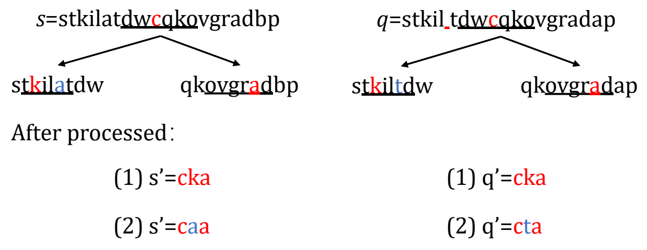

Fig. 1. A Running Example

TABLE I
SPACE COSTS OF EXISTING STUDIES

Algorithm	Space Complexity
minIL	$O(LN)$
minIL+trie	$O(LN)$
MinSearch [27]	$O(Nnlogn)$
Bed-tree [28]	NA
HS-tree [24]	$O(\sum_{l=l_{min}}^{l=l_{max}} \sum_{n=n_{min}}^{n=n_{max}} l*(\lvert S_l\rvert+n))$

candidate strings and the query. To construct the sketch representation, we first apply an independent stochastic hash function (e.g., minhash [3]) on an interval in the middle of a string to fetch a pivot. The string is divided into two substrings by the pivot. Then we recursively process the substrings to fetch more pivots. Based on the previous assumption, the probability that a string and the query string produce a same pivot at each recursion is $1-\frac{k}{n}$, while the probability of producing a different pivot each time is $\frac{k}{n}$. Obviously, if two strings are similar, their sketches are likely to be the same or to have only a few different pivots. In contrast, if two strings are dissimilar, most of the pivots between the sketch representations are different.

Consider an example in Fig. 1. We have two strings q and s where $|q|=19$ (the red underline in q represents a position shift compared with s) and $|s|=20$. As the edit distance between q and s is 2, the probability of obtaining the same pivot at each recursion approximates 0.9. First, we apply minhash to fetch a pivot from the middle 6-characters interval of s and q. The pivots are the same since s and q have the same interval *"dwcqko"*, and *"c"* (in red) is captured from both s and q. Afterwards, both s and q are divided to two substrings. The pivots are recursively fetched from the middle of the 6-characters intervals of the substrings. Finally, s and q are compacted to much shorter sketch strings s' and q', which are likely to be identical or differ by only one character. For example, (1) s'=q'=*"cka"*, or (2) s'=*"caa"*, q'=*"cta"*.

By combining the sketching method with two concise indexes, i.e., a trie-based index and a mutil-level inverted index, respectively, we develop two methods minIL+trie and minIL that achieve improvements on both space and time costs. Benefiting from the sketch representation, the space costs of the two proposed indexes are reduced to $O(LN)$, where N is the dataset cardinality and L is the sketch length. Since the space cost is independent on the string length, the method gains better performance on long strings. Table I shows the comparison on space costs of existing methods. We can see that the space cost of minIL is smaller than existing methods (The space cost of Bed-tree is not indicated explicitly in study [28], it requires more space than MinSearch [27]). Moreover, to further improve the query efficiency, we replace the length filter with a learned index technique to quickly locate the positions of the candidates in inverted lists. The experiments results show that both the space cost and query efficiency of the proposed methods outperform the competitors.

The major contributions are summarized as follows:

- We propose a novel sketching method that implicitly encodes alignments between strings and guarantees the similarity between sketch representations of similar strings with high probability.
- We propose a simple and small index minIL that substantially reduces the space cost compared with existing studies. We innovatively apply the learned index to replace the length filter to improve the query efficiency.
- We conduct extensive experiments on real-world datasets to offer insight into the performance of minIL that outperforms the existing methods, and the results show high efficiency and low space consumption of the method.

The rest of the paper is organized as follows. In Section II, we formalize the problem definition. In Section III, we introduce the sketch representation construction algorithm. We cover the index structure and the threshold search algorithm in Section IV. Section V introduces the optimizations for the extreme string shift problem. Experimental studies are presented in Section VI. Finally, we review the related work in Section VII and conclude the paper in Section VIII.

II. PRELIMINARIES

We proceed to formalize the problem definition. Frequently used notation is summarized in Table II.

Definition 1 (Edit Distance)**.** *Given two strings s and q, the edit distance between s and q, denoted as $ED(s,q)$, is the minimum number of edit operations, including substitution, insertion and deletion on a single character, needed to transform s to q.*

Definition 2 (Threshold-based Similarity Search with Edit Distance)**.** *Given a set of string $\mathcal{S}=\{s_1,s_2,...,s_N\}$ and a query string q, and a threshold k, the threshold-based similarity search with edit distance reports a set $\mathcal{R}$ of all strings $s_i\in\mathcal{S}$ such that $ED(s_i,q)\le k$.*

Example 1. *Consider a set of strings in Table III. Assume a query string q="above" with length of 5, and $k=1$, the threshold similarity search returns "abode" since the edit distance between "abode" and the query "above" is $1\le k$.*

III. STRING SKETCH CONSTRUCTION

We proceed to introduce the sketching method, called minhash compacting (MinCompact), that compacts long strings

TABLE II
SUMMARY OF NOTATIONS

Notation	Definition
$\mathcal{S}$	A set of input strings
N	Number of strings in $\mathcal{S}$
Σ	Alphabet of dataset
s or y	A string in $\mathcal{S}$
q or x	Query string
n	Length of a string
k	Threshold
t	Threshold factor $t = k/n \in [0, 1]$
l	Recursion times of the MinCompact method
L	Length of a sketch string
ε	Interval length parameter in MinCompact method
s' or y'	Sketch of a string in $\mathcal{S}$
q' or x'	Sketch of the query string
α	Number of different characters between x' and y'
$\mathcal{P}_\alpha$	Probability that x' and y' have α distinct characters
$\mathcal{R}$	The result set

TABLE III
A SET OF STRINGS

ID	String	Length
s_1	abode	5
s_2	about	5
s_3	abound	6
s_4	aboard	6
s_5	abstract	8

into short sketch strings by using an independent minhash family, which implicitly aligns the strings. MinCompact enables to find approximate matches to the query with a perfect accuracy (> 0.99).

A. Minhash Compacting

MinCompact compacts a string with length n to a sketch string with length $2^l - 1$ after l recursions (l is a small value). The details of the method are described in Algorithm 1. It first apples an independent minhash function on the middle $[(1/2 - \varepsilon)n : (1/2 + \varepsilon)n]$ characters of an input string y to find a *pivot* that has the minimal hash value. The pivot is then stored in the sketch string y'. The input string is divided into two substrings by the pivot, then we recursively process the substrings as input at the next recursion. The process is repeated by l recursions. MinCompact is to some extent inspired by the study [5] that is designed for embedding strings into a Hamming space. Different from [5] that embeds a string into a long, sparse string, MinCompact compacts a string into a short sketch string.

Example 2. *Consider $y = w_1w_2...w_{18}$ in Fig. 2. We set $l = 2$ and $2\varepsilon n = 4$ ($2\varepsilon n$ is the length of the interval for fetching a pivot). At the first recursion, the pivot w_9 is fetched from $[w_8 : w_{11}]$, then we push w_9 into y'. At the second recursion, pivots w_5 and w_{13} are obtained from $[w_3 : w_6]$ and $[w_{13} : w_{16}]$, respectively. Finally, after pushing the pivots into y', we have $y' = w_9w_5w_{13}$.*

Algorithm 1: MinCompact

```
Input: A string $y = w_1w_2w_3...w_n$, $l$, $\varepsilon$
Output: A new string $y'$
1 Initial $y'$ of size $2^l - 1$;
2 Select an independent minhash function $h$ and let $i$
    minimize $h(w_i)$ out of the
    $i \in [(1/2 - \varepsilon)n : (1/2 + \varepsilon)n]$ in $y$. We call $w_i$ the
    pivot;
3 Push $w_i$ into $y'$;
4 Recursively process $[w_1, ..., w_{i-1}]$ and $[w_{i+1}, ..., w_n]$
    until $l$-th level;
5 Return $y'$;
```

MinCompact has two advantages. One advantage is that it implicitly encodes alignments between strings. The string shifts always exist between similar strings, and a large string shift may decrease the probability of fetching a same pivot between similar strings, which leads to correct candidates missing. Therefore, the alignment between strings during fetching the pivots is required. MinCompact aligns strings by taking the pivot as the boundary of input strings at the next recursion. Once two strings produce a same pivot, the substrings are aligned from the location of the pivot. Therefore, the string shift issue at the next recursion is reduced. For example in Fig. 1, there is a position string shift between s and q (indicated by the red underline in q). After the first pivot "*c*" is fetched, the string shift remains between the first substrings of s and q, while the second substrings of s and q have no string shift, since they are aligned by the pivot "*c*", i.e., the string shift in the second substrings is eliminated. The other advantage is that the method can readily control the output length. The output length is a constant value $L = 2^l - 1$ determined by the parameter l.

B. Analysis

We proceed to illustrate the idea of MinCompact. If two strings x and y are similar, they are close naturally in length, $|x| \approx |y| \leq n$, where n is the string length after trans-forming one string into the other. For ease of presentation, we first discuss edit operations of substitution and insertion. For example in Fig. 1, instead of deleting "*a*" from s, we substitute "*b*" to "*a*" in s and insert "*a*" into q to transform two strings into the same. If the edit distance between x and y is k ($k \ll n$), there are k characters need to be edited. Assume that k characters are uniformly distributed in strings. Intuitively, the probability that a stochastically selected character in the string that requires to be edited approximates to $\frac{k}{n}$. The probability remains unchanged when the character is selected from an interval of the string. That is to say, if we stochastically selected a character as a pivot from each of two similar strings, the probability $\mathcal{P}(differ)$ that the pivots are different approximates to $\frac{k}{n}$. The probability $\mathcal{P}(same)$ that

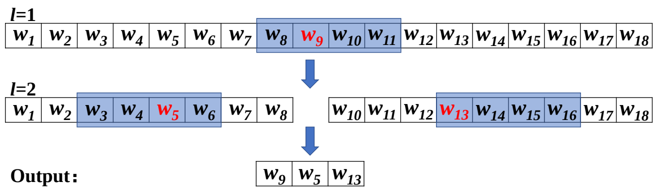
l=1
w1 w2 w3 w4 w5 w6 w7 w8 w9 w10 w11 w12 w13 w14 w15 w16 w17 w18
l=2
w1 w2 w3 w4 w5 w6 w7 w8 w10 w11 w12 w13 w14 w15 w16 w17 w18
Output:
w9 w5 w13

Fig. 2. An Example of MinCompact Procedure

the pivots are identical approximates to $1-\frac{k}{n}$. For the deletion operation, the only difference is that n will be slightly smaller, which does not affect the conclusion.

Based on the above conclusion, the sketch strings $x'$and y' that have L independent pivots are likely to differ only in a few pivots. The probability that x' and y' have α different pivots is:

$$\mathcal{P}_\alpha \approx C_L^\alpha(\frac{k}{n})^\alpha(1-\frac{k}{n})^{L-\alpha} \tag{1}$$

The probability that x' and y' have no more than α different pivots is:

$$\mathcal{P}(\mathit{differ} \le \alpha) \approx \sum_0^\alpha \mathcal{P}_i \tag{2}$$

For instance, if $l=3$, $|x|=|y|=n$, and $ED(x,y)\le 0.1n$, in this case, y is a result of the query x with a threshold $k=0.1n$. According to the above derivation we have the probabilities: $\mathcal{P}_0 \approx 0.478, \mathcal{P}_1 \approx 0.372, \mathcal{P}_2 \approx 0.124$, $\mathcal{P}_3 \approx 0.023$ and so on. Then probability that x' and y' have no more than 3 different pivots is $\mathcal{P}=\mathcal{P}_0+\mathcal{P}_1+\mathcal{P}_2+\mathcal{P}_3 \approx 0.997$. In other words, if x' and y' have no more than 3 different pivots, the probability that the original strings x and y are similar is 0.997. It indicates that the accuracy of the results whose sketch string has no more than 3 different pivots to x' is 0.997. Therefore, we enable to find the results whose edit distances with x are no larger than k with an accuracy $\sum_0^\alpha \mathcal{P}_i$. We can readily achieve a perfect accuracy (> 0.99) by adjusting α.

C. Effectiveness of Parameters

MinCompact considers two parameters l and ε. The parameter l dictates the maximum recursion depth and determines the length of sketch strings. It affects the computation cost and the probability $\mathcal{P}_\alpha$ of candidates being similar with the query. Obviously, a small l is needed such that the length of the input string at each recursion is no smaller than the number of characters to scan. At each recursion, the algorithm scans $2\varepsilon n$ characters, and the input length at the i-th recursion is at least $(1/2-\varepsilon)^{i-1}n$. We need to choose l such that at the l-th recursion, the input length is no smaller than $2\varepsilon n$. Therefore, it suffices to set l that satisfies:

$$l \le \log_{1/2-\varepsilon} 2\varepsilon + 1. \tag{3}$$

The parameter ε controls the computation cost of MinCompact. When ε decreases, the input length at each recursion decreases, and the computation cost is reduced. The method scans $2\varepsilon n$ ($\frac{1}{2n} < \varepsilon < \frac{1}{2}$) characters at each time, and 2^l-1 times in total. So the time cost is $O(\beta n)$, where $\beta = 2(2^{\log_{1/2-\varepsilon} 2\varepsilon+1}-1)\varepsilon < 1$, and iff $\varepsilon=\frac{1}{2n}$ or $\varepsilon=\frac{1}{2}$, $\beta=1$. The parameter ε also affects the ability of string shift tolerance. A larger ε implies greater tolerance. Although the method can tolerate string shift, if ε is too small while the string shift is large, the characters in middle intervals between similar strings may be different. Therefore, the probability of producing the same pivot in the intervals is decreased, which reduces the algorithm accuracy. So there is a trade-off between the computation cost and the ability of string shift tolerance that requires a proper setting of parameter ε.

D. Extreme String Shift Issue

It is worth noting that although a proper ε enables to improve the string shift tolerance ability, the method suffers from extreme string shift cases, i.e., the shifts are all con-centrated at the beginning or at the end of the string. To deal with the problem, we propose two optimizations in this paper. The first one is to utilize a slightly larger ε at the first recursion in MinCompact. As discussed, a large ε can tolerate large string shift. If we apply a large ε at the first recursion, a high probability of producing a same pivot in the intervals remains. If a same pivot is obtained between similar strings at the first recursion, the input strings in the next recursion are aligned, and so does the input strings in the subsequent recursions. Therefore, the string shift is reduced and the ability of string shift tolerance is improved. The second optimization developed on top of the query algorithm and the index structure is described in Sec. V.

E. Other Issues

MinCompact helps to find similar strings and filter dissimilar ones with high probability. However, two types of inappropriate candidates may be produced. One type is the candidates that have a large length difference that exceeds the threshold with the query. The string length varies greatly, which can be observed from the dataset statistics in Table IV. After applying MinCompact, since the sketch strings are with the same length, the strings with large length differences may have similar sketch strings. The other type is the candidates that are dissimilar to the query, but contain pivots that happens to be the same as the query. For instance, the string segments "*acdfge*" and "*hkljma*" from two dissimilar strings can produce the same pivot "*a*". To deal with these issues, two pruning strategies are introduced along with our index structure in Sec. IV.

IV. Threshold Based Similarity Search

After applying MinCompact, each string s is compacted into a sketch string s'. To answer the threshold based similarity query, we first find the candidates $\{s_i\}$ whose sketch strings s_i' have no more than α different pivots with the sketch string q' of the query. Then, we verify whether $ED(s_i,q)$ of all candidates s_i satisfies the criteria. To prune inappropriate candidates quickly, we propose a trie-based index. To further

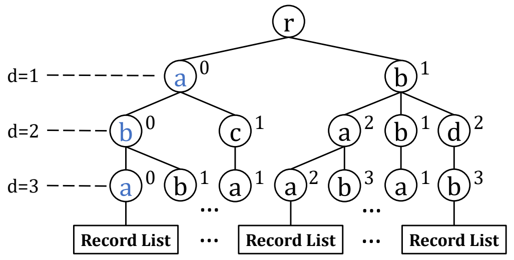

Fig. 3. An Example of Marked Equal-Depth Trie

reduce the space cost, we develop a multi-level inverted index with a learned index based length filter.

A. Trie-based Index

A straight-forward way to find the candidates is the linear scan method that compares each s'_i with q' to compute the number of different pivots. Although the sketch strings are short, it is costly to scan all strings especially when the dataset cardinality is large. The trie index is a well-adopted structure for strings. Although it cannot be applied directly for searching the original long strings under the edit distance, it is suitable for searching the short sketch strings. Unlike the traditional trie index that stores strings with various lengths and searches for strings with common prefixes, we need a structure to save strings with equal length and to search strings that have no more than α different pivots to the query sketch q'. According to the requirements, we design a marked equal-depth trie index structure.

Index Structure and Search Algorithm Fig. 3 shows an example of the marked equal-depth trie with a depth of 3. Each node represents a character with a mark at the top right of the node. The mark records the number of different pivots up to the node during the search process. Each leaf node links a record list of sketch strings whose characters are represented by the route from the root to the leaf node. The trie is built by inserting all sketch strings into the tree. And the marks attached on nodes are 0 initially. Algorithm 2 presents the recursive searching procedure on the trie. We start from the root node and traverse all the child nodes of the root node. We check the mark on the child node whether it is no larger than α. If the mark $\hat{\alpha}$ exceeds α, the successor nodes of the child can be pruned (Line 6-7). If the mark is within the limit of α, then we check the character represented by the node. If the character is equal to the current character of q', we recursively process the child nodes of the node (Line 10), otherwise, we recursively process the child nodes with their marks adding 1 (Line 12). In each iteration, if the input node is a leaf node, the linked record list is merged into the results $\mathcal{R}$ after length filtering and position filtering (Line 1-3).

Length Filter. As mentioned before, the string length varies in a dataset. But after MinCompact, the sketch strings have the same length. The candidates may have a large length difference that exceeds the threshold with the query. To address the issue, we attach the original string length to each record in the record

Algorithm 2: TrieSearch

```
   Input: Query sketch string $q'$, a node of trie $node$, $\alpha$,
          mark $\hat{\alpha}$
   Output: String Set $\mathcal{R}$
1  if $node$ is the leafnode then
2      $\mathcal{R}_n \leftarrow node.RecordList$ after LengthFilter and
         PositionFilter;
3      $\mathcal{R} \leftarrow \mathcal{R} \cup \mathcal{R}_n$
4  else
5      foreach $child \in node$ do
6          if $\hat{\alpha} > \alpha$ then
7              continues;
8          else
9              if $q'[i] == node.character$ then
10                 TrieSearch($q'[i+1], child, \alpha, \hat{\alpha}$)
11             else
12                 TrieSearch($q'[i+1], child, \alpha, \hat{\alpha}+1$)
13 Return $\mathcal{R}$;
```

lists to do the filtering. When we look up a record, the attached length is compared with query length. If the length difference is larger than the given threshold, the record can be pruned since its original string can not be a result.

Position Filter. Dissimilar strings may produce similar sketch strings by coincidence. Pruning these candidates can further improve the query efficiency. To alleviate the problem, the position of each pivot in the original string is attached to each record in the list. Then, if a record contains a same pivot with the query, positions of the pivot in their original strings are compared. If the position difference is larger than the threshold, which is not a feasible alignment, the pivot can be considered as different. In this way, we can reduce the improper candidates whose sketches happen to be similar to the query.

Example 3. *Consider the trie of sketch strings in Fig. 3. Suppose the query sketch string q'="aba" and $\alpha = 1$. We traverse all the nodes in the tree from root r to the leaf nodes. When we meet the nodes at depth 1, node a equals to $q'[0]$="a" while node b does not, then the mark of node a keeps 0 while the mark of node b adds 1, and the marks are passed to their child nodes, respectively. When we meet the node a at depth 2, since it does not equal to $q'[1]$="b", its mark becomes 2 which is larger than α, then its successor nodes can be pruned. Similarly, the successor nodes of d at depth 2 can be pruned. Once we meet a leaf node and its mark do not exceed α, the records linked to it is merged into the results after two filtering.*

Cost Analysis. The space cost of the trie-based index is smaller than $O(LN)$ since the common prefixes reduces a certain amount of storage. On the other hand, the trie-based index needs to express various relationships between

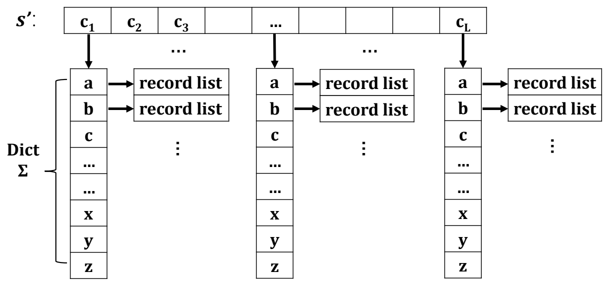

Fig. 4. Multi-level Inverted Index

characters, so its implementation is more complicated, which incurs additional space costs. Suppose the average number of branches of the nodes is σ ($\sigma \leq |\Sigma|$, where Σ is the alphabet of all strings), the time cost of searching the trie is at least $O(\sigma^{\alpha})$ since we need to traverse all the nodes within depth α before any pruning. In the worst case, the time cost is $O(\sigma^{L})$.

B. Multi-level Inverted Index

Although the trie-based index is smaller compared to the existing tree-based indexes [24], [28], its space consumption is still non-negligible, especially when the size of dataset is large. To further reduce the space consumption and improve the search efficiency, we propose a simple and small index structure called multi-level inverted index (minIL) along with a learned index based length filter.

Index Structure. The multi-level inverted index consists of L levels of inverted indexes corresponding to the sketch string length L. For each position j of the sketch strings, there is one level inverted index. Each level inverted index consists of record lists of characters $c \in \Sigma$. Each record list of a character c consists of the sketch strings that contain character c at position j. Fig. 4 illustrates the structure of the index. And the Index is built as Algorithm 3 shown. We first initialize the index with L levels. Secondly, each string $s_i \in \mathcal{S}$ is transformed into s'_i by the MinCompact. Thirdly, for each character c at position j in s'_i, we add s'_i into the record list of c in the j-th level of the inverted index, denoted as $\mathcal{I}_j(c)$. Finally, after all strings are processed, the index is set up.

Searching algorithm. The algorithm for threshold based similarity search on the multi-level inverted index is described in Algorithm 4. Given a set of strings $\mathcal{S}$, a query string q and a similarity threshold k, the algorithm returns all strings $s_i \in \mathcal{S}$ such that $ED(s_i, q) \leq k$. The search method can be used for different thresholds with different accuracy at query time. First, we use Algorithm 1 and Algorithm 3 to process all input strings and generate the index $\mathcal{I}$. Then we construct q' for q. Afterwards, for each pivot $q'[i]$ at position i, we scan the list of $\mathcal{I}_i(q'[i])$. After the length filtering (Line 5) and position filtering (Line 6), for each s_i in the list, if it is not in the hashmap $Map\langle s_i, f\rangle$, which contains records and their frequencies, it is inserted into the map, otherwise, its frequency plus one (Lines 7-9). After all positions of q' are processed, we obtain the map contains strings whose sketch strings have intersection with q'. We check the strings whose frequency satisfies $L - f \leq \alpha$ to determine the candidates. Finally, we verify the candidates and return the results (Lines 12-14).

Algorithm 3: Building Index

```
Input: String Set $\mathcal{S}$,$L$,$\Sigma$
Output: Indxe $\mathcal{I}$
1 Initial $\mathcal{I}$ with $L$ levels and each level has $\Sigma$ record
  lists. foreach $s_i \in \mathcal{S}$ do
2     $s'_i \leftarrow MinCompact(s)$;
3     foreach $c \in s'_i$ do
4         $c$ is at position $j$ of $s'_i$;
5         $\mathcal{I}_j(c) \cup s'_i$;
6 Return $\mathcal{I}$;
```

Algorithm 4: Threshold Based Similarity Search

```
Input: A query string $q$, Index $\mathcal{I}$, threshold $k$
Output: String Set $\mathcal{R}$
1 Choose $\alpha$ according to the accuracy requirement;
2 $q' \leftarrow MinCompact(q)$;
3 for $i \leq L$ do
4     foreach $s'_i \in \mathcal{I}_i(q'[i])$ do
5         $LengthFliter(s_i)$;
6         $PositionFliter(s_i)$;
7         if $s_i \in Map\langle s_i, f\rangle$ then
8             $f \leftarrow f + 1$;
9         Add $\langle s_i, 1\rangle$ into $Map$;
10 foreach $s_i \in Map$ do
11     if $L - f \leq \alpha$ then
12         if $ED(s_i, q) \leq k$ then
13             $\mathcal{R} \leftarrow \mathcal{R} \cup s_i$;
14 Return $\mathcal{R}$;
```

Cost Analysis. Different from the existing indexes that store various substrings with a large redundancy, the multi-level inverted index reserves the sketch strings with only a small redundancy, and does not use extra structure compared to the trie-based index. Although the space cost is still $O(LN)$, the cost in practice is smaller than that of the trie-based index. The search algorithm scans L record lists and each record list has a length $\frac{N}{|\Sigma|}$ at average. Therefore, the time cost of the search method excluding the verification phase is $O(\frac{LN}{|\Sigma|})$.

Remark. Our proposed method can achieve the perfect accuracy (> 0.99) by adjusting α. What's more, α is data independent to achieve the perfect accuracy: it depends on the threshold factor $t = \frac{k}{n}$ and the parameter l. The selection of α is further illustrated in the experimental study. It is worth noting that even α is not selected appropriately, a high accuracy can be achieved by repeating the MinCompact with

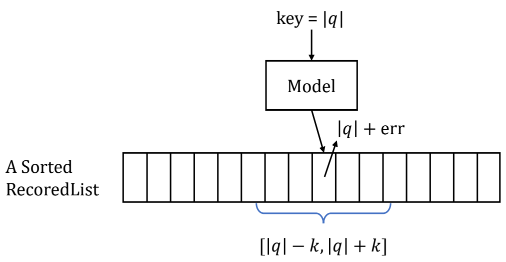

Fig. 5. Searching Strings by A Learned Index

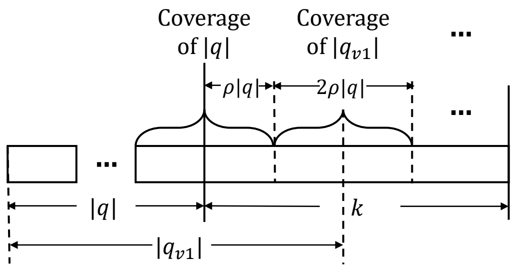

Fig. 6. Analysis of Filling A Query String

different minhash families. In this way, multiple sketch strings are produced for each string, which results in larger index size. It is usually not necessary in practice, as a single MinCompact is already good enough. Moreover, the multi-level inverted index can be scanned in parallel without any modification.

C. Improvements with Learned Index Technique

We use a naive way that traverses the record list to do the length filtering. It is inefficient when the size of record list is large. Actually, we only need to retrieve the strings whose lengths are within the range of $[|q| - k, |q| + k]$ with the threshold k. To quickly locate positions of strings within the range, many technologies can be applied after sorting strings by length. Binary search or B-tree is a common option. A recently proposed novel learned index structure which is much more efficient than binary search or B-tree is available here. Learned indexes [9], [11], such as RMI, use machine learning techniques to model the cumulative distribution function (CDF) of a sorted array. We apply the learned index technique to the length filter, called *learned length filter*, by training a model for each record list separately to replace the plain length filter when building the index. Then the models are utilized to speed up searching the sorted record lists. For example, Fig. 5 shows the process of searching strings within the length range $[|q| - k, |q| + k]$ using the learned length filter. We take the length $|q|$ as the input key of the learned model, the model outputs key location in the sorted record list within O(1) time. Although the model of learned index always has a search error err, using the length $|q|$ as the key can avoid the error as long as the key location falls in the search range, i.e., $|q| + err \in [|q| - k, |q| + k]$. With an acceptable accuracy of the model, it happens with high probability. Next, we retrieve both sides from the location $|q| + err$ to traverse the search range sequentially.

Compared with traditional index structures, the learned index has significant high efficiency. The performance of our search method benefits from utilizing learned index: The time cost of searching a single record list is reduced to $O(2k)$, thus the time cost of searching the candidates is reduced to $O(2kL)$.

V. Optimization for String Shift Issue

In the analysis of MinCompact in Sec. III, we make an optimistic assumption that the characters need to be edited are uniformly distributed in the string. If the string is long and random, the distribution of characters need to be edited satisfies the assumption with high probability. However, in practise, there exist exceptions to the assumption, i.e., the extreme string shift issue that the shifts are all concentrated together at the beginning or at the end of the string. The extreme string shift issue may be caused by human or system errors are uncommon but inevitable. For example, in text strings, an article loses a sentence at the beginning, or in gene strings, the last segment of a gene sequence is missing. Although MinCompact has the ability to tolerate string shift, it still suffers from the extreme cases.

When MinCompact has a small parameter ε, the extreme string shift cases could be problematic: At the first recursion, the characters in fetching intervals between similar strings may be totally different, which decreases the probability of producing a same pivot. Afterwards, without the same pivot at the first recursion, the substrings are not aligned, the characters in fetching intervals at the next recursion may also be different. This is a chain reaction that eventually leads to dissimilar sketch strings. Recall the discussion in Section III, we propose to use a slightly larger ε at the first recursion to deal with the issue. However, the optimization cannot completely solve the problem. In experiments, we use a synthetic dataset that only contains strings that have extreme string shifts to test the effectiveness of the optimization in Sect. VI. The optimization helps to achieve about 40% accuracy on the extreme data which is still not acceptable.

A. Improvements on Query Processing

We propose another optimization to improve the accuracy on extreme string shift cases. Inspired by [15], we come up with an idea of making the query string be aligned with shifted strings by truncating or filling the query string at the beginning or the end to reduce or even eliminate string shifts. Since strings have various lengths, we apply different alignment schemes on the query to cover different string shifts. For candidate strings that are shorter than the query, we truncate several characters at the beginning or end of the query. For candidate strings that are longer than the query, we fill the query with several place-holders at the beginning or the end.

We obtain multiple variants of the query through the above operations. The question is: how many characters/placeholders should be truncated/filled, in another words, how many variants of the query should be acquired. Suppose the string shift tolerance size of a query string q using MinCompact is $\rho \cdot |q|$. Therefore, the query string can cover shifted strings with lengths in the range of $[|q| - \rho \cdot |q|, |q| + \rho \cdot |q|]$. Similarly, a variant q_{v_i} of the query can cover shifted strings with lengths in the range of $[|q_{vi}| - \rho \cdot |q|, |q_{vi}| + \rho \cdot |q|]$ (for simplicity, suppose the string shift tolerance size of q_{vi} is still $\rho \cdot |q|$). Thus, each variant q_{v_i} can extend the shift coverage of q by $2\rho \cdot |q|$ at most. Consider the situation in Fig. 6 that we need to cover all the shifted strings with lengths in the range of $[|q|, |q| + k]$, where k is the threshold. Then, we need at least m variants each of which covers a subrange $[|q| + (2i-1)\rho \cdot |q|, |q| + (2i+1)\rho \cdot |q|], (i = 1, 2, .., m)$ to satisfy $(2m+1) \cdot \rho \cdot |q| \geq k$. Accordingly, the total filling size is $2i \cdot \rho \cdot |q|, (i = 1, 2, ..., m)$. Since ρ is an uncertain value, we approximate it with $\frac{k}{(2m+1)|q|}$. Therefore, the filling size is $\frac{2i \cdot k}{2m+1}$. The total number of the variants is $4m$ (truncating/filling strings at the beginning/end). In practice, $m = 1$ is always enough to cover all the string shift possibilities when k is small. When $m = 1$, we fill the query string with $\frac{2}{3}k$ placeholders for longer strings, and similarly, we truncate $\frac{2}{3}k$ characters of the query for shorter strings.

Aligning query string to extreme string shift cases produces multiple variants that leads to time increase. Even though $m = 1$ is enough in the most cases, there are four variants that need to be processed. Fortunately, the search ranges in the record lists of the variants are half the length of the original query and the candidates can be readily found using the learned length filter. For the variants of longer strings, we only retrieve the strings with a length in the range of $(|q|, |q| + k]$. While for the variants of shorter strings, we only retrieve the strings with a length in the range of $[|q| - k, |q|)$. Take searching strings within the range of $(|q|, |q| + k]$ as an example. We use a median $\frac{2|q|+k}{2}$ of the search range as the input key of the learned length filter to avoid the model error. As long as the location of key falls in the search range, i.e., $\frac{2|q|+k}{2} + err \in (|q|, |q| + k]$, we can traverse the records within the range sequentially to find the candidates. Benefiting from the high performance of learned length filter, although the time cost increases due to the multiple variants, it is still acceptable.

VI. Experimental Study

We report on extensive experiments with real-world datasets that offer insight into the performance of our proposed algorithms. In particular, we aims to answer the following questions:

$Q1$: How to choose parameters that affect the performance of minIL?

$Q2$: How much does minIL outperform the competitors under default settings?

$Q3$: How much does minIL reduce the space consumption compared with the competitors?

$Q4$: How does the optimizations work under string shift?

TABLE IV
Statistics of Datasets

Dataset	Cardinality	avg-len	max-len	$\|\Sigma\|$	q-gram
DBLP	863053	104.8	632	27	1
READS	1500000	136.7	177	5	3
UNIREF	400000	445	35213	27	1
TREC	233435	1217.1	3947	27	1

A. Experiment Setup

All the algorithms are implemented in C++ compiled using 64-bit addressing. All experiments are run on a Window 10 machine with an Intel 3.4Hz CPU and 32GB memory. We release our source code on Github[1].

Datasets. We conduct the experiments on four real-world datasets:

- DBLP[2]: A dataset of DBLP publication information including authors, title and key words of papers.
- READS[3]: A dataset contains short DNA sequencing reads, which was used in the edit similarity joins and search competition.
- UNIREF[4]: A dataset of protein sequences obtained from the website of UniProt project.
- TREC[5]: A dataset of publication information including the authors, title, and abstract of papers in 270 medical journals.

The datasets vary in cardinality, average length, maximum length, and the dictionary size $|\Sigma|$. Table IV shows the dataset statistics. The average length of strings in DBLP and READS is small, while their cardinalities are large. The average string length in UNIREF and TREC is much larger, especially the average string length in TREC, which is over 1000. The max length of UNIREF is quite large, it reaches 35213.

Algorithms. We compare our proposed method with the following competing algorithms:

- MinSearch[6] [27]. It uses a local hash minima method to extract common substrings between similar strings and stores the substrings in a hash table.
- Bed-tree[7] [28]. It uses several word ordering strategies together with the B+-tree structure to perform the search.
- HS-tree[8] [24]. It uses a hierarchical segment tree that recursively stores the substrings of the half of the string that contained in the parent node.
- minIL+trie. The proposed method using the trie-based index.

[1]https://github.com/yangzhong901/minIL
[2]https://dblp.uni-trier.de/
[3]https://www2.informatik.hu-berlin.de/~leser/searchjoincompetition2013/
[4]http://trec.nist.gov/data/t9_filtering.html
[5]http://fimi.ua.ac.be/data/
[6]https://github.com/kedayuge/Search
[7]https://github.com/ZhangZhenjie/bed-tree
[8]https://github.com/TsinghuaDatabaseGroup/Similarity-Search-and-Join/tree/master/hstree

TABLE V
DEFAULT PARAMETER SETTINGS

parameters	**values**
l	2, 3, <u>4</u>, <u>5</u>, 6
γ	0.3, 0.4, <u>0.5</u>, 0.6, 0.7
t	0.03, 0.06, 0.09, 0.12, <u>0.15</u>

TABLE VI
A BRIEF SELECTION OF α

$l=3$			$l=4$			$l=5$		
t	α	accuracy	t	α	accuracy	t	α	accuracy
0.03	2	0.999	0.03	2	0.990	0.03	4	0.998
0.06	2	0.994	0.06	4	0.998	0.06	5	0.991
0.09	3	0.998	0.09	4	0.992	0.09	7	0.995
...	...	...	...	...	...	...	...	...

- minIL. The proposed method using the multi-level in-verted index.

We download the source codes of these algorithms and recompile under our environment settings. For parameter settings of algorithms, we always choose the recommended parameter combinations from the papers. MinSearch has one parameter α using a small value (e.g., $\alpha = 3$). Bed-tree has several parameters to choose. HS-tree has no input parameter.

We compare the above algorithms on all datasets. However, HS-tree is not applicable on UNIREF and TREC, since it takes too much memory usage that exceeds our computer's limit. Therefore, we do not display its results on UNIREF and TREC. As mentioned in Sec. I, most existing methods suffer from the massive memory usage and may be inapplicable on large datasets, where HS-tree is a representative one.

B. Self Evaluation (Q1)

We report the performance of minIL under different parameter settings, and explore the effectiveness of the parameters.

Parameters Settings. We now present the parameter settings. Two main parameters l and ε are considered, while the parameter α is spontaneously determined by the given threshold k, string length n and the parameter l to achieve a perfect accuracy. For ease of presentation, we use a threshold factor $t = \frac{k}{n}$ instead of the threshold k in the experiments to keep α consistent for different query lengths under a same accuracy. A small l is required to avoid running out of the length of the string in the recursion. A proper selection of ε is required to balance between the computation cost and the accuracy.

We employ a heuristic method to tune the parameters l and ε. The main idea is that we first set a large l according to the average length of string (For example, the average string length in DBLP is about 100, we set $l = 4$, and the average string length in READS is about 500, we set $l = 5$) and then vary ε to check whether l is feasible. If not, we decrease l and repeat the above procedure. In addition, since the equation of computing ε is complicated when l is given, we set $\varepsilon = \frac{\gamma}{2(2^l-1)}$ and

TABLE VII
PERFORMANCE OVERVIEW WITH DEFAULT SETTINGS

Dataset	Algorithm	Memory Usage (GB)	Query Times (s)
DBLP	minIL	<u>**0.52**</u>	<u>**0.003**</u>
	minIL+trie	1.5	0.006
	MinSearch	1.7	0.011
	Bed-tree	4.8	0.110
	HS-tree	7.8	0.007
READS	minIL	<u>**1.1**</u>	<u>**0.006**</u>
	minIL+trie	6.6	0.371
	MinSearch	4.3	0.389
	Bed-tree	4.8	3.208
	HS-tree	4.4	7.007
UNIREF	minIL	<u>**0.84**</u>	<u>**0.006**</u>
	minIL+trie	2.2	0.297
	MinSearch	3.6	0.019
	Bed-tree	4.8	1.028
	HS-tree	-	-
TREC	minIL	<u>**1.2**</u>	<u>**0.016**</u>
	minIL+trie	1.9	0.339
	MinSearch	5.2	1.477
	Bed-tree	5.1	39
	HS-tree	-	-

vary the value of $\gamma \in (0, 1)$ to simplify the tuning in practice. The reason we set ε in this way is that MinCompact produces pivots from $2^l - 1$ different intervals, and the average length of the interval is $\frac{n}{2^l-1}$. So ε needs to satisfy that $2\varepsilon n < \frac{n}{2^l-1}$, i.e., $\varepsilon < \frac{1}{2(2^l-1)}$. In the experiments, l and γ are always feasible when we set $l \leq 6$ and $\gamma \leq 0.5$. The settings are shown in Table V. The default values of l are 4, 4, 5 and 5 on DBLP, READS, UNIREF and TREC, respectively. The default value of γ is 0.5 on all datasets, and the default value of t is 0.15. To achieve a perfect accuracy (> 0.99), α is determined by the cumulative probability $\mathcal{P}_\alpha$, that is $\mathcal{P}_0 + \mathcal{P}_1 + ... + \mathcal{P}_\alpha > 0.99$. Note that α is data independent. Table VI shows a brief selection of α with different threshold factor t and parameter
l. For example, when $l = 3$ and $t = 0.09$, we select $\alpha = 3$ to achieve an accuracy $\mathcal{P} = 0.998 > 0.99$.

Effectiveness of l. We conduct the experiments by varying l to find the best l for each dataset and to explore the effect of l on query time. Table VIII shows the query time of minIL with different l when $t = 0.15$ over four datasets. Since the query time of minIL is insensitive to t, we avoid to display the results of the other values of t. We observe that, as the average string length of DBLP and READS is small, the value of l can not be larger than 4 and 5, respectively. On DBLP dataset, with the increase of l, the running time drops rapidly. On READS and UNIREF datasets, the results are similar to the results on DBLP. With the increase of l, the running time first drops and then keeps stable with a slight increase. On TREC

TABLE VIII
QUERY TIME WITH DIFFERENT l

Dataset	$l = 2$	$l = 3$	$l = 4$	$l = 5$	$l = 6$
DBLP	28ms	21ms	3ms	-	-
READS	26ms	23ms	6ms	6ms	-
UNIREF	22ms	13ms	6ms	6ms	7ms
TREC	16ms	17ms	17ms	16ms	16ms

dataset, the running time has a different trend, the times almost remains the same when l changes. The results of running time on the first three datasets is reasonable. The query time is mainly determined by the verification phase, where the time of searching on the index takes a small part. The smaller l is, the larger the distortion of the compacted string is, which may result in more candidates to be verified. Meanwhile, the increase of index size, when l increases, has only a little effect on the query time. The result of running time on TREC is different. We infer that the number of candidates changes little when l changes, so the running time is stable.

Effectiveness of ε and α. We conduct the experiments by varying ε and α to explore the effects on the number of candidates. ε is controlled by γ as mentioned in Sec. VI-B, and we set γ = 0.3, 0.4, 0.5, 0.6 and 0.7, respectively. Fig. 7 illustrates the comparison on the number of candidates when varying γ and α over UNIREF and TREC datasets. Figs. 7 (a) and (b) show the distributions of the numbers of candidates with different α. The y-coordinate represents the numbers of sketch strings found in the index when α equals to a certain value. Figs. 7 (c) and (d) show the cumulative numbers of candidates. The value of the y-coordinate represents the total numbers of sketch strings found in the index when α is no larger than a certain value. For example, when $\alpha = 6$ and $\gamma = 0.5$ in (c), the cumulative number is the summation of all candidates when $\gamma = 0.5$ and $\alpha \leq 6$ in (a). We observe that the distributions of the numbers of candidates in (a) and (b) approximate a normal distribution. When γ varies, the position of the peak shifts. The plots in (c) and (d) are the cumulative distributions of plots in (a) and (b). The cumulative number is the number of candidates that need to be verified, so the cumulative number directly impacts the algorithm efficiency. We observe in (c) and (d), when α increases, the cumulative number first goes up smoothly, then it increases rapidly approaching the maximum value, and at last it reaches the maximum value slowly. In addition, the smaller γ is, the later the curve goes up rapidly. According to the above observations, we prefer to choose a small ε and γ when α is chosen. However, an extremely small ε may limit the selection of l and exacerbate the string shift issue. Therefore, there is a trade-off to choose a proper ε.

C. Comparison On Query Time (Q2)

We report on the query performance of minIL+trie and minIL under the default settings and the comparison of query time with competing algorithms.

Fig. 8 reports the average query time of the algorithms as a function of the threshold factor t where $t = \frac{k}{|q|} \in (0, 1)$. It is clear that our minIL performs the best, and Bed-tree always performs the worst. Although HS-tree achieves the best performance on DBLP when t is small, its average query time increases significantly a nd e xceeds t he q uery t ime of minIL when t increases, and its performance is even worse than Bed-tree on READS when t increases. MinSearch and our proposed methods perform better than Bed-tree on all datasets and better than HS-tree on READS over an order of magnitude, while minIL always performs the best. The performance of minIL+trie and MinSearch are similar on READS and TREC, and minIL+trie performs better than MinSearch on DBLP while the opposite is true on UNIREF. In addition, minIL is insensitive to the threshold. When t increases, the average query time of minIL does not increase obviously. Table VII presents the results of query time when $t = 0.15$. It shows that minIL can speed up by at least 3.6, 36.7, and 2.3 times over the competitors. In summary, minIL and MinSearch have the high efficiency a nd stability over all datasets, and HS-tree is more applicable on small datasets with short strings. Bed-tree has a relatively stable performance but a low efficiency.

It is worth noting that although minIL+trie offers no improvements compared with minIL on the query time in most experimental results, it provides better query performance in some cases. Specifically, the time cost of minIL+trie can be smaller than that of minIL, i.e., $O(\sigma^d) < O(LN/|\Sigma|)$, especially when N is large and d is small, where d $(\alpha < d < L)$ is the average depth of the searching on the tree. In these cases, the trie-based index enables to outperform minIL. For example, in Fig. 8 on DBLP when t is small, minIL+trie performs better than minIL.

D. Comparison On Memory Usage (Q3)

Table VII shows the comparison of memory usage of the algorithms. The memory usage of minIL is related to parameters l and ε that are set to the default values. The observation is that our minIL is clearly the best on all datasets that has the smallest memory usage. The memory usages of the other algorithms are 3.2-15 times larger than that of minIL. For example on DBLP, the memory usages are 0.52GB, 1.5GB, 1.7GB, 4.8GB, and 7.8GB for all algorithms, respectively. As mentioned before, the memory usages on UNIREF and TREC of HS-tree exceed our computer's limit, which is larger than 32GB. In this case, it is over 30 times larger than the memory usage of minIL. We can observe that the memory usage of minIL+trie is low on most datasets, but it is the largest on READS. The reason for the difference is that the dictionary size of READS is much larger than the other datasets. It is well known that a large dictionary size has a significant negative impact of a trie-based index. The results also indicate that minIL+trie is more suitable for datasets with small dictionary size. But minIL does not have such limitation. We can draw the conclusion that minIL is a simple and small index method for string similarity search with edit distance.

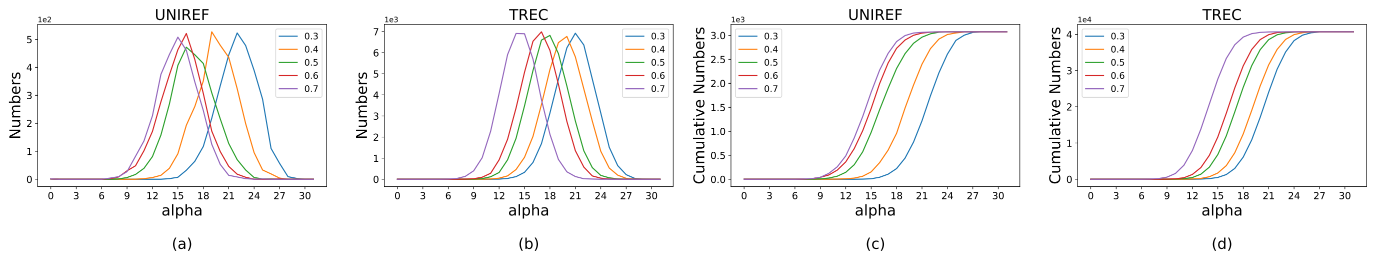

Fig. 7. Number of Candidates with Different ε

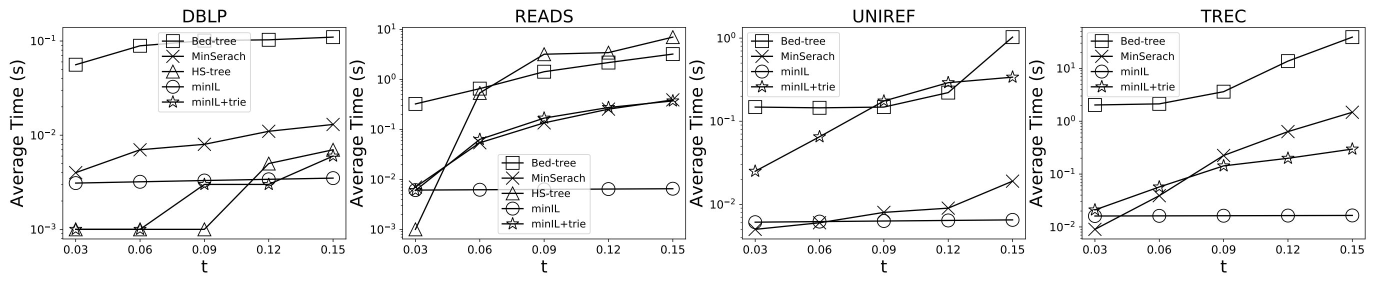

Fig. 8. Average Query Time with Different t

It has a significant advantage in handling large datasets with long strings.

E. Evaluation of the Optimizations for String Shift (Q4)

We use a synthetic dataset to evaluate the effectiveness of the proposed optimizations for string shift. The synthetic dataset only contains strings with string shift at the beginning or at the end of the query string. We generate the data as follows: 1) First, we randomly generate a query string with length of 1200, 2) Given a shift length factor η, we fill or truncate the query string at the beginning or at the end of it with $\tilde{s}$ characters, where $\tilde{s}$ is a random value in $[0, \eta|q|]$, 3) We generate 100K strings according to step 2) for different η, respectively. We set $\eta = 0.05, 0.1, 0.15, 0.2$.

Fig. 9 presents the average accuracy of the algorithms with different shift lengths. NoOpt is the proposed method minIL without any optimization. Opt1 is minIL with the first optimization described in Sec. III, and we use 2ε at the first recursion. Opt2 is minIL with both two proposed optimizations (the second optimization is introduced in Sec. V). We set $m = 1$ throughout the experiments. The accuracy is defined as ratio of the number of candidate strings to the dataset cardinality. From the results, we observe that the original method has difficulty in dealing with the extreme string shift, its accuracy is always less than 0.1. But with the first optimization, the accuracy of the method improves up to 0.7 when shift length is $0.05|q|$, and then it decreases quickly with the increase of the shift length. The second optimization has a significant effect. It helps the method achieve a perfect accuracy when shift length is small, and keeps a high accuracy when shift length increases. When the accuracy of Opt2 falls to a low level, for example when the shift length is $0.2|q|$, it indicates that the variants of the query of the optimization

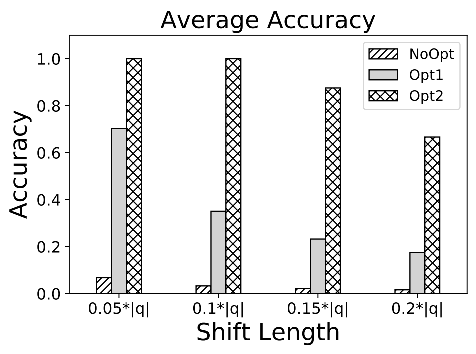

Fig. 9. Average Accuracy with Different Shift Length

cannot cover all string shift possibilities. In this case, we only needs to increase m to solve the problem.

VII. Related Work

In this section we review the related work on similarity search and join problem.

Similarity Search. Similarity search with edit distance has been studied extensively in the literature [7], [12], [13], [15], [16], [19], [24], [27], [28]. Many of existing studies are devised to address the threshold-based similarity search problem, while the others aim to support the top-k similarity search. For threshold-based similarity search, Li et al. [12] improved the count filter [16] based method and designed several list-merge algorithms to tackle the problem. Li et al. [13] proposed variable length grams based method which advisably chooses high-quality grams of different lengths to address the problem. Qin et al. [15] proposed asymmetric signature schemes named QChunk and developed several dynamic programmingbased candidate pruning methods. Wang et al. [19] proposed an adaptive prefix filtering framework and a cost model to judiciously select an appropriate prefix for each object to speed up the performance. Deng et al. [7] proposed a pivotal prefix filter based method which significantly reduces the unnecessary signatures. Zhang et al. [27] proposed a local hash minima method to extract common substrings between similar strings. For top-k similarity search, Yang et al. [23] utilized an adaptive q-gram selection and several efficient strategies to explore the problem. Deng et al. [8] used a trie index to share common prefixes and proposed a range-based algorithm by grouping specific entries to avoid duplicated computations. Wang et al. [21] designed a novel filter-and-refine pipeline approach that used long but approximate n-gram matches for candidates pruning. Zhang et al. [28] developed Bed-tree utilizing B+-tree to index strings and Yu et al. [24] devised a unified framework using hierarchical segment tree (HS-tree) to support both threshold and top-k similarity search.

Most existing methods have two limitations. First, many algorithms using q-gram based signatures have poor pruning power, since the value q is typically very small to avoid missing results, and a small q has limited pruning power. Second, most existing algorithms do not perform well on long strings compared with short strings. Long strings may cause massive redundancy indexes that reduce the search efficiency, and signature-based methods cannot capture long signatures .

Similarity Join. Similarity join, i.e., given two string collection to find all similar string pairs, is a closely related problem. The problem has been studied extensively as well [14], [15], [18]–[20], [22], [24]–[26]. Yu et al. [24] provided a comprehensive survey of similarity join. Wandelt et al. [17] reported some state-of-the-art methods of string similarity search and join. Bayardo et al. [2] first proposed the prefix filter-based method for finding similar pairs of vectors. Xiao et al. [22] proposed the Ed-join with perspective of investigating mismatching pairs to improve the prefix filter. Ciaccia et al. [6] proposed M-tree that organizes and searches large data sets from a generic metric space. Arasu et al. [1] devised enumeration-based algorithm to produce exact answers with precise performance guarantees. Wang et al. [18] utilized the trie structure to efficiently find the similar string pairs based on subtrie pruning to support similarity join of short strings. Li et al. [14] proposed Pass-Join that partitions a string into a set of segments with efficient substring selection. Adapt [19], the adaptive prefix filtering framework, and QChunk [15] can also be applied to similarity join problem. Wang et al. [20] proposed VChunk extracting non-overlapping substrings from strings with a class of new chunking schemes. Zhang et al. [25] proposed EmbedJoin which integrated the CGK-embedding [4] and LSH techniques [10] to handle the join problem. Zhang et al. [26] proposed string partition based local hash minima method to achieve a perfect accuracy of the join results.

VIII. Conclusion

In this paper, we study the threshold-based string similarity search problem with the edit distance. We present a MinCompact algorithm to construct sketch representations for strings and propose a small and simple index, minIL, to store and search the sketch strings. Benefiting from the sketch representations and the learned index technique, minIL outperforms the existing methods on large datasets with long strings and substantially reduces the space consumption. We consider that minIL has an advantage for solving the threshold-based similarity query with edit distance. It works well on different datasets and it is applied to each string independently while implicitly aligning the strings. In addition, although minIL is an approximate method, it enable to achieve a perfect accuracy by adjusting parameters.

In the future work, we plan to study how to apply the technique of minIL for other important and relevant problems, such as the similarity join and top-k similarity search.

References

[1] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929. ACM, 2006.

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

[3] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *J. Comput. Syst. Sci.*, 60(3):630–659, 2000.

[4] Diptarka Chakraborty, Elazar Goldenberg, and Michal Koucký. Streaming algorithms for embedding and computing edit distance in the low distance regime. In *STOC*, pages 712–725. ACM, 2016.

[5] Moses Charikar, Ofir Geri, Michael P. Kim, and William Kuszmaul. On estimating edit distance: Alignment, dimension reduction, and embeddings. In *ICALP*, volume 107 of *LIPIcs*, pages 34:1–34:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[6] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, pages 426–435. Morgan Kaufmann, 1997.

[7] Dong Deng, Guoliang Li, and Jianhua Feng. A pivotal prefix based filtering algorithm for string similarity search. In *SIGMOD Conference*, pages 673–684. ACM, 2014.

[8] Dong Deng, Guoliang Li, Jianhua Feng, and Wen-Syan Li. Top-k string similarity search with edit-distance constraints. In *ICDE*, pages 925–936. IEEE Computer Society, 2013.

[9] Paolo Ferragina and Giorgio Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.*, 13(8):1162–1175, 2020.

[10] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529. Morgan Kaufmann, 1999.

[11] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *SIGMOD Conference*, pages 489–504. ACM, 2018.

[12] Chen Li, Jiaheng Lu, and Yiming Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266. IEEE Computer Society, 2008.

[13] Chen Li, Bin Wang, and Xiaochun Yang. VGRAM: improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, pages 303–314. ACM, 2007.

[14] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. PASS-JOIN: A partition-based method for similarity joins. *Proc. VLDB Endow.*, 5(3):253–264, 2011.

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

[16] Sunita Sarawagi and Alok Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, pages 743–754. ACM, 2004.

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

[18] Jiannan Wang, Guoliang Li, and Jianhua Feng. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *Proc. VLDB Endow.*, 3(1):1219–1230, 2010.

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

[20] Wei Wang, Jianbin Qin, Chuan Xiao, Xuemin Lin, and Heng Tao Shen. Vchunkjoin: An efficient algorithm for edit similarity joins. *IEEE Trans. Knowl. Data Eng.*, 25(8):1916–1929, 2013.

[21] Xiaoli Wang, Xiaofeng Ding, Anthony K. H. Tung, and Zhenjie Zhang. Efficient and effective KNN sequence search with approximate n-grams. *Proc. VLDB Endow.*, 7(1):1–12, 2013.

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

[23] Zhenglu Yang, Jianjun Yu, and Masaru Kitsuregawa. Fast algorithms for top-k approximate string matching. In *AAAI*. AAAI Press, 2010.

[24] Minghe Yu, Jin Wang, Guoliang Li, Yong Zhang, Dong Deng, and Jianhua Feng. A unified framework for string similarity search with edit-distance constraint. *VLDB J.*, 26(2):249–274, 2017.

[25] Haoyu Zhang and Qin Zhang. Embedjoin: Efficient edit similarity joins via embeddings. In *KDD*, pages 585–594. ACM, 2017.

[26] Haoyu Zhang and Qin Zhang. Minjoin: Efficient edit similarity joins via local hash minima. In *KDD*, pages 1093–1103. ACM, 2019.

[27] Haoyu Zhang and Qin Zhang. Minsearch: An efficient algorithm for similarity search under edit distance. In *KDD*, pages 566–576. ACM, 2020.

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