The VLDB Journal
DOI 10.1007/s00778-016-0449-y

REGULAR PAPER

A unified framework for string similarity search with edit-distance constraint

Minghe Yu[1] · **Jin Wang**[1] · **Guoliang Li**[1] · **Yong Zhang**[1] · **Dong Deng**[1] · **Jianhua Feng**[1]

Received: 10 February 2015 / Revised: 25 September 2016 / Accepted: 24 November 2016

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 for either the former or the latter; most of them cannot 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 develop effective pruning techniques to further improve the performance. To support large data sets, we extend our techniques to support the disk-based setting. Experimental results on real-world data sets show that our method achieves high performance on the two problems and outperforms state-of-the-art algorithms by 5–10 times.

Keywords Similarity search · Edit distance · Top-k · Disk-based method · Partition

✉ Guoliang Li
liguoliang@tsinghua.edu.cn

Minghe Yu
yumh12@mails.tsinghua.edu.cn

Jin Wang
wangjin12@mails.tsinghua.edu.cn

Yong Zhang
zhangyong05@tsinghua.edu.cn

Dong Deng
dd11@mails.tsinghua.edu.cn

Jianhua Feng
fengjh@tsinghua.edu.cn

[1] Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China

1 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 [2,20]. 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 [8,19,23,28]. In this paper, we focus on using edit distance to quantify string similarity.

Existing studies on the threshold-based similarity search problem [20,21,28,37] 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 [8,43,45] are inefficient for threshold-based search because they cannot make full use of the given threshold to do pruning. In

Published online: 17 December 2016

addition, in the big data era, many data sets are rather large and existing methods usually adopt an in-memory-based setting and cannot support large data sets that cannot be loaded into the memory. For example, all existing algorithms cannot handle the PubMed data set (with 6 GB). To summarize, existing methods have two limitations. First, they are efficient either for threshold-based search or for top-k search and most of them cannot efficiently support both of the two problems. Second, they cannot support large data sets. Thus, it calls for a unified framework to efficiently support the two variants of string similarity search, which can be easily extended to support the disk-based setting.

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. Moreover, our method can be easily extended to support the disk-based setting. We study how to effectively organize the index on disks and utilize the index to efficiently answer queries. Experimental results on real data sets 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 extend our techniques to support disk-based setting and propose disk-based indexes and algorithms.
- We have conducted an extensive set of experiments on both in-memory and disk-based settings. Experimental results show our method achieves high performance on both threshold-based similarity search and top-k similarity search and outperforms state-of-the-art algorithms by 5–10 times.

The rest of the paper is organized as follows. We formulate our problem and review related works in Sect. 2. We introduce our hierarchical index in Sect. 3. The HS-Search algorithm is proposed to support threshold-based similarity search in Sect. 4, and the HS-Topk algorithm is presented to support top-k similarity search in Sect. 5. We discuss disk-based indexes and algorithms in Sect. 6. Experimental results are reported in Sect. 7. We conclude in Sect. 8.

2 Preliminaries

In this section, we first formulate the problem in Sect. 2.1 and then review related works in Sect. 2.2.

2.1 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 on a single letter) 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 1. 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 1 A string collection

ID	String	Length
s_1	Brother	7
s_2	Brothel	7
s_3	Broathe	7
s_4	Breathe	7
s_5	Brecher	7
s_6	Brachels	8
s_7	Swingable	9
s_8	Deduction	9
s_9	Abna levina	11
s_{10}	Christopher swenson	19

2.2 Related works

2.2.1 Threshold-based similarity search

There are many similarity search algorithms [6,21,28,37,45]. 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. [29] proposed count filter based on n-grams. Li et al. [20] extended the count filter and developed several list-merge algorithms to improve the performance. Wang et al. [37] proposed an adaptive prefix filtering framework to improve the performance. Zhang et al. [45] proposed B^{ed}-tree which utilized B^+-tree to index strings. Li et al. [21] used variable length grams as signatures to support string similarity search. Qin et al. [28] devised an asymmetry signature. Hadjieleftheriou et al. [16] 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; but 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 expensive as they require to perform the search algorithms many times.

2.2.2 Top-k similarity search

There have already been some studies focusing on top-k similarity search. Yang et al. [43] proposed a 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. [8] 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. But for long strings, there are fewer common prefixes and the efficiency will be limited. B^{ed}-tree [45] 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. [39] 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.

Different from our conference version [34], we make the following new contributions. First, we extend our techniques to support disk-based setting and propose new disk-based indexes and algorithms. Section 6 is newly added. Second, we implement our disk-based methods and compare with state-of-the-art techniques. Section 7.4 is newly added. Third, we formally prove all the lemmas.

2.2.3 Similarity join

There are many studies on string similarity join [2,4,9–12,23,25,35,36,41,42]. Given two string sets, similarity join finds all similar string pairs. An experimental survey is made in [18]. Yu et al. provided a survey [44]. Bayardo et al. [2] proposed the prefix filter-based method for similarity join. Xiao et al. proposed the position filter [42] and mismatch filter [41] to improve the prefix filter. Wang et al. [35] proposed a trie-based method to support similarity join. Li et al. [23] proposed the segment filter with efficient substring selection and verification methods to perform similarity join. Although Adapt [37] and QChunk [28] extended join techniques to support search, they perform much worse than our method (see Sect. 7). Wandelt et al. [33] reported the results of EDBT competition. In this competition, there were 9 teams from different fields turning their algorithms for threshold-based similarity search and joins, including PassJoin [23], Masai [31] and WallBreaker [13] et al, which used different techniques such as the pigeonhole principle, approximate partition and directed acyclic word graph to solve these problems.

In this paper, we extend the segment-based filter in [23] 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. Different from QChunk [28], our algorithm only generates the valid substrings and eliminates the invalid substrings based on the position filtering. QChunk generates all q-grams and then uses a dynamic-programming method to remove the invalid matching.

2.2.4 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 [7]. Wang et al. [38] proposed a neighborhood deletion-based method to solve this problem. Li et al. [22] proposed a unified framework to support approximate entity extraction under various similarity functions. Recently Kim et al. [19] 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. [17] proposed a trie-based structure to support fuzzy prefix search, and Li et al. [24,26] improved it by removing unnecessary nodes. Chaudhuri et al. [5] proposed a similar solution. Xiao et al. [40] extended the neighborhood deletion method to improve the performance of query autocompletion. Ahmadi et al. [1] provided an algorithm Hobbes for aligning short string set which utilized bitvector filter and optimized gram-based method for candidate pruning. Gusfield [15] proposed suffix contraction techniques, Mansour et al. [27] provided a suffix tree construction method to support long strings storage, and they constructed suffix tree to effectively organize the strings. Our method is different from the techniques in [15,27]. Firstly, Gusfield [27] utilizes the suffix tree to compress the strings (or segments) while our method does not compress the segments and we reduce the size of the inverted lists. The basic idea is that each lower-level inverted list is a sublist of the upper-level inverted list, and we utilize this idea to reduce the space. Secondly, Mansour et al. [27] focus on constructing the suffix tree, while our method emphasizes on compressing the inverted lists and reducing the number of disk IOs. Thirdly, they only focus on reducing the space but do not discuss how to use the index to efficiency support queries. Our objective is to improve the query performance by organizing the inverted lists.

3 The hierarchical segment tree

This section introduces 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 S by length;
3     for l = l_min to l_max do
4         Calculate the maximum level L = ⌊log2 l⌋;
5         Generate sets S_l^{1,1}, S_l^{1,2}, nodes n_l^{1,1}, n_l^{1,2}, indexes L_l^{1,1},
          L_l^{1,2};
6         for i = 2 to L do
7             for j = 1 to 2^i do
8                 Generate sets S_l^{i,2j-1}, S_l^{i,2j}, nodes n_l^{i,2j-1}, n_l^{i,2j},
                  indexes L_l^{i,2j-1}, 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 tree 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 1. We first group them by length and then iteratively build HS-Tree for each length l. Take group $\mathcal{S}_7$ in Fig. 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

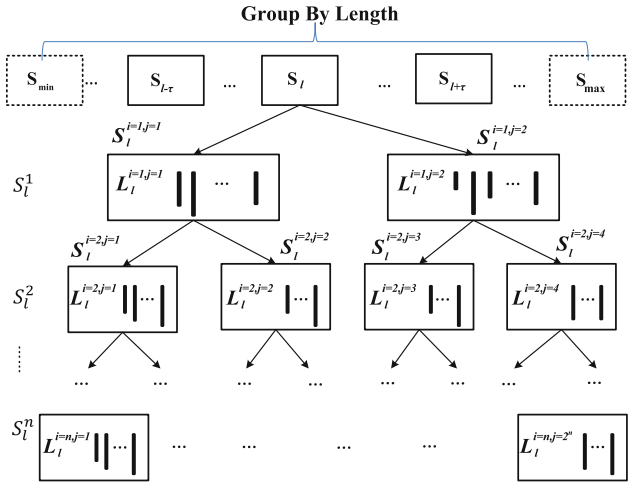

Fig. 1 The HS-Tree index

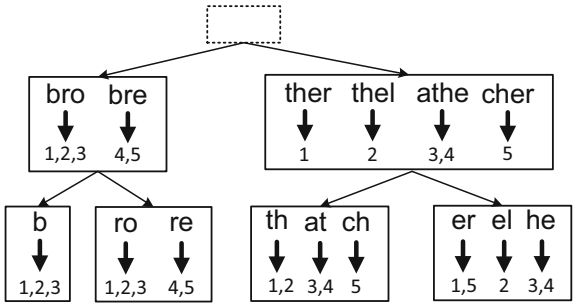

Fig. 2 An HS-Tree example for $\mathcal{S}_7$

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 to s_5 to build the index.

Space complexity We analyze the space complexity of the HS-Tree. Suppose $l_{\min}$ and $l_{\max}$ are the minimum and maximum string length, respectively. And $n_{\min}$ and $n_{\max}$, respectively, denote the minimum and maximum number of strings in the groups. 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. For the inverted lists in the HS-Tree, as each internal segment set contains n segments, where n is the total number of strings in $\mathcal{S}_l$, and the size of inverted lists in the $\mathcal{S}_l$ is $n_l + n_l * 2 + \cdots + n * 2^{\lfloor \log l \rfloor} = \mathcal{O}(n_l * l)$. Obviously, each string is contained in at most $\mathcal{O}(l)$ inverted lists, and thus the space complexity of the HS-Tree is $\mathcal{O}(\sum_{l=l_{\min}}^{l=l_{\max}} \sum_{n=n_{\min}}^{n=n_{\max}} l * (|\mathcal{S}_l| + n))$, which is exactly the total number of characters in all input strings.

From Sects. 3 to 5, we focus on in-memory setting. Our method can also be extended to support disk-based setting, and the details are shown in Sect. 6.

4 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 Sect. 4.1). We first introduce a filter-verification framework and then develop novel techniques to improve both the filter step (see Sect. 4.2) and the verification step (see Sect. 4.3).

4.1 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, it is easy to prove that if s is similar to q, q must contain at least $2^i - \tau$ segments of s. 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.

Example 3 Consider a query q={"swaingbe"}, the data string s_7 = {swingable}, s_8 = {deduction}, and $\tau = 2$. In level 2, the 4 segments of s_7 are {"sw","in","ga","ble"}, and q has $2^2 - 2 = 2$ common substrings "sw" and "in" with s_7, and thus, s_8 is a candidate for query q and $\tau = 2$. On the contrary, the 4 segments of s_8 = {deduction} are {"de","du","ct","ion" }. And s_8 has no matched segments with q. So we can safely prune s_8.

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 levels 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

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

```
Input: S: The string set; q: The query string
       τ: The given edit-distance threshold
Output: R = {(s ∈ S) | ED(s, q) ≤ τ}
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          for s where N_i(s, q) ≥ 2^i − τ do
10             if ED(s, q)≤τ then R=R∪{s};
11 end
```

Sect. 4.2. 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.

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 4). 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 10). To improve the performance, we design efficient techniques to reduce the substring-set size $\mathcal{W}(q, \mathcal{L}_l^{i,j})$ in Sect. 4.2 and improve the verification cost in Sect. 4.3.

Example 4 Consider the string set in Table 1. Suppose we have built the HS-Tree index as shown in Fig. 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 $\mathsf{ED}(s, q) \leq 2$, 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 to s_5 in the inverted list of "b" share a common segment with q, we increase all of their common segment number by 1. Then, we check $\mathcal{L}_7^{2,2}$ and string q matches one of them, "re", which inverted list contains s_4 and s_5. We increase $\mathcal{N}_2(s_4, q)$ and $\mathcal{N}_2(s_5, q)$ by 1. 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, s_2, s_4 and s_5 have two matched segments with q, so we verify them and get $\mathsf{ED}(q, s_1) = 2$, $\mathsf{ED}(q, s_2) = 3$, $\mathsf{ED}(q, s_4) = 3$ and $\mathsf{ED}(q, s_5) = 2$. We put s_1 and s_5 into the result. As $\mathcal{N}_2(s_3, q) = 1 < 2$, we safely prune s_3. Then, we perform similar procedure on $\mathcal{L}_8^{i,j}$,$\mathcal{L}_9^{i,j}$.

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 Sect. 4.2. 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 $\mathcal{O}(\tau * \min(|q|, |s|))$ [23], and we will further improve the verification cost in Sect. 4.3.

4.2 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 $1 \leq j \leq 2^i$; and (2) remove invalid segment matchings.

Reduce $\mathcal{W}(q, \mathcal{L}_l^{i,j})$. It is obvious smaller $\mathcal{W}(q, \mathcal{L}_l^{i,j})$ 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_7="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 $\mathtt{Pos}_l^{i,j}$ and its length is $\mathtt{Len}_l^{i,j}$. We can easily get the lower bound of start positions of substrings in q, denoted as $\mathsf{LB} = \max(1, \mathtt{Pos}_l^{i,j} - \lfloor \tau - \Delta \rfloor)$, and the upper bound, denoted as $\mathsf{UB} = \min(|q| - \mathtt{Len}_l^{i,j} + 1, \mathtt{Pos}_l^{i,j} + \lfloor \tau + \Delta \rfloor)$. We only need to check the substrings starting within the range [LB, UB]. Moreover, by looking both from the left-side and right-side perspective [23], we can further reduce the value of LB and UB to $\mathsf{LB} = \max(1, \mathtt{Pos}_l^{i,j} - (j -$

$1), \mathtt{Pos}_l^{i,j} + \Delta - (\tau + 1 - j))$ and $\mathsf{UB} = \min(|s| - \mathtt{Len}_l^{i,j} + 1, \mathtt{Pos}_l^{i,j} + (j-1), \mathtt{Pos}_l^{i,j} + \Delta + (\tau + 1 - j))$. Thus $\mathcal{W}(q, \mathcal{L}_l^{i,j}) = \{q[\mathtt{Pos}_l^{i,j}, \mathtt{Len}_l^{i,j}]\}$ where $q[\mathtt{Pos}_l^{i,j}, \mathtt{Len}_l^{i,j}]$ is the substring of q with start position $\mathtt{Pos}_l^{i,j} \in [\mathsf{LB}, \mathsf{UB}]$ and length $\mathtt{Len}_l^{i,j}$. The correctness is stated in Lemma 1.

Lemma 1 *Given a query string q and a threshold τ, using the set $\mathcal{W}(q, \mathcal{L}_l^{i,j}) = \{q[\mathtt{Pos}_l^{i,j} \in [\mathsf{LB}, \mathsf{UB}], \mathtt{Len}_l^{i,j}]\}$ to find matching candidates, our method will not miss any result.*

Proof For a string s that is similar to q, any transformation $\mathcal{T}$ from q to s satisfies $|\mathcal{T}| \leq \tau$. q must have at least $x = 2^i - \tau$ common substrings with s in the transformation. Suppose the last segment in s which matches a substring of q is s_k in $\mathcal{T}$, and the start point of the matched substring in q is p_k. Without loss of generality, we only need to prove that $p_k \in [\mathsf{LB}, \mathsf{UB}]$. To this end, we need to prove $p_k \in [\mathtt{Pos}_l^{i,j} - (k-1), \mathtt{Pos}_l^{i,j} + (k-1)]$ and $p_k \in [\mathtt{Pos}_l^{i,j} + \Delta - (\tau + 1 - k), \mathtt{Pos}_l^{i,j} + \Delta + (\tau + 1 - k)]$.

First we prove $p_k \in [\mathtt{Pos}_l^{i,j} - (k-1), \mathtt{Pos}_l^{i,j} + (k-1)]$ by contradiction. Suppose $p_k \notin [\mathtt{Pos}_l^{i,j} - (k-1), \mathtt{Pos}_l^{i,j} + (k-1)]$. The set of matched segments is $\mathcal{M}_i(s, q) = \{m_1, m_2, \cdots, m_x\}$, and the set of unmatched segments of query q (string s) is $Q = \{q_1, q_2, \cdots, q_{x+1}\}$ ($S = \{s_1, s_2, \cdots, s_{x+1}\}$), where $q_u(s_u)$ is the substring between m_u and m_{u+1} for $u \in [1, x]$ and $q_{x+1}(s_{x+1})$ is the substring after p_x (s_x). We have $\sum_{u=1}^{x} \mathsf{ED}(s_u, q_u) \geq |p_k - \mathtt{Pos}_l^{i,j}| \geq k$. As $\mathcal{T}$ transforms q_u to s_u and matches s and q in m_u ($u \in [1, x]$), we have $\tau > |\mathcal{T}| \geq \sum_{u=1}^{x+1} \mathsf{ED}(s_u, q_u) \geq k + \sum_{u=1}^{x} \mathsf{ED}(s_u, q_u)$. Thus, $\mathsf{ED}(s_{x+1}, q_{x+1}) \leq \tau - k$. On the other hand, as there are $\tau + 1 - k$ segments in s_{x+1}, there must exist a segment in s_{x+1} that matches a substring in q_{x+1}. This contradicts with the assumption that s_k is the last matched segment. Thus, $p_k \in [\mathtt{Pos}_l^{i,j} - (k-1), \mathtt{Pos}_l^{i,j} + (k-1)]$.

Similarly we can prove $p_k \in [\mathtt{Pos}_l^{i,j} + \Delta - (\tau + 1 - k), \mathtt{Pos}_l^{i,j} + \Delta + (\tau + 1 - k)]$. Therefore, using the set $\mathcal{W}(q, \mathcal{L}_l^{i,j}) = \{q[\mathtt{Pos}_l^{i,j} \in [\mathsf{LB}, \mathsf{UB}], \mathtt{Len}_l^{i,j}]\}$ to find matching candidates, our method will not miss any result. □

To identify string s with $\mathcal{N}_i(s, q) \geq 2^i - \tau$, we use the list-merge algorithm [20] 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: HS-Search-Filter()

```
1 begin
    // Replace Line 10 of Algorithm 2 with
       the following
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   if D[N_i(s, q)] ≥ 2^i − τ then
6     if ED(s, q)≤τ then R=R∪{s};
7 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:

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

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 $\mathcal{N}_3(s,q) = 3 \geq 2^3 - 5$. 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 matched segments. As $2 < 2^3 - 5 = 3$, we safely prune s.

4.3 Improving the verification step

In our HS-Search algorithm, after generating the candidate strings, we verify whether their real edit distances to the 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 $s[1,j]$ [30]. 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 $\tau \cdot \min(|q|,|s|)$ [23]. These algorithms can also do early termination when the values in a row are all larger than τ to further improve the time complexity.

Next we discuss how to effective verify whether s and q are similar. Consider the value of $\mathcal{N}_i(s,q)$. (1) If $\mathcal{N}_i(s,q) < 2^i - \tau$, we can safely prune s, because s does not share enough common segments with q; (2) If $\mathcal{N}_i(s,q) \geq 2^i - \tau$, we need to verify s. Note there could be multiple alignments between s and q. If $\mathcal{N}_i(s,q) \geq 2^i - \tau$, there are $\binom{\mathcal{N}_i(s,q)}{2^i-\tau}$ possible alignments (by selecting $2^i - \tau$ matched segments from $\mathcal{N}_i(s,q)$ segments). (2.1) If $\binom{\mathcal{N}_i(s,q)}{2^i-\tau}$ is very large, i.e., $\binom{\mathcal{N}_i(s,q)}{2^i-\tau} \geq |s|$, it is expensive to verify every alignment, and thus we directly verify the pair using the dynamic-programming algorithm. (2.2) If $\binom{\mathcal{N}_i(s,q)}{2^i-\tau}$ is small, i.e., $\binom{\mathcal{N}_i(s,q)}{2^i-\tau} < |s|$, we propose an efficient verification method as follows. We first enumerate every possible alignment, i.e., checking each subset $M_i(s,q)$ of $\mathcal{M}_i(s,q)$ with $2^i - \tau$ segments. For each alignment $M_i(s,q)$, we propose an extension-based method. As discussed in Sect. 4.1, given a candidate string s, $y = \mathcal{N}_i(s,q)$, and 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 $M_i(s,q) = \{m_1, m_2, \ldots, m_y\}$, and the set of unmatched segments of query q (string s) is $Q = \{q_1, q_2, \ldots, q_{y+1}\}$ ($S = \{s_1, s_2, \ldots, s_{y+1}\}$), 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; 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 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:

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

We propose an early-termination technique (Lemma 2).

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

Proof We prove it by contradiction. Consider any transformation $\mathcal{T}$ from s to q with $|\mathcal{T}| \leq \tau$ edit operations. $|\mathcal{T}| = \sum_{j=1}^{y+1} \mathsf{ED}(s_j,q_j) \leq \sum_{j=1}^{y+1} \tau_j = p_1 + p_2 - p_1 - 1 + \cdots + 2^i - p_y = 2^i - y = \tau$. As there are exactly $y = 2^i - \tau$ common segments, the $|\mathcal{T}|$ edit operations are distributed in the τ mismatched segments. If $\exists j \in (1, y+1)$ satisfying $\mathsf{ED}(q_j,s_j) > \tau_j$, there must exist a segment in s_j with more than one edit operation. Then, there are at most $\tau - 2$ edit operations and at least $\tau - 1$ mismatched segments. According to the pigeon hole principle, there must be at least one matched segment. This contradicts with the condition that there are exactly $x = 2^i - \tau$ common segments. Thus, for $j \in (1, y+1)$ and $\mathsf{ED}(q,s) \leq \tau$, we have $\mathsf{ED}(q_j,s_j) \leq \tau_j$. Similarly we can reach the same conclusion when $j = 1$ or $j = y+1$. □

Based on Lemma 2, we can devise the verification algorithm MultiExtension as shown in Algorithm 4. If there are multiple possible alignments (i.e., $\binom{\mathcal{N}_i(s,q)}{2^i-\tau} \geq |s|$), we directly verify it (lines 2-3); otherwise, we can use the length-aware method to efficiently verify each alignment $M_i(s,q)$, which is a subset of $\mathcal{M}_i(s,q)$ with $2^i - \tau$ elements (lines 5-13). If the edit distance between each segment is not larger than the threshold τ_j (i.e., $\mathsf{ED}(q_j,s_j) \leq \tau_j$), s is an answer and we add it to the result (lines 9-11); otherwise, we skip the alignment (lines 12-13). Next we walk through the two verification algorithms SingleThreshold and MultiExtension using the following example as shown in Fig. 3.

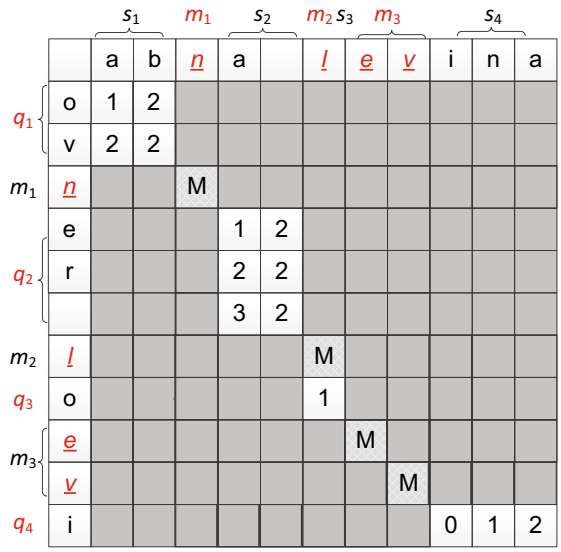

Fig. 3 The MultiExtension verification ($y = 4$)

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

```
1  begin
       // Replace Line 6 of Algorithm 3 with the
          following
2      if (C(N_i(s,q), 2^i − τ)) ≥ |s| then
3          if ED(s, q)≤τ then R=R∪{s};
4      else
5          for each alignment case M_i(s, q) (a subset of M_i(s, q)
           with 2^i − τ elements do
6              Generate sets S and Q based on M_i(s, q);
7              Generate thresholds based on M_i(s, q);
8              Calculate ED(q_j, s_j) with length-aware method;
9              if ∀j ∈ [1, N_i(s, q) + 1], ED(q_j, s_j) ≤ τ_j then
10                 R=R∪{s};
11                 return;
12             else
                   // ∃j, ED(q_j, s_j) > τ_j
13                 break;
14 end
```

Example 6 Consider two strings s_9="abna levina" in Table 2 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 Fig. 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. SingleThreshold continues 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.

5 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 to prune large numbers of dissimilar strings effectively. To this end, we first propose a batch-pruning-based method in Sect. 5.1 and then present an effective pruning strategy in Sect. 5.2. Finally we devise our HS-Topk algorithm in Sect. 5.3.

5.1 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, 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 3. 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

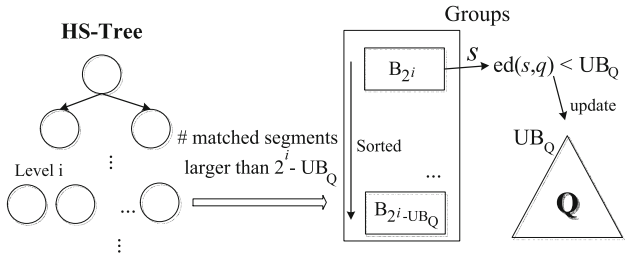

Fig. 4 The batch-pruning-based method

with larger x. These two observations motivate us devise a batch-pruning-based method.

Lemma 3 *If $LB_{\mathcal{B}_x} \geq UB_{\mathcal{Q}}$, strings in $\mathcal{B}_x$ can be pruned.*

Proof According to the definition of $\mathsf{LB}_{\mathcal{B}_x}$, strings in bucket $\mathcal{B}_x$ are with the edit distance larger than $2^i - x$. If $\mathsf{LB}_{\mathcal{B}_x} \geq \mathsf{UB}_{\mathcal{Q}}$, there is no chance for strings in $\mathcal{B}_x$ to pass the verification step. Thus, these strings 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 reduces not only 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 (Fig. 4).

Example 7 Consider a top-2 query q = "`brachers`" on the data set in Table 1. Suppose string s_6 = "`brachels`" is already in $\mathcal{Q}$ as it has a common segment "`brac`" in level 1 with q. $\mathsf{ED}(q, s_4) = 2$. $\mathsf{UB}_{\mathcal{Q}} = \infty$. Then, consider the HS-Tree for strings with length 7 ($\mathcal{S}_7$) in Fig. 2. We start from level 1. As there is no matched segment, we move to level 2. There are four matched segments "`b`,`ch`,`er`, `he`". After calculating occurrence of each string in the inverted lists of these segments, finally we have $\mathcal{N}_2(s_1, q) = 2, \mathcal{N}_2(s_3, q) = \mathcal{N}_2(s_4, q) = 1$ and $\mathcal{N}_2(s_5, q) = 3$. We put s_5 into $\mathcal{B}_3$ and verify $\mathcal{B}_3$. As $\mathsf{ED}(q, s_5)=2 < \mathsf{UB}_{\mathcal{Q}}$, we add s_5 into $\mathcal{Q}$ and update $\mathsf{UB}_{\mathcal{Q}} = 2$. As $2^2 \geq \mathsf{UB}_{\mathcal{Q}} + 1$, the algorithm is terminated and strings in $\mathcal{B}_2 = \{s_1\}$ and $\mathcal{B}_1 = \{s_3, s_4\}$ are pruned.

5.2 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 data set in Table 1. 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 tree 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 3. 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 Sect. 4.3. 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.

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 $\mathcal{W}(q, \mathcal{L}_{|s|}^{r,j})$ for each level r ($1 \leq r \leq n$), when we look for a matched substring for segment j, we just check $\mathcal{W}(q, \mathcal{L}_{|s|}^{r,j})$ and do not need to scan inverted lists any more.

Example 8 Consider s_{10}="`christopher swenson`" and query q="`atrmstophbwcmrense`". The length of s_{10} is 19, so there are 4 levels. Suppose the current threshold $\mathsf{UB}_{\mathcal{Q}}$

Algorithm 5: GREEDYMATCH (s, q, i, τ)

```
Input: s, q: the strings to be verified
       i: level; τ: the current threshold
       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 return False;
11   return True;
12 end
```

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

```
Input: q: the query string; k: the size of result set
       S: The string set
Output: 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
```

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_{10} will be pruned and we do not need to compute its real edit distance to q.

5.3 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 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 Sect. 4.3 (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) (Fig. 5).

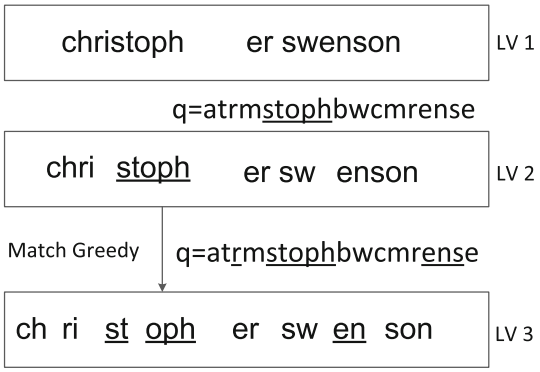

Fig. 5 An example for greedy-match strategy

6 Supporting disk-based settings

In this section, we extend our index and algorithms to support disk-based settings and propose new disk-based indexes and algorithms. For memory-based algorithms, the primary goal is to reduce computational time and thus we adopt a filter-verification framework to avoid expensive computation cost. However, for disk-based settings, we also need to consider the I/O cost and make a tradeoff between computation time and I/O time.

A straightforward way to deploy HS-Tree on disk is to store all inverted lists on the disk and directly use algorithms in Sects. 4 and 5 to support similarity search. However, this method has several weaknesses. First, it needs to access inverted lists in every level and thus accesses disk multiple times, leading to expensive I/O cost. Second, it involves many random accesses, because each query will match multiple segments, which are stored in-consecutively (with their corresponding inverted lists) on disk. To achieve high efficiency of disk-based algorithms, we need to address the following two challenges. The first is to reduce the index size and the second is to avoid random accesses.

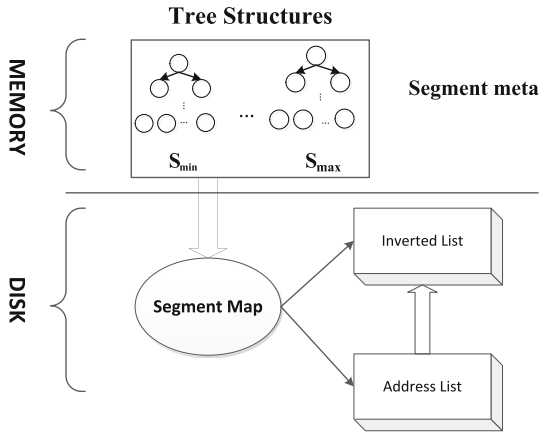

Fig. 6 The architecture of compact HS-Tree index

6.1 Disk-based index: compact HS-Tree

We observe that in the HS-Tree, for $i > 1$, all inverted lists in node $n_l^{i,j}$ are the union of some inverted lists in node $n_l^{i-1,\lceil j/2 \rceil}$. For example, in Fig. 2, the inverted list of "th" in node $n_l^{2,3}$ contains strings s_1 and s_2, which is the union of inverted lists of "ther" and "thel" in node $n_l^{1,2}$. We can utilize this property to reduce the index size of the HS-Tree. Next we formulate this problem.

Definition 3 (Parent Segment) Given a segment w in node $n_l^{i,j}$, a segment w_p in node $n_l^{i-1,\lceil j/2 \rceil}$ is a *parent segment* of w, if w is a segment partitioned from w_p, i.e.,

(1) if $j\%2 = 1$, w is a prefix of w_p or
(2) if $j\%2 = 0$, w is a suffix of w_p.

For example, in Fig. 2, segments "ther" and "thel" in node $n_l^{1,2}$ are parent segment of "th" in node $n_l^{2,3}$.

We can prove that the inverted list of w in node $n_l^{i,j}$ can be gotten based on its parent segments' inverted lists, i.e.,

$$\mathcal{L}_l^{i,j}[w] = \cup_{w_p} \mathcal{L}_l^{i-1,\lceil j/2 \rceil}[w_p]. \tag{3}$$

where w_p is a parent segment of w.

Based on Equation 3, we do not need to maintain inverted lists in every level. Instead, we only maintain the inverted lists for the first level and the inverted lists for other levels can be deduced from inverted lists of the first level.

Formally, in the first level, we still build inverted lists for each segment like HS-Tree. For level $i (i > 1)$, for each segment w of node $n_l^{i,j}$, we construct an address list $\mathcal{A}_l^{i,j}[w]$, which is a list of pointers that point to inverted lists of w's parent segments in node $n_l^{i-1,\lceil j/2 \rceil}$ (if $i = 2$) or address lists of w's parent segments in node $n_l^{i-1,\lceil j/2 \rceil}$ (if $i > 2$).

Next, we introduce the compact HS-Tree index as shown in Fig. 6, which consists of three components: in-memory segment metadata, disk-resident segment map and disk-resident (inverted and address) lists. The segment meta is similar to HS-Tree, and the difference is that, for each node $n_l^{i,j}$, we do not maintain segments and inverted lists. Instead, we only record a pointer that points to disk-resident segments. The segments in node $n_l^{i,j}$, i.e., $\mathcal{S}_l^{i,j}$, are consecutively stored on the disk. Each segment is associated with a pointer that points to the disk-resident (inverted or address) lists. For each segment in the first level, we keep its inverted list; otherwise, we keep its address list.

Algorithm 7: CompactHS-TreeConstruction ($\mathcal{S}$)

Input: $\mathcal{S}$: The string set
Output: The compact 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 $\mathcal{S}_l^{1,1}$, $\mathcal{S}_l^{1,2}$, inverted lists $\mathcal{L}_l^{1,1}$, $\mathcal{L}_l^{1,2}$, Write $\mathcal{L}_l^{1,1}$, $\mathcal{L}_l^{1,2}$ to disk;
6 **for** $i = 2$ *to* L **do**
7 **for** $j = 1$ *to* 2^{i-1} **do**
8 **for** *each segment* $\mathcal{S}_l^{i-1,j}$ **do**
9 Generate sets $\mathcal{S}_l^{i,2j-1}$ and $\mathcal{S}_l^{i,2j}$, address lists $\mathcal{A}_l^{i,2j-1}$, $\mathcal{A}_l^{i,2j}$;
10 Put the disk address of $\mathcal{S}_l^{i,j}$ into address list $\mathcal{A}_l^{i,2j-1}$ and $\mathcal{A}_l^{i,2j}$;
11 Write $\mathcal{A}_l^{i,2j-1}$, $\mathcal{A}_l^{i,2j}$ to disk;
12 **end**

Example 9 An example compact HS-Tree index for the strings in Table 1 is shown in Fig. 7. The string length is 7, so there are two levels. The in-memory segment meta contains the pointer to the segment map of different segments in each level. By visiting the segment map, we can get (inverted or address) lists. For example, in Fig. 7 the segment "bro" in $\mathcal{S}_l^{1,1}$ has a pointer pointing to the invert list $\mathcal{L}_7^{1,1}$[bro]. For disk-resident lists, we generate the inverted lists by partitioning strings into two segments in the level 1 (same as HS-Tree). In level 2, we store the address lists. For example, address list $\mathcal{A}_7^{2,1}$[b] contains the addresses of inverted lists $\mathcal{L}_7^{1,1}$[bro] and $\mathcal{L}_7^{1,1}$[bre]. At the same time, in HS-Tree the inverted list $\mathcal{L}_7^{2,1}$[b] stores 5 entries "1,2,3,4,5". But in our compact index, $\mathcal{A}_7^{2,1}$[b] only needs to store two addresses. Thus, the space overhead is much lower.

Algorithm 7 shows the algorithm to build the compact index. The whole process is similar to Algorithm 1. The difference is that it only builds inverted lists in the first level. In other levels, it iteratively builds address lists (line 9). As we can efficiently find the parent segments of a segments based on the partition strategy, we can efficiently construct the compact HS-Tree index.

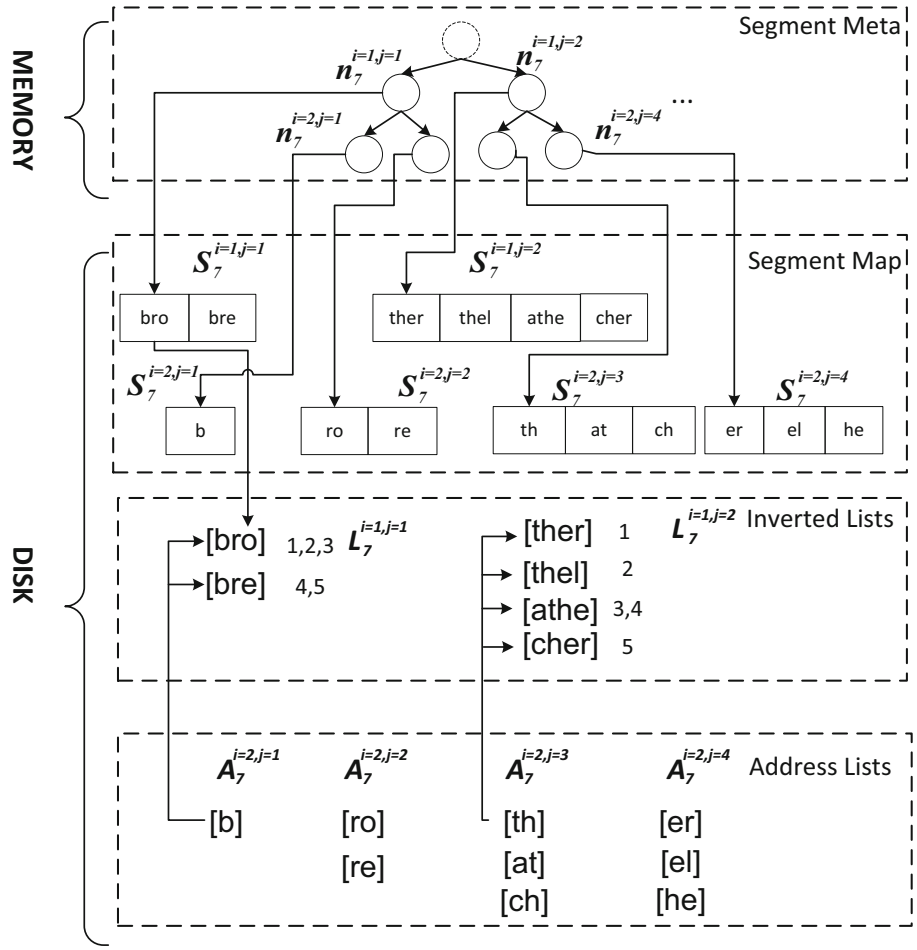

Fig. 7 An example of compact HS-Tree index

Space Complexity of Compact HS-Tree. We analyze the space complexity of the compact HS-Tree. The in-memory metadata consist of disk addresses in different levels of different groups. Suppose $l_{\min}$ is the minimum string length, and $l_{\max}$ is the maximum string length. Then, for group $\mathcal{S}_l$, the maximum number of levels is $\lfloor \log l \rfloor$. As level i has 2^i segments, there are totally $\sum_{i=1}^{\lfloor \log l \rfloor} 2^i = \mathcal{O}(l)$ segments in group $\mathcal{S}_l$. Therefore, the space complexity of the in-memory component is $\mathcal{O}(\sum_{l=l_{\min}}^{l=l_{\max}} l)$. Note that the in-memory index only depends on the string length but not the number of strings, and thus the in-memory index is rather small and can be maintained in the memory. For example, the in-memory index of a data set with 6 GB is only 8KB (see Sect. 7).

The disk-resident components consist of the segments and lists. The space complexities of segments and inverted lists are the same as those in Sect. 3, which is $\mathcal{O}(\sum_{l=l_{\min}}^{l=l_{\max}} l * |\mathcal{S}_l|)$.As strings in the first level are partitioned into 2 segments, each string is contained in 2 inverted lists; and in group $\mathcal{S}_l$ there are at most $2 * |\mathcal{S}_l|$ inverted lists. So the size of inverted list in group $\mathcal{S}_l$ is at most $4 * |\mathcal{S}_l|$. And the total space complexity of inverted lists is $\mathcal{O}(\sum_{l=l_{\min}}^{l=l_{\max}} |\mathcal{S}_l|)$. As $\mathcal{A}_l^{i,j}$ $(i \geq 2)$ consists of addresses of lists in level $i-1$, and the number of lists in level $i-1$ is $\mathcal{O}(|\mathcal{S}_l|)$, the space complexity of address list is also $\mathcal{O}(\sum_{l=l_{\min}}^{l=l_{\max}} l * |\mathcal{S}_l|)$. However, the length of $\mathcal{A}_l^{i,j}$ is determined by the number of lists in $\mathcal{A}_l^{i-1,\lceil j/2 \rceil}$ rather than the number of strings in $\mathcal{L}_l^{i-1,\lceil j/2 \rceil}$, so the constant parameter in space complexity of address lists is smaller than that of the inverted lists in HS-Tree.

Update. For inserting a string, we need to insert it into both the in-memory metadata and the disk-based segment map and inverted/address list. Firstly, we consider the in-memory metadata. We generate its segments. If there is already a segment, we ignore the segment; otherwise, we insert it into the in-memory metadata. Secondly, we consider the disk-based index. For the segment map, if the segment is already in the map, we do not need to update the segment map and only need to append it to the corresponding inverted/address list; otherwise, we need to append it into the segment map, create a new inverted list/address list and append it to the new list. Note that appending an element to segment map, inverted lists and address lists are the same as updating inverted lists, which is widely studied in [14,32]. A nature idea is to use a delta update techniques, which uses a temporary structure

to store the updated inverted lists and incrementally merges them with the original index.

For deleting a string, we need to delete it from both the in-memory metadata and the disk-based segment map and inverted/address lists. Firstly, we consider the in-memory metadata. We generate its segments. For each segment, if there exist other strings matching the segment, we ignore the segment; otherwise, we delete it from the in-memory index. For disk indexes, we only need to keep a flag that the deleted string is not valid. We can also incrementally update the original index based on the flags.

6.2 Disk-based algorithms

In this section, we discuss how to utilize the compact HS-Tree index to answer a query. To answer a threshold-based similarity query, we first identify the level $i = \lceil \log_2(\tau + 1) \rceil$ and retrieve all relevant nodes in level i from the in-memory segment meta. Then, for each relevant node, we load the corresponding segment maps into main memory and perform substring matching. For each substring, if the level $i = 1$, we directly load the inverted lists and count the number of matched segments; Otherwise, we iteratively search the address lists from level i to level 1 and then retrieve the corresponding inverted lists. Finally we perform verification to remove false positives similar to the in-memory algorithms.

However, this process is inefficient because it involves many duplicated I/Os. For each substring, it needs to load address lists in each level in a bottom-up manner. Although the index size has been reduced, it still needs a large number of I/O operations to do iterative search among different levels. Moreover, this search method also leads to repeated I/O operations for a same address list. Thus, this method had random I/O operations, which is much more expensive than sequential I/Os.

To address this issue, we observe that there are redundant I/O operations in the process of iteratively getting inverted lists for each substring. Substrings may have common addresses in the upper level. For example, in Fig. 7, if we need to check substrings "th"and "er" in level 2, we search their address lists and get the corresponding addresses $\mathcal{L}_7^{1,2}$[ther], $\mathcal{L}_7^{1,2}$[thel], $\mathcal{L}_7^{1,2}$[athe],$\mathcal{L}_7^{1,2}$ [athe] and $\mathcal{L}_7^{1,2}$ [ther]. Then, we visit each address to get the lists in the upper level. In this case, the address list with address $\mathcal{L}_7^{1,2}$ [ther] is read twice and one of them are unnecessary.

Based on this consideration, we can share the I/O operation of different substrings if they have common list addresses in the upper level. To this end, we propose a scheduling function for searching each level. The core idea is that when getting address lists for substrings, we do not immediately perform disk I/O for each address. Instead, we get the mapping relation of each substring and its corresponding addresses. We record the addresses of each distinct list and store such mapping relation in a hash map in main memory. Then we perform disk seek in batch and load all lists. According to the mapping relation, we can get the combined lists of each substring in the upper level. In this way, we can perform one sequential disk seek for each level and thus reduce the number of random I/O operations as well as total I/O operations.

Algorithm 8: Scheduling ($\mathcal{W}(q, l, lv)$)

Input: $\mathcal{W}(q, l, lv)$: the set of substrings for group l and level lv
Output: $\mathcal{H}$: the set of addresses to be visited for level lv

```
1  begin
2      Initialize an empty hash map H;
3      for pn = 1 to 2^lv do
4          Get substrings W(q, L_l^{lv,pn}) from W(q, l, lv);
5          for each w ∈ W(q, L_l^{lv,pn}) do
6              Perform substring matching in segment map, get
               A_l^{lv,pn}[w] from disk;
7              for each address α ∈ A_l^{lv,pn}[w] do
8                  if α ∉ H then H = H ∪ {α};
9      return H;
10 end
```

Algorithm 8 shows the process of the scheduling function. For substrings in each level, we first initialize the list set and the hash map (line 2). Then, for substrings of each part, we get the set of lists in upper level by looking up the segment map (line 6). Then, we put the mapping relation of the substring and its list set into a hash map (line 7). The information in the hash map will be used in the threshold-based similarity search algorithm (Algorithm 9). For each list address in the set, we check whether it has been included. If not, we add it to the list address set of the current level (line 8); otherwise, we will skip it.

Example 10 An example of the scheduling procedure is shown in Fig. 8. Suppose we need to check the substrings {"b", "re","th","sh", "el" }. We first perform substring matching with the help of segment map. And we find that substring "sh" does not have corresponding address list. Then, we get the addresses of "b","re","th" and "el", which are $\mathcal{L}_7^{1,1}$[bro], $\mathcal{L}_7^{1,1}$[bre], $\mathcal{L}_7^{1,1}$[bre], $\mathcal{L}_7^{1,2}$[ther], $\mathcal{L}_7^{1,2}$[thel], $\mathcal{L}_7^{1,2}$[thel]. Then, we seek the lists in upper level by visiting each address. At the same time, we generate the hash map of the mapping relation between substrings and set of addresses. In this case, we need 7 disk seeks. By doing scheduling, we can avoid redundant disk seek to addresses $\mathcal{L}_7^{1,1}$[bre] and $\mathcal{L}_7^{1,2}$[thel]. And only 4 disk seeks are needed.

Next, we get the corresponding inverted lists for each substring by looking up the hash map. For example, for substring

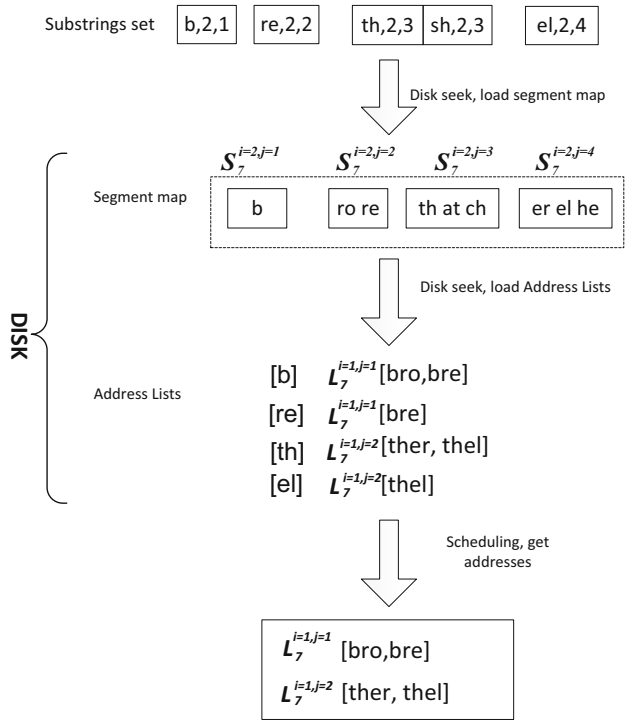

Fig. 8 The scheduling function

"b" we visit the hash map and find that the corresponding lists are from address $\mathcal{L}_7^{1,1}$[bro] and $\mathcal{L}_7^{1,1}$[bre]. Then, we merge the two lists in Fig. 7 and get the corresponding list {1,2,3,4,5}.

Based on the scheduling method, we propose the disk-based algorithm for threshold-based similarity search. The pseudo-code is shown in Algorithm 9. First, we need to calculate the maximum level and perform length filtering and generate substrings (which are similar to Algorithm 2). Then, we visit the in-memory segment meta to find disk addresses of relevant nodes. Next we visit the segment map and load these nodes' segments into memory and perform substring matching (line 2). Then, we perform iterative search in address lists from level i to level 1 to get the corresponding inverted lists for each substring (line 4–7). Next, we use the scheduling method in Algorithm 8 to get the corresponding address lists in the upper level from disk. We then get the corresponding substrings in the upper level by searching the hash map generated in the process of scheduling and doing list merging of loaded lists (line 7). We only need to keep the mapping relation for one level in memory once, so there will not be heavy space overhead of the hash map. After we reach level 1, we load and merge the corresponding inverted lists (line 10). The next procedures are the same as those in Algorithm 2: We remove invalid matching, count matched segments and perform verification for each candidate.

Algorithm 9: Disk-Based 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
       // Replace line 4 to line 8 in
          Algorithm 2 with the following:
2      Load the segment map of related nodes;
3      Generate substrings W(q, l, i) for level i;
4      for j = i to 1 do
5          H = Scheduling(W(q, l, j));
6          Perform disk seek for each address in H, get address lists;
7          Look up hash map and get set of substrings
           W(q, l, j − 1);
8      Perform disk seek for each address in H, load all the inverted
       lists;
9      for each w ∈ W(q, L_l^{i,j}) do
10         Generate L_l^{i,j}[w] by combining corresponding inverted
           lists;
11 end
```

Similarly, we can extend this algorithm to support top-k similarity search. One thing we need to notice is that the GreedyMatch strategy in Sect. 5.2 is inefficient for disk-based algorithm. The reason is that it needs extra random I/O operations to perform greedy matching in lower levels. So for disk-based top-k algorithm, we directly do verification after batch pruning.

7 Experimental study

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

7.1 Experiment setup

In-Memory Data Sets. We used publicly available real data sets in our experiments: DBLP Author, DBLP publication records,[1] QueryLog[2] and READ,[3] which were widely used in previous studies [18]. DBLP Author contained short strings, QueryLog contained medium-length strings, and DBLP contained long strings. READ is a DNA data set used in the string similarity search/join competition, organized by EDBT [33]. The details of data sets are shown in Table 2.
Disk Data Sets. To evaluate disk-based algorithms, we used DBLP and three larger data sets: PubMed Author,[4] a large QueryLog (denoted as QueryLog-L) and PubMed.

[1] http://www.informatik.uni-trier.de/~ley/db/.

[2] http://www.informatik.uni-trier.de/~ley/db/.

[3] http://www2.informatik.hu-berlin.de/~wandelt/searchjoincompetition2013/.

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

Table 2 Data sets

Data sets	#	Size (MB)	AvgLen	MaxLen	MinLen
DBLP author	613 K	9.21	15	46	6
QueryLog-S	464 K	21.3	45	522	30
DBLP	1.4 M	138.8	105	1626	1
READ	1.24 M	121	100	103	14
PubMed author	10.3 M	121.9	12	101	4
QueryLog-L	1.2 M	62.8	40	501	5
PubMed	230 M	6624	30	277	1

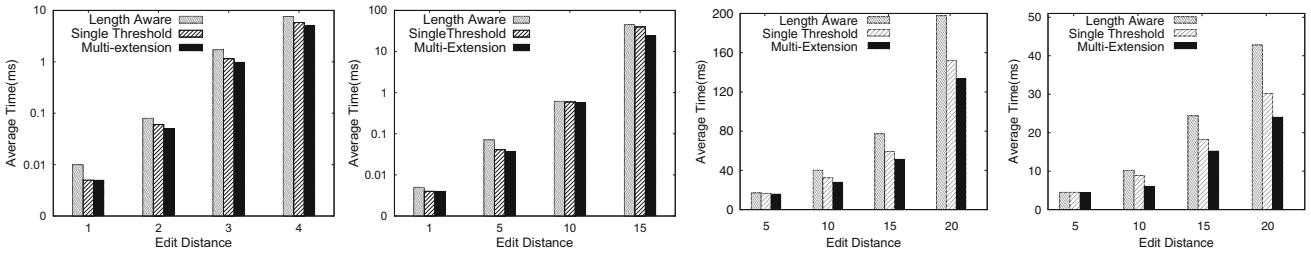

Fig. 9 Threshold-based similarity search: evaluation on different verification algorithms

We compared our in-memory algorithms with state-of-the-art methods. For threshold-based similarity search, we compared our HS-Search algorithm with Adapt [37], QChunk [28], PassJoin [23]B^{ed}-tree,[45]. Although there are other in-memory similarity search algorithms, e.g., Hobbes [1], Flamingo [20] and VChunk [21], previous studies have compared them and showed that PassJoin, Adapt and QChunk outperformed them [23,28,31,33,37]. So we only compared HS-Search with the three algorithms. For top-k similarity search, we compared HS-Topk with Appgram [39], Range [8], B^{ed}-tree and AQ [43]. We obtained the source codes of Appgram, Adapt, Range, B^{ed}-tree from the authors and implemented QChunk and AQ by ourselves [18].

We compared our disk-based algorithms with state-of-the-art algorithms. For threshold-based similarity search, we compared our method HS-Search-d with Flamingo [3], B^{ed}-tree [45], and the best algorithm of EDBT competition [33], PassJoin. For top-k similarity search, we compared our method HS-Topk-d with B^{ed}-tree [45] and Appgram [39].

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 machine with two Intel Xeon E5420 CPUs (8 cores, 2.5 GHz) and 32 GB memory. And we did not use the properties of multiple cores in our experiments.

7.2 Evaluation on threshold-based search

7.2.1 Evaluating different verification algorithms

We first evaluated the verification methods. We implemented three methods Length-aware, SingleThreshold and Multi-Extension. Length-aware was the algorithm which utilizes the string length for pruning [23]; SingleThreshold, the method has only one threshold τ; MultiExtension is our method that has separate thresholds for each matched part based on Lemma 2. All the three methods were implemented with early-termination techniques. Figure 9 shows the results by varying edit-distance thresholds on the three data sets. We can observe that SingleThreshold involved 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 had a different threshold and terminated as soon as the edit distance of one part was larger than the given threshold of that part. Thus, MultiExtension terminated earlier than SingleThreshold. For example, on the QueryLog-S data set, for $\tau = 15$, Length-aware took 55 ms on average, and SingleThreshold decreased the time to 39 ms,while MultiExtension further reduced the time to 24 ms.

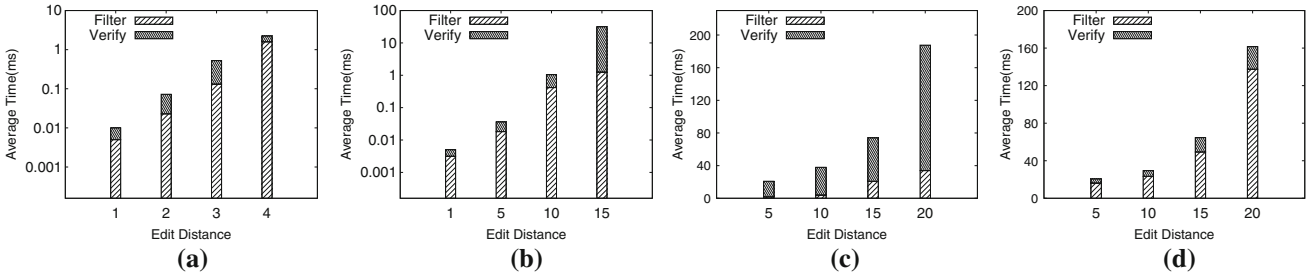

Fig. 10 Threshold-based similarity search: filter cost versus verification cost

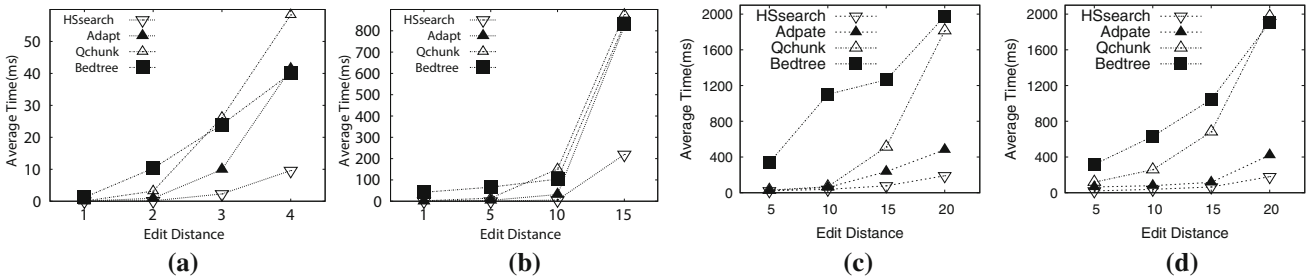

Fig. 11 Threshold-based similarity search: comparison with state-of-the-art methods

7.2.2 Filter time versus verification time

Next we evaluated the cost of the filter step and the verification step, and the result is shown in Fig. 10. We can see that our segment filter had great filtering power for small thresholds, and a large number of dissimilar strings can be pruned in the filter step, so the verification time was reduced. For large thresholds, e.g., $\tau = 15$ and $\tau = 20$ in Fig. 10c, the verification time was dominant in the overall time. This is because when the threshold become large, we needed to do segment matching in lower levels and the segments would be much shorter. It is obvious shorter segments had more chance to be matched, so the number of candidates was larger than that of small thresholds.

7.2.3 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 four data sets DBLP Author, QueryLog-S, DBLP and READ. Figure 11 shows the results. We can see that HS-Search achieved the best performance on all the data sets and outperformed existing algorithms by 3 to 20 times. For example, on the QueryLog-S data set for $\tau = 10$, HS-Search took 6 ms. And the average search time for Adapt, QChunk and B^{ed}-tree was 32, 151 and 104 ms, 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.

Our method achieved the best performance for the following reasons. Firstly, existing algorithms were based on n-grams, and our method used segments which had much stronger filtering power than gram-based methods (as segments were longer than n-grams). Moreover, the segments were selected across the string and not restricted to the prefix. Thus, our method always generated the least number of candidates. Secondly, comparing with other similarity search algorithms, we also designed efficient verification mechanism. In this way, we can take advantage of the results of filter step and avoid redundant computations. Thirdly, since previous algorithms were based on n-grams, they needed to tune the parameter n for different data sets even for different thresholds on the same data set to achieve the best performance. Our HS-Search algorithm does not need to tune any parameters. So the utility of HS-Search was much better than the existing algorithms.

In addition, we compared the index construction time and index size, and the result is shown in Table 3. We had the following observations. Firstly, 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. And the data with the same length can be constructed into HS-Search index together, so it was more efficient than inserted into index one by one, like B^{ed}-tree. Secondly, the index size of HS-Tree was nearly the same with state-of-the-art n-gram-based methods, because HS-Tree partitioned each string

Table 3 Threshold-based similarity search: index

Data set	Method	Index size (MB)	Index time (s)
DBLP author	HS-Search	43.5	1.38
	Adapt	56	8.59
	QChunk	62	1.36
	B^{ed}-tree	32	5.0
QueryLog-S	HS-Search	157	5.3
	Adapt	983	22.8
	QChunk	172	10.7
	B^{ed}-tree	129	15.0
DBLP	HS-Search	904	46.5
	Adapt	4194	121.9
	QChunk	425	70.5
	B^{ed}-tree	347	96.0
READ	HS-Search	559	24.1
	Adapt	4137.8	119.2
	QChunk	423.5	61.0
	B^{ed}-tree	318.0	75.0

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 generated $ln + 1$ grams. Thus, they generated similar number of n-grams/segments, and thus, the index sizes were also similar. In addition, in the inverted lists, for a segment, HS-Tree only needed to maintain the string containing it, but n-gram-based methods also needed to store the position of n-gram, so our index size can be smaller than state-of-the-art methods. Thirdly, B^{ed}-tree organized "similar" strings into a B-tree node based on specific orders and did not need to maintain inverted lists, and thus, its index size was smaller.

7.2.4 Scalability

We evaluated the scalability of HS-Search. We varied the number of strings in each data set and tested the average search time. Figure 12 shows the result on the three data sets. We can see that as the size of a data set 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 200,000 strings, 400,000 strings and 600,000 strings were, respectively, 20, 27 and 33 ms.

7.3 Evaluation on top-k similarity search

7.3.1 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 extended the threshold-based algorithm by increasing the threshold by 1 each time and executing the algorithm multiple times; Batch only implemented the batch-pruning-based method; Greedy only implemented the greedy-match strategy; and B+G used 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 Fig. 13. It was clear that Batch can prune dissimilar strings in batch and thus reduce

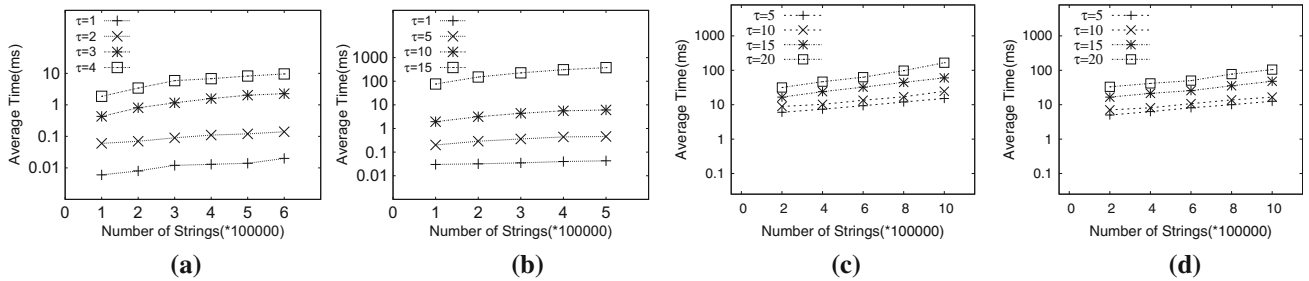

Fig. 12 Threshold-based similarity search: scalability

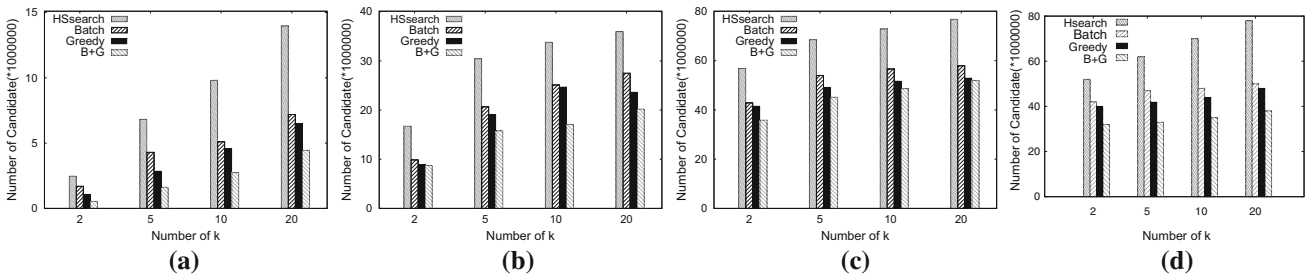

Fig. 13 Top-k similarity search: candidate number of different filters

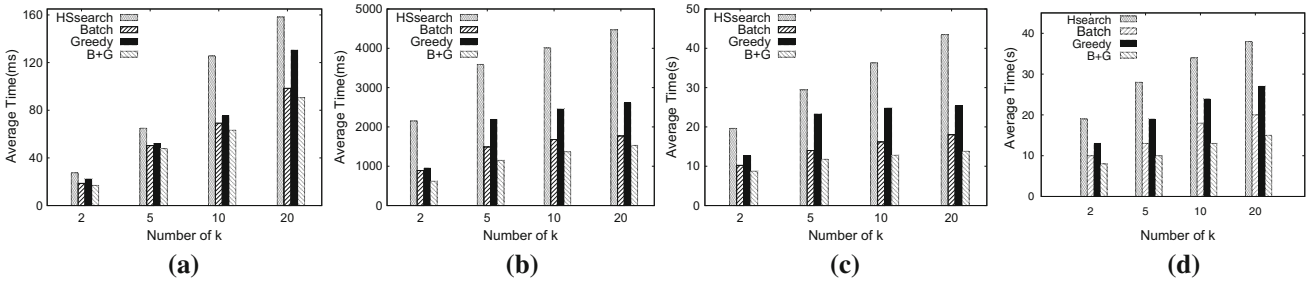

Fig. 14 Top-k similarity search: average time of different filters

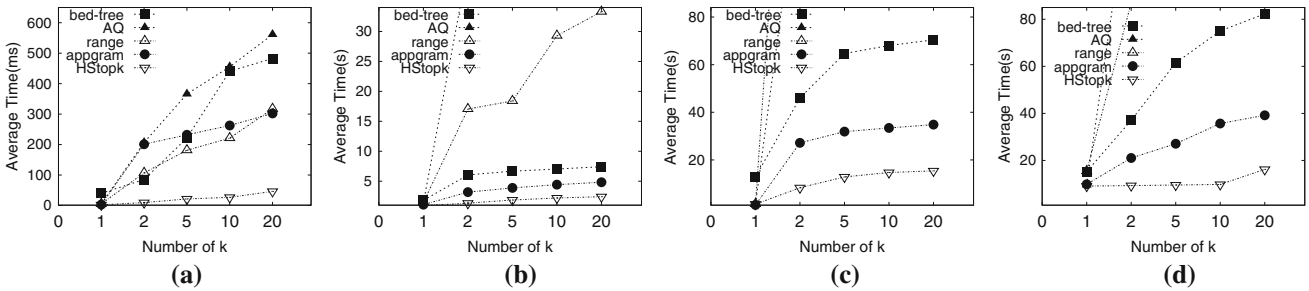

Fig. 15 Top-k similarity search: comparison with state-of-the-art methods

the number of candidates. As Greedy can find consecutive errors within a long segment, the number of candidates can also be reduced. Our method had significant filtering power by combining these two filters together. For example, on Author data set 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 showed the effectiveness of pruning techniques.

Then we evaluated the average search time. As shown in Fig. 14, the average search time of Batch was much larger than that of HS-Search because Batch can dynamically update the 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.

7.3.2 Comparison with state-of-the-art methods

We compared our HS-Topk algorithm with state-of-the-art methods AQ, B^{ed}-tree, Range and Appgram. We evaluated the performance on the same three data sets. For each experiment, we randomly selected 100 queries from the data set and reported the average search time. All the algorithms were in-memory, including B^{ed}-tree. The results are shown in Fig. 15.

We had the following observations. Firstly, our HS-Topk algorithm outperformed all the existing methods. Secondly, Appgram had the second best performance. This was because Appgram allowed approximate matching between n-grams, which can relax the filter condition. It also devised a double-level index structure and used the CA algorithm to accelerate top-k search. However, as Appgram used the mapping distance of approximate n-gram to estimate the lower bound of top-k results, the lower bound was loose. Moreover, the searching process on the double-level index also involved extra filter cost. Therefore, the overall performance of Appgram was worse than our HS-Topk algorithm. Thirdly, on the Author data set, Range outperformed B^{ed}-tree and AQ. This was because Range took advantage of the trie-based index [8]. 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. For instance, for $k = 10$, Range took 340 ms on average, but our HS-Topk only took 41 ms. Fourthly, on the DBLP data set with long strings, our method significantly outperformed other methods. We can see from Fig. 15c 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 was because in data sets with long strings, the length of common prefixes is relatively short and there would be a large number of long, single branches in the

Table 4 top-k search: index

Data set	Method	Index size (MB)	Index time (s)
DBLP author	HS-Tree	43.5	1.38
	Appgram	37	3.99
	Range	146	12.5
	AQ	316	6.4
	B^{ed}-tree	32	5.0
QueryLog-S	HS-Tree	157	5.3
	Appgram	104	11.3
	Range	1600	17
	AQ	224	78.3
	B^{ed}-tree	129	15.0
DBLP	HS-Tree	904	46.5
	Appgram	365	43.1
	Range	4480	102.3
	AQ	1824	167.2
	B^{ed}-tree	347	96.0
READ	HS-Tree	559	24.1
	Appgram	283.7	33.6
	Range	3808.9	90.6
	AQ	1202.9	108.9
	B^{ed}-tree	318.0	75.0

trie index, which brings both space and computational overhead. Finally,B^{ed}-tree had relatively well performance on each data set 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, B^{ed}-tree performed the worst because it involved many dissimilar strings in one node.

Table 4 shows the index size and index time of each algorithm. We can see that HS-Tree involved both less space and time overhead than Range and AQ. Compared with Appgram, our method involved less index construction time because Appgram needed to divide strings into n-grams with different sizes. Moreover, our method also outperformed them in query efficiency, due to the effective filtering power on our segments.

7.3.3 Scalability

We evaluated the scalability of HS-Topk. We varied the size of each data sets and tested the average query time for our HS-Topk algorithm. As shown in Fig. 16, our method scaled very well with different k values and can support large-scale data. This was attributed to our segment-based filtering techniques and effective indexes. For example, on the QueryLog-S data set 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.

7.4 Evaluation on disk-based algorithms

7.4.1 Evaluating the compact index structure and iterative search strategy

We first evaluated the index size and time of our compact HS-Tree index. We implemented two methods: Simple and Compact. Simple was the naive method which simply stored the HS-Tree on disk. Compact was the method using the compact index structure in Sect. 6.1. The index size and time are shown in Table 5 . We can see that Compact had much smaller index size than Simple because Simple contained redundant information in inverted lists. By using address lists instead of inverted lists, Compact can significantly reduce the index size. This was because Compact can match the address lists in the upper level during the process of iteratively dividing segments. At the same time, Compact only involved slightly larger index construction time than Simple, because Compact required to generate the address lists from the inverted lists.

Next we evaluated the efficiency of our disk-based search algorithms. We implemented three methods: Simple, Compact and Iter-Schedule. Simple was the naive method on HS-Tree, Compact used the simple iteratively search on the compact index, and Iter-Schedule was the iterative search algorithm with scheduling in Sect. 6.2. We compared both the threshold-based and top-k similarity search algorithms. We used two metrics: the number of I/Os and the overall query time.

The results of the number of I/O are shown in Figs. 17 and 18. Although Compact can reduce the index size, the number of I/O operations was still large because it involved random disk I/Os. It is obvious that Iter-Schedule had much smaller number of I/O operations because it can avoid duplicate I/Os with the help of the scheduling process. We can also see that Iter-Schedule was more significant in top-k than in threshold-based similarity search. The main reason was that top-k similarity search involved search operations on multiple levels, which involved more duplicate I/O operations in Compact method. For example, for top-k search on DBLP data set when $k = 4$, Simple involved 19,435 I/O operations, while Compact reduced the number to 6,978 and Iter-Schedule only involved only 1,635 I/O operations.

The results of overall query time are shown in Figs. 19 and 20. We can see that Iter-Schedule had the best performance because it had the least number of I/O operations. The iterative search algorithm on address lists in both Compact and Iter-Schedule involved extra CPU overhead. In some cases, Compact was even worse than Simple. But with the scheduling process, our method can efficiently trade the CPU time for the more expensive disk I/O operations in Iter-Schedule. Therefore our method can get a smaller overall query time.

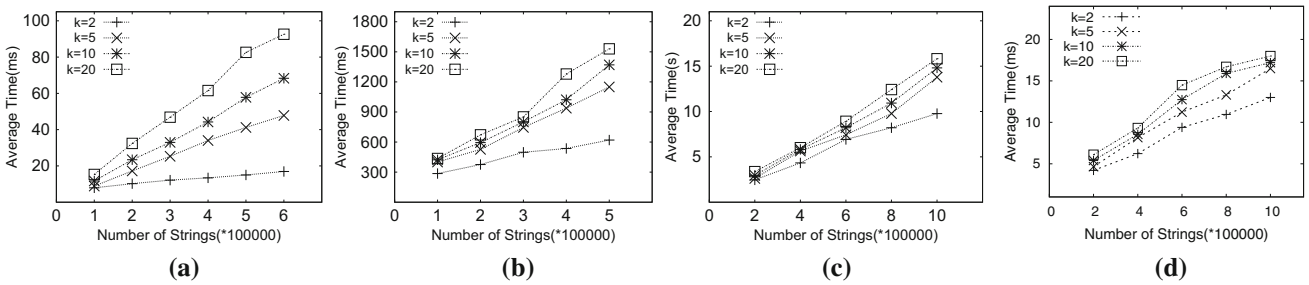

Fig. 16 Top-k similarity search: scalability

Table 5 Disk-based algorithms: index

Data set	Method	Index size (MB)	Index time (s)
PubMed author	Simple	817	60.3
	Compact	604	71.2
	Flamingo	629	57.4
	B^{ed}-tree	314	78
	PassJoin	336	69.5
	Appgram	328	33.9
QueryLog-L	Simple	170	19.68
	Compact	102	22.3
	Flamingo	158	16.8
	B^{ed}-tree	92	63
	PassJoin	62	19.1
	Appgram	103	11.2
DBLP	Simple	904	78.3
	Compact	667	84.5
	Flamingo	702	75.2
	B^{ed}-tree	347	96
	PassJoin	375	81
	Appgram	365	43.1
PubMed	Simple	51844	3102.1
	Compact	38233	3294.3
	Flamingo	41022	2933.4
	B^{ed}-tree	19288	3821
	PassJoin	22140	3110.7
	Appgram	20495	1677.2

7.4.2 Evaluating different segment size

We evaluated the effect of segments' size. We tested the average search time by varying the size of segments for our methods HS-Search-d and HS-Topk-d. We randomly combined words in the data set PubMed to generate data sets with different string lengths but with the same number of strings. Since the segment size depended on the string length, our generated data sets had different segment size. Table 6 shows the results by varying string length from 10 to 30. We can see that the in-memory segment meta of our index took very little size. This is because the segment meta depended on the string length but not the number of strings. The disk-based index was large because the disk-based index depended on the number of strings. The index size also scaled well with different number of segments. For example, the index sizes were 7.2 GB for 149 million segments and the size increased to 37 GB for 940 million segments.

We also evaluated the performance by varying the segment size, and Fig. 21 showed the results. We can see that our method scaled very well for different sizes of segments. For example, utilizing HS-Search-d algorithm with $\tau = 8$, our method took 167 seconds for the data set with average string length of 20. And it increased to 247 seconds when the average string length became to 30. Thus, even if the number of segments increased, our method still achieved rather high performance.

7.4.3 Evaluating updates

We evaluated the performance of updates on our disk-based algorithms. We first tested the update cost on the PubMed data set. We first built an index on the whole data set and then inserted or deleted 10K strings. Figure 22a shows the average update cost. We could see that the update cost was rather small and only took less than 1ms for each update. This was because our method used a temporary index to maintain the updated strings, and we only needed to merge indexes for 10K updates and thus the average update cost was small.

We then evaluated the query performance on the updated index. Figure 22b, c show the average query time of our two disk-based algorithms after updating a number of strings. For example, x-axis $= 1$ denoted the query performance after updating 10K strings and x-axis $= 0$ denoted the query performance when there was no update. In these experiments, we set $k = 8$ and $\tau = 15$. We can see that our method achieved higher performance, even on the updated indexes. As the number of updates was larger, the performance became slightly slower because we needed to use the temporary index to answer the queries and a larger number of updates led to larger temporary indexes.

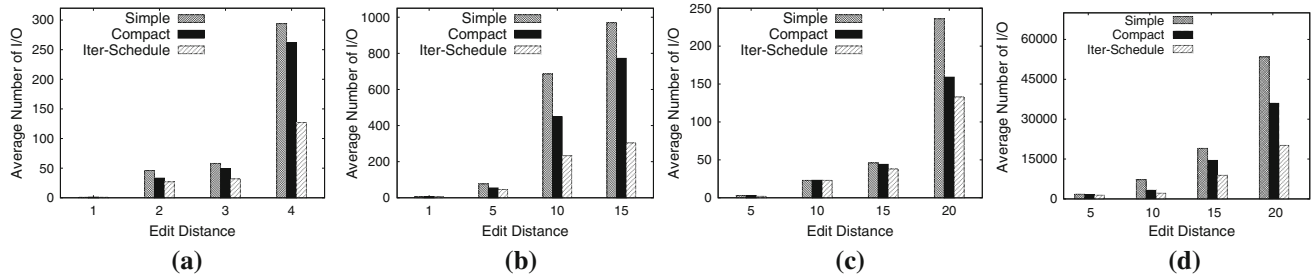

Fig. 17 Disk-based algorithm: average number of I/Os for threshold-based similarity search

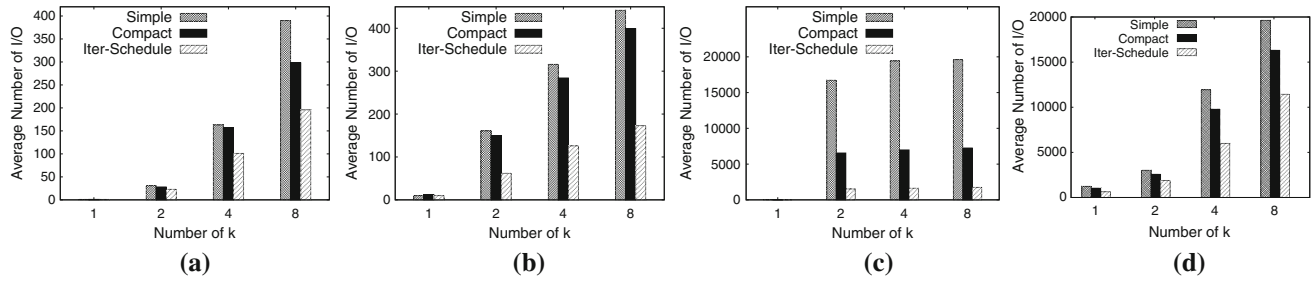

Fig. 18 Disk-based algorithm: average number of I/Os for top-k similarity search

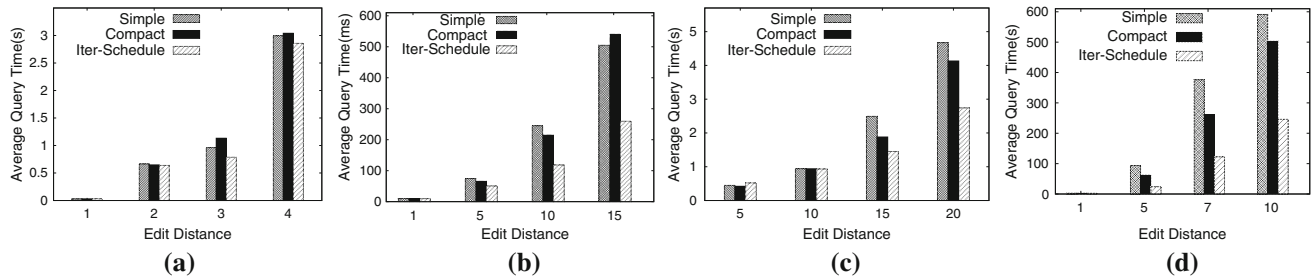

Fig. 19 Disk-based algorithm: average query time for threshold-based similarity search

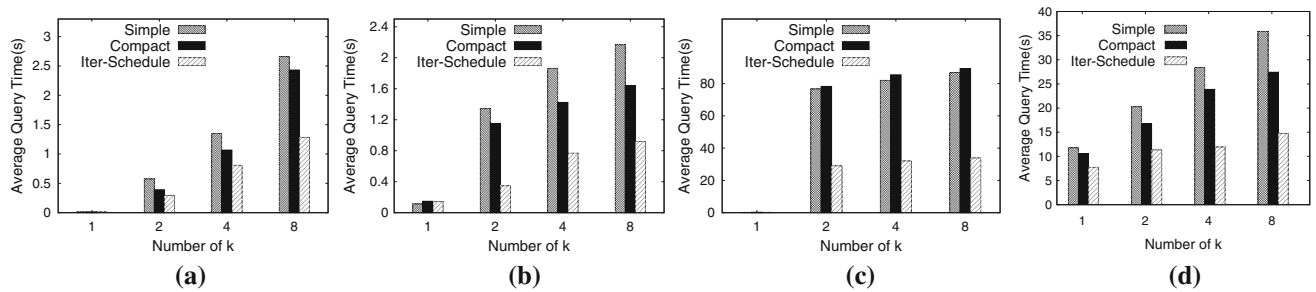

Fig. 20 Disk-based algorithm: average query time for top-k similarity search

7.4.4 Comparison with state-of-the-art methods

We compared our disk-based algorithms (the Iter-Schedule method) with state-of-the-art algorithms. For threshold-based similarity search, we compared our method HS-Search-d with Flamingo [3], B^{ed}-tree and PassJoin. For top-k similarity search, we compared our method HS-Topk-d with B^{ed}-tree and Appgram.

The results of threshold-based similarity search are shown in Fig. 23. We can see that HS-Search-d performed best on all the three data sets and outperformed existing methods by 2 to 20 times. For example, on data set PubMed Author when threshold was 3, HS-Search-d took 0.78 seconds on average, while the average query time of Flamingo, B^{ed}-tree and PassJoin was correspondingly 2.31, 9.62 and 1.66 seconds. This was because by reducing index size and scheduling the I/O operation, our method reduced the number of I/O opera-

Table 6 Disk-based algorithms: varying segment size

Avg Len	30	20	10
Disk-based index on PubMed			
Num of segments (million)	940	596	149
Segment meta (KB)	8.5	7.16	5.39
Segment map (GB)	7.8	4.45	1.28
Index size (GB)	37.3	23.8	7.2
Avg Len	100	50	10
In-memory-based index on READ			
Num of segments (million)	158	72	16
Index size (MB)	599	294	58

tions and avoided unnecessary random accesses. Flamingo used an adaptive algorithm to avoid visiting long inverted lists, but it cannot make accurate decision of list retrievals as it used heuristic-based cost model. So Flamingo still incurred heavy I/O overhead. As the range search operation in B^{ed}-tree needed to traverse multiple paths for a single threshold, it needed to visit many nodes on disk. So B^{ed}-tree had the worst I/O performance. PassJoin achieved slower performance because it required to find an appropriate threshold to answer the query and the threshold might be larger and thus had poor performance. The results for top-k similarity search are shown in Fig. 24. Our method HS-Topk-d outperformed B^{ed}-tree by an order of magnitude, and it was also faster than Appgram. For example, on QueryLog-L data set with $k = 8$, the average query time of HS-Topk-d was 0.92 seconds, while that of B^{ed}-tree and Appgram was 15.19 and 6.9 seconds. The main reason was that B^{ed}-tree involved many dissimilar strings in one node, and it needed to check a large number of nodes to get the top-k results. Thus, B^{ed}-tree involved much more disk I/O operations than our HS-Topk-d. In addition, Appgram cannot achieve high performance like ours because it required to constructed two-level index for filtering and its loose lower bound for mapping distance of n-grams made the verification step need extra time.

7.4.5 Scalability

Finally, we evaluated the scalability of our disk-based algorithms. We varied the size of each data set and tested the average query time for our HS-Search-d and HS-Topk-d algorithm. The results of HS-Search-d are shown in Fig. 25. As the size of a data set increased, our method scaled very well for different edit-distance thresholds and achieved near linear scalability. For example, on the PubMed Author data set for $\tau = 4$, our method took 115 ms for 2M strings, and the time increased to 268 ms for 4M strings and 354 ms for 6M strings. The results of HS-Topk-d are shown in Fig. 26. We can see that our method also achieved linear scalability for different k values. For example, on the QueryLog-L data

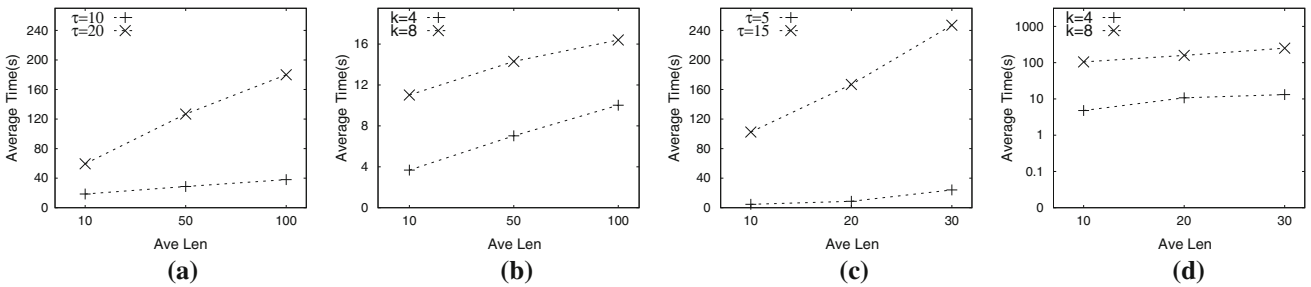

Fig. 21 Disk-based algorithm: evaluation on different segment size

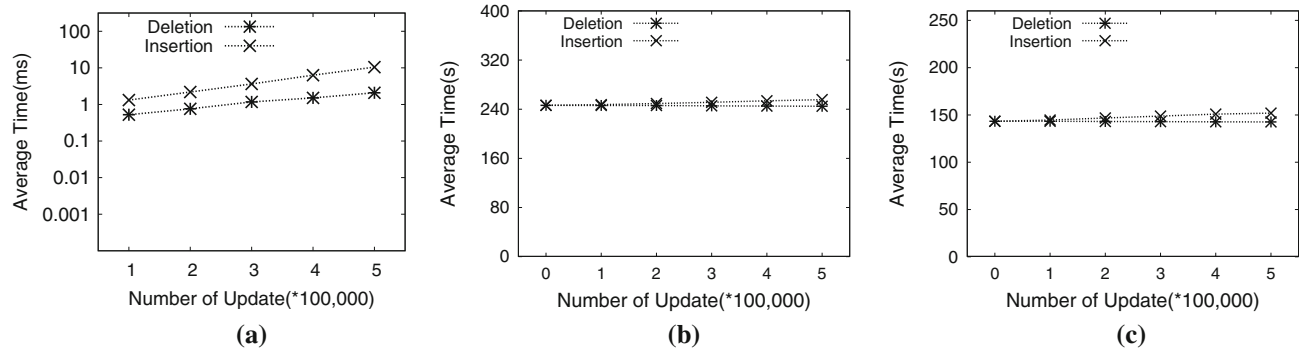

Fig. 22 Disk-based algorithm: evaluation on update on PubMed. **a** Update cost, **b** search efficiency (threshold $\tau = 15$) , **c** search efficiency (Top $-k$, $k = 8$)

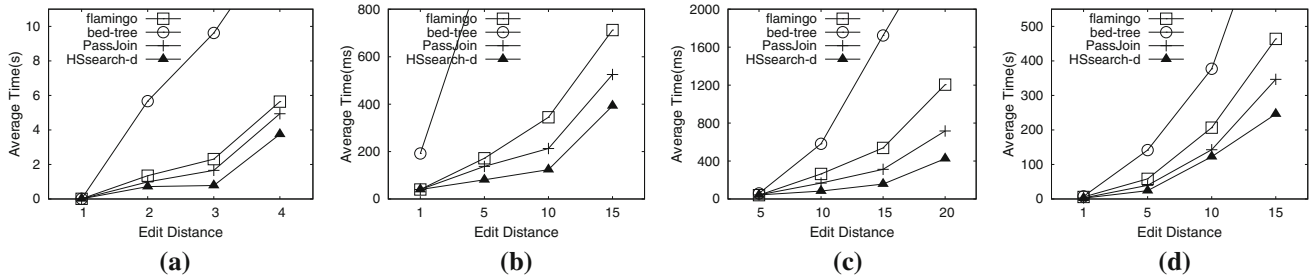

Fig. 23 Disk-based algorithm: comparison with state-of-the-art threshold-based similarity search methods

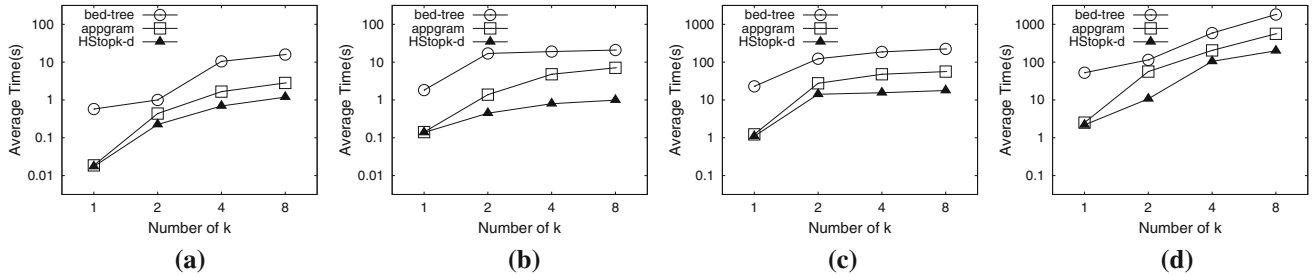

Fig. 24 Disk-based algorithm: comparison with state-of-the-art top-k similarity search methods

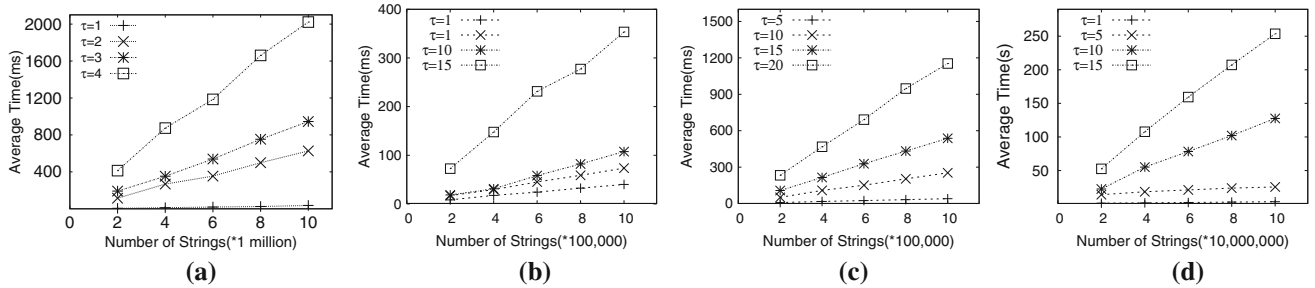

Fig. 25 Disk-based algorithm: scalability of threshold-based similarity search methods

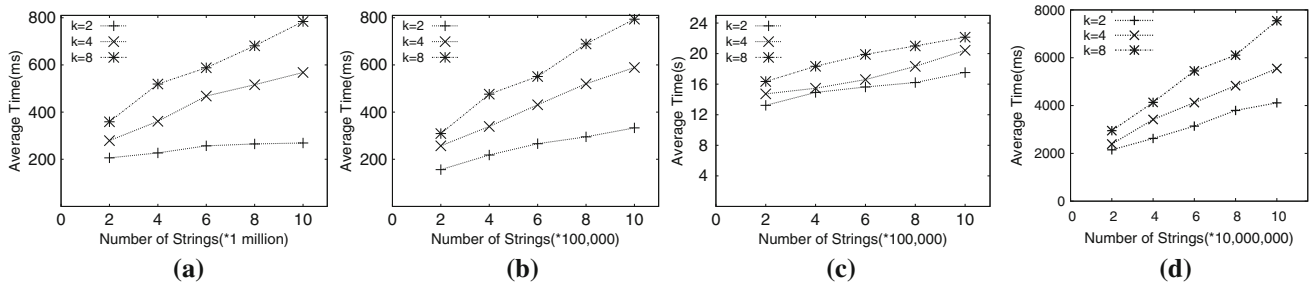

Fig. 26 Disk-based algorithm: scalability of top-k similarity search methods

set, when $k = 8$, the average search time for 40K strings, 60K strings and 80K strings was, respectively, 476, 551, and 689 ms. This is attributed to our compact index and efficient scheduling algorithm.

We also evaluated the index sizes by increasing the data set size. Table 7 shows the results. We can see that the index size scaled very well with the increase in data set size. For

Table 7 Disk-based algorithms: index size of different string numbers on PubMed

Num (*1 million)	50	100	150	200
Index size (GB)	8.3	16.8	24.6	31.1
Index time (s)	670.5	1421.3	1947.153	2533.6

Table 8 Disk-based algorithms versus in-memory algorithms

τ	HS-Search (ms)	HS-Search-d (ms)
5	21.3	42.8
10	38.0	83.6
15	77.5	157.4
20	190.0	424.9
k	HS-Topk (ms)	HS-Topk-d (ms)
1	1.5	2.74
2	8.75	17.58
4	12.44	21.43
8	14.72	30.45

example, the index was 19.8G and the construction time was 1421.3 seconds when the data set contained 100 million strings. And when the data set increased to 200 million, the size of index became to 31.1G and the construction time was 2533.6 seconds.

7.5 In-memory algorithms versus disk-based algorithms

We compared the disk-based algorithms HS-Search-d, HS-Topk-d and the in-memory algorithms HS-Search, HS-Topk. Table 8 shows the results on the DBLP data set. We can see that our disk-based methods were a bit slower than the in-memory algorithms. When $k = 8$, the average search time of HS-Topk and HS-Topk-d was 14.7 seconds and 30.4 seconds, respectively. When $\tau = 10$, the average search time of HS-Search and HS-Search-d were 38 seconds and 83 seconds, respectively. The gap between the in-memory algorithms and disk-based algorithms was not large, due to our compact disk-based index and efficient search algorithms. Thus, our disk-based algorithms can efficiently support the large data sets that cannot be fit in the memory.

8 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. We also devised disk-based indexes and algorithms to support large data sets that cannot be loaded into memory. Experimental results show that our method significantly outperforms state-of-the-art algorithms on both threshold-based and top-k similarity search problems for both in-memory and disk-based settings.

Acknowledgements This work was supported by 973 Program of China (2015CB358700), NSF of China (61373024, 61632016, 61422205, 61472198, 61661166012), Shenzhou, Tencent, TNList, FDCT/116/2013/A3, and MYRG105 (Y1-L3)-FST13-GZ.

References

1. Ahmadi, A., Behm, A., Honnalli, N., Li, C., Weng, L., Xie, X.: Hobbes: optimized gram-based methods for efficient read alignment. Nucleic Acids Res. **40**, e41 (2012)
2. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In: WWW, pp. 131–140 (2007)
3. Behm, A., Li, C., Carey, M.J.: Answering approximate string queries on large data sets using external memory. In: ICDE, pp. 888–899 (2011)
4. Chaudhuri, S., Ganti, V., Kaushik, R.: A primitive operator for similarity joins in data cleaning. In: ICDE (2006)
5. Chaudhuri, S., Kaushik, R.: Extending autocompletion to tolerate errors. In: SIGMOD Conference, pp. 707–718 (2009)
6. Deng, D., Li, G., Feng, J.: A pivotal prefix based filtering algorithm for string similarity search. In: SIGMOD Conference, pp. 673–684 (2014)
7. Deng, D., Li, G., Feng, J., Duan, Y., Gong, Z.: A unified framework for approximate dictionary-based entity extraction. VLDB J. **24**(1), 143–167 (2015)
8. Deng, D., Li, G., Feng, J., Li, W.-S.: Top-k string similarity search with edit-distance constraints. In: ICDE, pp. 925–936 (2013)
9. Deng, D., Li, G., Hao, S., Wang, J., Feng, J.: Massjoin: a mapreduce-based method for scalable string similarity joins. In: ICDE, pp. 340–351 (2014)
10. Deng, D., Li, G., Wen, H., Feng, J.: An efficient partition based method for exact set similarity joins. PVLDB **9**(4), 360–371 (2015)
11. Deng, D., Li, G., Wen, H., Jagadish, H.V., Feng, J.: META: an efficient matching-based method for error-tolerant autocompletion. PVLDB **9**(10), 828–839 (2016)
12. Feng, J., Wang, J., Li, G.: Trie-join: a trie-based method for efficient string similarity joins. VLDB J. **21**(4), 437–461 (2012)
13. Gerdjikov, S., Mihov, S., Mitankin, P., Schulz, K.U.: Wallbreaker: overcoming the wall effect in similarity search. In:EDBT/ICDT, pp. 366–369 (2013)
14. Guo, L., Shanmugasundaram, J., Beyer, K.S., Shekita, E.J.: Efficient inverted lists and query algorithms for structured value ranking in update-intensive relational databases. In: ICDE, pp. 298–309 (2005)
15. Gusfield, D.: Algorithms on Strings, Trees, and Sequences—Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
16. Hadjieleftheriou, M., Yu, X., Koudas, N., Srivastava, D.: Hashed samples: selectivity estimators for set similarity selection queries. PVLDB **1**(1), 201–212 (2008)
17. Ji, S., Li, G., Li, C., Feng, J.: Efficient interactive fuzzy keyword search. In: WWW (2009)
18. Jiang, Y., Li, G., Feng, J.: String similarity joins: an experimental evaluation. PVLDB **7**(8), 625–636 (2014)
19. Kim, Y., Shim, K.: Efficient top-k algorithms for approximate substring matching. In: SIGMOD Conference, pp. 385–396 (2013)
20. Li, C., Lu, J., Lu, Y.: Efficient merging and filtering algorithms for approximate string searches. In: ICDE, pp. 257–266 (2008)
21. Li, C., Wang, B., Yang, X.: Vgram: improving performance of approximate queries on string collections using variable-length grams. In: VLDB, pp. 303–314 (2007)

22. Li, G., Deng, D., Feng, J.: Faerie: efficient filtering algorithms for approximate dictionary-based entity extraction. In: SIGMOD Conference, pp. 529–540 (2011)
23. Li, G., Deng, D., Wang, J., Feng, J.: Pass-join: a partition-based method for similarity joins. PVLDB **5**(3), 253–264 (2011)
24. Li, G., Feng, J., Li, C.: Supporting search-as-you-type using SQL in databases. IEEE Trans. Knowl. Data Eng. **25**(2), 461–475 (2013)
25. Li, G., He, J., Deng, D., Li, J.: Efficient similarity join and search on multi-attribute data. In: SIGMOD, pp. 1137–1151 (2015)
26. Li, G., Ji, S., Li, C., Feng, J.: Efficient fuzzy full-text type-ahead search. VLDB J. **20**(4), 617–640 (2011)
27. Mansour, E., Allam, A., Skiadopoulos, S., Kalnis, P.: Era: Efficient serial and parallel suffix tree construction for very long strings. Proc. VLDB Endow. **5**(1), 49–60 (2011)
28. Qin, J., Wang, W., Lu, Y., Xiao, C., Lin, X.: Efficient exact edit similarity query processing with the asymmetric signature scheme. In: SIGMOD Conference, pp. 1033–1044 (2011)
29. Sarawagi, S., Kirpal, A.: Efficient set joins on similarity predicates. In: SIGMOD Conference, pp. 743–754 (2004)
30. Sellers, P.H.: The theory and computation of evolutionary distances: pattern recognition. J. Algorithms **1**(4), 359–373 (1980)
31. Siragusai, E., Weese, D., Reinert, K.: Fast and accurate read mapping with approximate seeds and multiple backtracking. Nucleic Acids Res. **41**(7), e78 (2013)
32. Tomasic, A., Garcia-Molina, H., Shoens, K.A.: Incremental updates of inverted lists for text document retrieval. In: SIGMOD, pp. 289–300 (1994)
33. Wandelt, S., Deng, D., Gerdjikov, S., Mishra, S., Mitankin, P., Patil, M., Siragusa, E., Tiskin, A., Wang, W., Wang, J., Leser, U.: State-of-the-art in string similarity search and join. SIGMOD Rec. **43**(1), 64–76 (2014)
34. Wang, J., Li, G., Deng, D., Zhang, Y., Feng, J.: Two birds with one stone: an efficient hierarchical framework for top-*k* and threshold-based string similarity search. In: ICDE (2015)
35. Wang, J., Li, G., Feng, J.: Trie-join: efficient trie-based string similarity joins with edit-distance constraints. PVLDB **3**(1), 1219–1230 (2010)
36. Wang, J., Li, G., Feng, J.: Fast-join: An efficient method for fuzzy token matching based string similarity join. In: ICDE, pp. 458–469 (2011)
37. Wang, J., Li, G., Feng, J.: Can we beat the prefix filtering? An adaptive framework for similarity join and search. In: SIGMOD Conference, pp. 85–96 (2012)
38. Wang, W., Xiao, C., Lin, X., Zhang, C.: Efficient approximate entity extraction with edit distance constraints. In: SIGMOD Conference, (2009)
39. Wang, X., Ding, X., Tung, A.K.H., Zhang, Z.: Efficient and effective knn sequence search with approximate n-grams. PVLDB **7**, 1–12 (2014)
40. Xiao, C., Qin, J., Wang, W., Ishikawa, Y., Tsuda, K., Sadakane, K.: Efficient error-tolerant query autocompletion. PVLDB **6**(6), 373–384 (2013)
41. Xiao, C., Wang, W., Lin, X.: Ed-join: an efficient algorithm for similarity joins with edit distance constraints. PVLDB **1**(1), 933–944 (2008)
42. Xiao, C., Wang, W., Lin, X., Yu, J.X.: Efficient similarity joins for near duplicate detection. In: WWW, pp. 131–140 (2008)
43. Yang, Z., Yu, J., Kitsuregawa, M.: Fast algorithms for top-k approximate string matching. In: AAAI (2010)
44. Yu, M., Li, G., Deng, D., Feng, J.: String similarity search and join: a survey. Front. Comput. Sci. **10**(3), 399–417 (2016)
45. Zhang, Z., Hadjieleftheriou, M., Ooi, B.C., Srivastava, D.: Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In: SIGMOD Conference, pp. 915–926 (2010)