

A Two-Tire Index Structure for Approximate String Matching with Block Moves^{*}

Bin Wang^{1,2}, Long Xie³, and Guoren Wang^{1,2}

¹ Key Laboratory of Medical Image Computing (Northeastern University),
Ministry of Education

² School of Information Science and Engineering,
Northeastern University, Shenyang, China
{binwang, wanggr}@mail.neu.edu.cn

³ Information School, Liaoning University, Shenyang, China
dragonxie1983@gmail.com

Abstract. Many applications need to solve the problem of approximate string matching with block moves. It is an NP-Complete problem to compute block edit distance between two strings. Our goal is to filter non-candidate strings as much as possible. Based on the two matured filter strategies, frequency distance and positional q -gram, we propose a two-tire index structure to make the use of the two filters more efficiently. We give a full specification of the index structure, including how to choose character order to achieve a better filterability and how to balance number of strings in different clusters. We present our experiments on real data sets to evaluate our technique and show the proposed index structure can provide a good performance.

1 Introduction

Approximate string matching has been widely used in various fields, such as finding DNA pieces, which might be changed by some possible operations in computational biology, regaining the original signals which are transmitted through noisy channels in communication, and searching texts with error tolerance in IR, etc. To measure the distance between two strings, there are many methods. And the most widely used is *edit distance* a.k.a. Levenshtein distance. Calculating edit distance between two strings s and p is to find the minimum number of insertion, delete, and substitute operations required to transform s to p . Besides the above three operations, block move (i.e. substring move) operation is very general and has been of interest in many real applications. For example, we can use two block move operations shown in Fig. 1 to change `aaccg` to `ggcaa`, which was called *block edit distance*.

It is well known that the problem of calculating block edit distance is an NP-Complete problem [1, 2]. The block moves operation is flexible, i.e. the length

^{*} Supported by the National Natural Science Foundation of China (Grant No. 60828004), the Program for New Century Excellent Talents in Universities (Grant No. NCET-06-0290), and the Fok Ying Tong Education Foundation Award 104027.

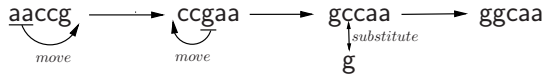


Fig. 1. Translating `aaccg` to `ggcaa` by using block moves.

of a block is not fixed and even the position where it should be moved to is unfixed. For instance, in Fig. 1, to translate `aaccg` to `ggcaa`, the blocks of `aaccg` are all its substrings, $\{\text{a, aa, aac, } \dots\}$, and for each block, it can be moved to every possible position in the string, such as `aa`, it can be moved to the rear of the first `c`, the front of `g` and the rear of `g`.

Therefore, given a collection of strings S and a query string p , it is expected to reduce the search space to answer the query p , i.e., to choose a candidate set $S_C \subseteq S$ efficiently, such that the block edit distance between $s(\in S_C)$ and p is not larger than a given threshold.

Many filter approaches have been deployed to reduce the search space of queries on string collection. Frequency distance [3], denoted by FD , is a widely known lower bound of edit distance. Intuitively, if two strings do not share close number of same characters, then they could not be similar. For instance, string $s_1 = \text{acc}$ has one `a` and two `c`, while string $s_2 = \text{abb}$ has one `a` and two `b`. Since s_1 and s_2 do not share the two `c` or `b`, then their edit distance could not be less than 1. FD is easy to calculate, but the filterability is weak, since the character position information is ignored. For instance, string `abcdef` and `badcfe` share the same number of characters, but they are not similar.

Positional q -grams is another commonly used filter approach [4, 5]. Given a string s , its positional q -grams are obtained by “sliding” a window of length q over the characters of s . If two strings are similar, they must share enough number of common grams. Positional q -grams can prune away more non-candidate strings since it considers overlapped grams, however, it requires more overhead on decomposing strings into gram sets.

Therefore, it is desired to combine the advantages of both FD and positional q -gram. In this paper, we propose a novel two-tier index structure to and generate a candidate set as small as possible.

The remaining of this paper is organized as follows: Section 2 briefly discusses related work; Section 3 provides background material; Section 4 discusses the construction of two-tier trie, $2TI$; Section 5 introduces how to use $2TI$ to filter; Section 6 reports the experimental results; finally, we summarize the paper and give the future work in Section 7.

2 Related Work

Approximate string matching is one of the basic problems in computer science, the earlier discuss of it begins at 1960s. There are deeply and detailed introductions to the basic string matching problem in [6–8]. And [9] by Navarro is an

excellent survey about approximate string matching with classical edit distance. Lopresti etc. give ten modules of approximate string matching with block moves in [10], but they mainly focus on the complex of those modules. Since substitution can be regarded as deleting a character and then inserting another character, Dana etc. introduce the edit distance with moves, which substitutes substring movement for substitution [1, 2]. They consider the text as a link list module, and the cost of adding a node, deleting a node and moving a sublist of nodes to other place is the same. And the authors of [11, 12] embed strings to vector space, then use \mathcal{L}_1 distance between the string vectors to get an approximate result to the block edit distance. And the approximation to optimal of greedy algorithms is given in [13]. The difference between our work and the above is that they mainly focus on how to calculate the block edit distance between two strings, but we are interested in the problem of string collection.

There are also some indexing methods for approximate string matching [14], and the data structure used are mainly suffix tree/array and q -grams/samples. The methods used suffix tree/array as their index structure, such as in literature [15], are based on *pigeonhole principle* which is based on the fact that if there are n pigeons but only $n - 1$ pigeonholes, then it must satisfy that there are at least two pigeons in the same pigeonholes. If k is maximal number of tolerant errors, then the query is partitioned into $k + 1$ parts, and only the string with at least one part as its substring need to be verified. And the methods used q -grams/samples are based on the fact that if two strings are similar, they should share a certain number of q -grams/samples, such as the methods proposed in [4, 5, 16]. And a new type of grams with variable lengths, called *VGRAM*, is proposed in [17, 18].

3 Preliminaries

3.1 Block Edit Distance

Let Σ denotes a finite alphabet with size $|\Sigma|$. $s \in \Sigma^*$ denotes a string consists of characters from Σ with length $|s|$. And $s[i]$ represents the i -th character of s , while $s[i, j]$ is a substring of s from $s[i]$ to $s[j]$, where $1 \leq i \leq j \leq |s|$. If $j < i$, $s[i, j]$ is an empty string, denoted by ε . Especially, $s[1, j]$ is a prefix of s , while $s[i, |s|]$ is a suffix of s .

Example 1. Given $\Sigma = \{\mathbf{a}, \mathbf{c}, \mathbf{g}, \mathbf{t}\}$ and $s = \mathbf{acagtc}$, then $|\Sigma| = 4$, $|s| = 6$, $s[3] = \mathbf{a}$, $s[2, 4] = \mathbf{cag}$, $s[4, 3] = \varepsilon$, $s[1, 3] = \mathbf{aca}$ is a prefix of s , and $s[4, 6] = \mathbf{gtc}$ is a suffix.

As an extension of the three classical edit operations, block edit involves another edit operation, block moves. That is, given a string s , we can use a block move operation $move(p_1, l, p_2)$ to move the substring $s[p_1, p_1 + l - 1]$ to the destination position p_2 ($1 \leq p_2 \leq |s| + 1$), where p_1 ($1 \leq p_1 \leq |s|$) is starting position of the substring and l is the length of the substring. For instance,

$$s = \mathbf{abcde} \xrightarrow{move(1,2,4)} s' = \mathbf{cabde}.$$

Definition 1 (Block edit distance). *The block edit distance between two strings is the minimum number of block edit operations to translate a string to the other.*

Problem 1. Given a string collection $S = \{s_1, s_2, \dots, s_n\}$, a query string p and a threshold k , find out a subset $T \in \mathcal{D}$, such that $T = \{s_i | s_i \in S \wedge BED(s_i, p) \leq k\}$.

Example 2. Given $S = \{\text{aaccg}, \text{aagcct}, \text{aaggct}, \text{ccaa}, \text{ctaa}, \text{aagcgt}\}$, $p = \text{ccgaa}$, $k = 1$. Then $BED(\text{aaccg}, \text{ccgaa}) = 1$, since we can move the block **aa** to the rear of **g**. Similar, $BED(\text{ccaa}, \text{ccgaa}) = 1$. So the result set $T = \{\text{aaccg}, \text{ccaa}\}$.

3.2 Lower Bounds of Block Edit Distance

Frequency Distance Given a string s , we call the number of duplicated number of a character in s is the *local frequency*. $f(s)$ denotes the set of local frequencies for different characters in s . Literature [3] defines the frequency distance (FD) between two strings s and p as follows.

$$\begin{cases} posDist = \sum_{i \in \Sigma \wedge f(s)[i] \geq f(p)[i]} f(s)[i] - f(p)[i] \\ negDist = \sum_{i \in \Sigma \wedge f(p)[i] > f(s)[i]} f(p)[i] - f(s)[i] \end{cases} \quad (1)$$

$$FD(s, p) = \max(posDist, negDist).$$

For instance, given $s = \text{aggc}$ is a string over $\{\text{a}, \text{c}, \text{g}, \text{t}\}$, $f(\text{aggc}) = \{\text{a} \cdot 1, \text{c} \cdot 1, \text{g} \cdot 2, \text{t} \cdot 0\}$, where $\text{a} \cdot 1$ means the string contains one **a**. $FD(\text{aggc}, \text{aaccg}) = \max\{2 - 1, 2 - 1 + 2 - 1\} = 2$.

It is proved in [3] that frequency distance is a lower bound of edit distance. Since the block moves operation has no effect on the character's frequency in a string, frequency distance is also a lower bound of block edit distance, i.e. $FD(s, p) \leq BED(s, p)$. Therefore, if $FD(s, p) > k$, then $BED(s, p)$ must be larger than k . We can use it to filter non-candidate strings.

Positional q -Grams Given a string s and a positive integer q , a *positional q -gram* of s is a pair $(i, s[i, i+q-1])$. Intuitively, if two string are similar, they must share enough common grams. In order to let two strings share more common grams and make the lower bound tight, according to the definition in [5], two characters $\#$ and $\$$ that do not belong to Σ are introduced, and a new string s' by prefixing $q-1$ copies of $\#$ and suffixing $q-1$ copies of $\$$ on s is constructed. The set of *positional q -grams* of s , denoted by $G(s, q)$, is obtained by sliding a window of length q over the characters of the new string s' . There are $|s| + q - 1$ positional q -grams in $G(s, q)$. For instance, suppose $q = 3$, and $s = \text{abcde}$, then $G(s, q) = \{(1, \#\#\text{a}), (2, \#\text{ab}), (3, \text{abc}), (4, \text{bcd}), (5, \text{cde}), (6, \text{de}\$), (7, \text{e}\#\#\$)\}$.

It is proved in [5] that if $BED(s, p) \leq k$, then s and p must share at least $(\max(|s|, |p|) - 1 - 3(k-1)q)$ q -grams.

Cluster Trie A *cluster trie* is a tree-like structure, which consists of two types of nodes as follows.

- *Character node*. Each character node contains only one character, which is depicted as a cycle in Fig. 3(a);
- *Frequency node*. Each frequency node records the local frequencies of the character, which is depicted as a box in Fig. 3(a).

If a local frequency of a character c in a string s is m , then we build up a character node n_c whose value is c and let it point to a frequency node n_f with value m . The string s will appear in a certain cluster that rooted at n_f . For example, cluster C_3 contains strings id_2 and id_5 . Both of them contain one g character, two t characters, and three c characters.

Given a collection of strings $S=\{s_1, \dots, s_n\}$. If the local frequencies of a character c in different strings are different, then we build up several frequency nodes to record each local frequency. For instance, in Fig. 2(a), the local frequency of character c in string id_4 is 2, whereas in string id_6 is 5. We build up two frequency nodes rooted at the same character node, for example, there are two frequency nodes with values 2 and 5, respectively, rooted at the character node with value c in the right path of the cluster trie in Fig. 3(a). We call the number of different local frequencies for a set of strings the *variety number of local frequencies*.

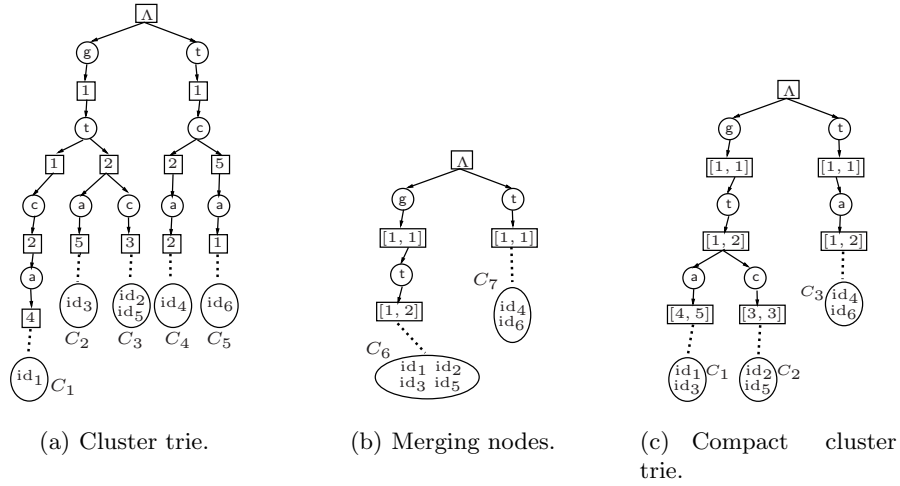


Fig. 3. Constructing a compact cluster trie.

Ordered Cluster Trie For a query q , we can calculate the frequencies of different characters in q . By computing the FD distance between q and each string s in a cluster can determine whether the string s is a candidate. Such

procedure requires comparing q with all paths in a $2TI$ to get all candidates. Ideally, we expect to traverse as few nodes as possible. That is, we should be careful to determine the structure of a constructed cluster trie.

Definition 2 (Partially ordered relationship \prec). Let a and $b \in \Sigma$. $a \prec b$, if and only if character node a is the ancestor node of character node b in $2TI$.

The different partial orders of characters will cause different cluster trie structure. Ideally, we expect to filter a path as early as possible.

Observation 1. The cluster trie with fewer top-level nodes are better than the one with more top-level nodes.

Based on the above observation, in a $2TI$, the character in a top-level character node should appear in most strings and has less frequency nodes. The following are two possible policies to be used to construct partial order for characters.

- **Small variety number of local frequencies take precedence.** Given a collection of strings S , let c_1 and c_2 be two characters in S . If the variety number of c_1 is smaller than that of c_2 , then we build up a partial order: $c_1 \prec c_2$.
- **Large global frequency take precedence.** In a collection of strings S , the *global frequency* of a character c is the number of strings containing c . For any two characters c_1 and c_2 , if their variety number of local frequencies are the same and the global frequency of c_1 is larger than that of c_2 , then we build up a partial order: $c_1 \prec c_2$;

For example, in Fig. 2(a), the variety number of local frequency of g is 1, so we first construct a character node g . The global frequency of t is 6, so we also try to put t as higher as possible in the cluster trie.

4.2 Compact Cluster Tire

Using cluster trie, every path might be traversed to determine whether the strings indexed by the path are candidates or not. There could be too much paths in the cluster trie, which causes high traversal cost. Furthermore, strings in different clusters are skewed distributed. In order to solve the above problems, we merge nodes in the cluster trie to construct a compact cluster trie.

Fig. 3(c) shows a compact cluster trie for the six strings in Fig. 2(a). In a compact cluster tire, the content of a frequency node is a frequency region $[f_1, f_2]$, which means the local frequency of a character is in between f_1 and f_2 . For instance, in the left subtree in Fig. 3(c), t has a frequency node with $[1, 2]$.

We use the following two rules to determine whether two frequency nodes should be merged or not:

- for a frequency region $[f_1, f_2]$, $f_2 - f_1$ should be smaller than a given threshold θ_r . For instance, if a character c has two different local frequencies: 1 and 5, then we should not merge them into one node, since merging them will decrease the filterability greatly;

- the size of each cluster should be balanced, i.e. in between $[\theta_{min}, \theta_{max}]$, where θ_{min} and θ_{max} denote the minimum and maximum size of the cluster, respectively.

Based on the above discussion, we use Algorithm 1 to construct the compact cluster trie. We use Fig. 4.1 to show how Algorithm 1 works.

Algorithm 1: Construction of Compact Cluster Tire

Data: Ordered Cluster Trie OCT , frequency region threshold θ_r and clusters' size range $[\theta_{min}, \theta_{max}]$
Result: Compact Cluster Tire CCT

```

1 repeat
2   foreach cluster  $C_i \in OCT$  do
3     if  $C_i.size < \theta_{min}$  then
4       Reverse traverse  $c_i$ 's path to find out the deepest frequency node
5        $node_d$  which can be merged with its brother node  $node_b$ ;
6       Merged  $node_d$  and  $node_b$ ;
7     if  $C_i.size > \theta_{max}$  then
8       Find character  $\sigma_1$  which is not occurrence in the path indexing  $C_i$ 
9       and is with the largest variety number of local frequencies;
10      Use  $\sigma_1$  to re-partition  $C_i$ ;
11    if  $\theta_{min} \leq C_i.size \leq \theta_{max}$  then
12      Find character  $\sigma_2$  which is not occurrence in the path indexing  $C_i$ 
13      and whose frequency region is smaller than or equal to  $\theta_r$ ;
14      Add  $\sigma_2$  and its frequency region to the path;
15 until all thresholds are satisfied or there is no change of  $OCT$ ;
16  $CCT \leftarrow OCT$ ;
17 return  $CCT$ ;
```

Let frequency region threshold θ_r be 1 and $[\theta_{min}, \theta_{max}]$ be $[2, 2]$. In the first iterator, in Line 2, Algorithm 1 scans all clusters. In Fig. 3(a), the number of strings in C_1 is less than θ_{min} , the two frequency nodes under t should be merged to make a merged cluster contain more strings. These two nodes are merged to a new node with frequency region $[1, 2]$ that points to a new cluster $C_6 = \{id_1, id_2, id_3, id_5\}$. Then C_4 is processed and cluster C_4 and C_5 are merged into C_7 in Fig. 3(b). Then the second iteration is started. Since there are four strings in C_6 , the cluster should be partitioned. The local frequency of a and c are both 2, so it randomly chooses one (a is chosen in this case) and C_6 is partitioned to two parts: $\{id_1, id_3\}$ and $\{id_2, id_5\}$. Since the local frequencies of a in C_7 are 1 and 2, respectively, a and range $[1, 2]$ are added under $t \rightarrow [1, 1]$. A compact cluster trie is shown in Fig. 3(c).

Notice that the merger and depth limitation will cause more false positives, for instance, given $p = taagcga$ and $k = 1$, using the cluster trie in Fig. 3(a),

the candidate string is id_1 , while using the compact cluster trie in Fig. 3(c), the candidates are id_1 and id_3 . Although compact cluster trie decrease the filterability of the first tier, the size of the cluster trie is reduced significantly. There are 23 nodes and 5 paths in Fig. 3(a), but only 13 nodes and 3 paths in Fig. 3(c). Meanwhile, the clusters become balanced, which is good for q -gram based filter in the second tier.

5 Search Candidate Strings

Given a query string p and a block edit distance threshold k , we first use $2TI$ to filter non-candidate strings in the string collection S . Then we adopt the Greedy Algorithm proposed in [1, 1] to calculate block edit distance between each candidate c and p . If $BED(c, p) \leq k$, then c is the answer.

Now we use an example to show how $2TI$ works. Given $p = taagcga$, $k = 1$ and the string collection is as illustrated in Fig. 2(a). First, the frequency set of p is calculated $f(p) = \{g \cdot 2, t \cdot 1, c \cdot 1, a \cdot 3\}$. The first tier of $2TI$, compact cluster trie, is traversed by using depth first search (DFS), lines 5–24 in Algorithm 2. To evaluate how many errors will be brought in when $f(p)$ is matched with a (whole or partly) path in the trie, *error cost* is used. We use *posDist* to record the number of deletion operations applied to the path, and *negDist* to record the number of addition operations applied to the path. Add error cost is $\max(posDist, negDist)$. Obviously, if error cost is bigger than k , the visit of a path can be terminated.

It is started at the root of the structure, and the node contained character g is visited afterwards, which means that the ranges in its sons are frequency regions of g . Since there are two g in p and the frequency region is $[1, 1]$, the *negDist* is set to 1. And the error cost is $\max(0, 1) = 1$ which equals to k , the traversal continues. It is a t node in the next, and the son of the t node contains $[1, 2]$. The local frequency of t in p is 1 which is in between $[1, 2]$, so the error cost is not changed. Then the node a is accessed, there are two sons of this node. The one contains $[4, 5]$ is visited firstly. Since there are three a in p , *posDist* is set to 1. And the error cost of this node is $\max(1, 1) = 1$, so cluster C_1 is a candidate. Then get back to the a node. Since all sons of a node are visited, then get back to the parent of a node. The remaining of the trie is traveled in this way, and the finally candidate cluster is C_1 . Algorithm 2 presents the complete filter algorithm.

So, here after the first tier filter using compact cluster trie, we use q -gram inverted lists in the second tier filter to refine the candidates. To use the q -gram inverted list, the q -grams set of query p is generated, which is $G(p, 2) = \{\#t, ta, aa, ag, gc, cg, ga, a\}$ in this case. Then querying these q -grams in the q -grams index to count the number of common q -grams with the candidate strings. In this case, there are six q -grams shared by p and $taacgaa$, and the query $taagcga$ and $agaatta$ share five q -grams. According to the minimum number of shared q -grams is $(\max(|s|, |q|) - 1 - 3(k - 1)q) = \max(7, 7) - 1 - 3 \times (1 - 1) \times 2 = 6$,

Algorithm 2: DFS Filtering

Data: The first tier of *2TI CCT*, query p and error threshold k
Result: The candidate cluster set S_C

```
1  $f(p) \leftarrow$  the frequency set of  $p$ ;  
2  $posDist \leftarrow 0, negDist \leftarrow 0, ec \leftarrow 0$ ;  
3 Let  $nStack$  and  $disStack$  are stacks of nodes and pair  $(posDist, negDist)$ ;  
4  $p_n \leftarrow$  the root of  $CCT$ , push  $p_n$  to  $nStack$ ;  
5 repeat  
6   if All sons of  $nStack.top()$  are visited then  
7     if  $nStack.top()$  is a character node then  
8        $(posDist, negDist) \leftarrow disStack.top()$ ;  
9        $disStack.pop()$ ;  
10       $nStack.pop()$ ;  
11    else  
12       $p_n \leftarrow$  one of  $nStack.top()$ 's unvisited sons;  
13      Mark  $p_n$  visited and push  $p_n$  to  $nStack$ ;  
14      if  $p_n$  is a cluster then  
15        Add  $p_n$  to  $S_C$ ;  
16      else  
17        if  $p_n$  is a character node then  
18           $c \leftarrow p_n.character$ ;  
19          Push  $(posDist, negDist)$  to  $disStack$ ;  
20        else  
21          if  $f(s)[c] < p_n.min$  then  $posDist+ = p_n.min - f(s)[c]$ ;  
22          if  $f(s)[c] > p_n.max$  then  $negDist+ = f(s)[c] - p_n.max$ ;  
23           $ec = \max(posDist, negDist)$ ;  
24          if  $ec > k$  then  $nStack.pop()$ ;  
25 until  $nStack$  is empty ;  
26 return  $S_C$ ;
```

the string `agaatta` dose not satisfy this bound, so there is only one string `taacgaa` may satisfy that $BED(\text{taacgaa}, \text{taagcga}) \leq 1$.

6 Performance Evaluation

Our experiments are written by C++, and run on PC with 256M RAM, 2.00GHz CPU, Windows[®] XP. Two real data sets, the book names⁴, S_1 and titles of papers⁵, S_2 , are used to evaluate our indices. The alphabet's size of S_1 is 86, and there are totally 96 different characters in S_2 . The average length of strings in S_1 is 18.12, and it is 65.7 in S_2 .

⁴ Questia, <http://www.questia.com/>

⁵ DBLP, <http://www.informatik.uni-trier.de/~ley/db/>

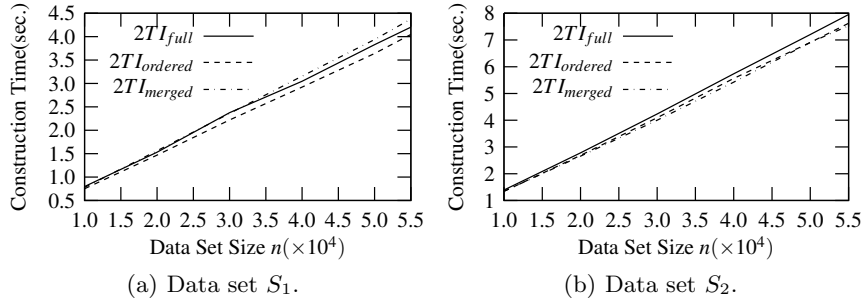


Fig. 4. Construction time v.s. the size of data set.

6.1 Construction Overhead

In this section, we show the performance of constructing full cluster and compact cluster. $2TI_{full}$ denotes the two-tier index with the full cluster as its first tier, $2TI_{ordered}$ denotes the two-tier index with reordered full cluster as its first tier, and $2TI_{merged}$ denotes the two-tier index with compact cluster as its first tier.

Time v.s. n As shown by Fig. 4, the construction time grows linearly with the size of data set, n . The construction time of the three indices are almost the same. The cost of counting local frequencies of characters are too small to be seen. This is mainly because the alphabet reorder rules reduce the number of nodes in $2TI_{ordered}$. It also can be seen that when the size of S_1 is large, construction time of $2TI_{merged}$ is more than that of $2TI_{full}$, but the construction time of $2TI_{merged}$ is still smaller than that of $2TI_{full}$ at S_2 , it is because that the average string length of S_1 is short, and it is more likely to repeat in S_1 when the size is large, reducing the addition of node in $2TI_{full}$.

Fig. 4(a) and Fig. 4(b) also show that the size of alphabet and average length of string affect the construction of indices: the larger they are, the more time is needed to construct the indices.

Size v.s. n As illustrated by Fig. 5, all indices' sizes linearly grow with size of data set, n . The alphabet reorder rules excellently reduce the size of $2TI_{full}$ of S_1 but only reduce the size of S_2 's ordered cluster trie a little. It is because the average string length is long and variety number of local frequencies is large in S_2 , there may be many branches in a node of the $2TI_{full}$ and it is hard to reduce the size. It also can be seen from Fig. 5 that the $2TI_{merged}$ s always keep much smaller.

6.2 Effect of Filtering

Two query sets, Q_1 and Q_2 are used to evaluate the filtering performance of our indices. Q_1 and Q_2 are formed by 100 randomly selected strings from data

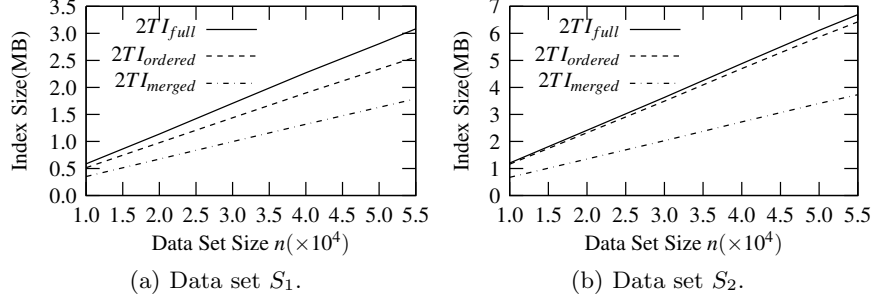


Fig. 5. Size of indices v.s. the size of data set.

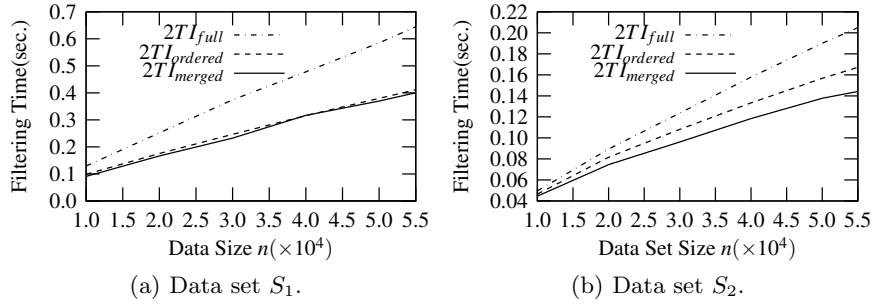


Fig. 6. Filtering time v.s. the size of data set.

set S_1 and S_2 respectively, some random noises are also be added to the strings in both sets. And the query sets are run 100 times on their corresponding data set's indices (Q_1 and Q_2 correspond to S_1 and S_2 respectively). The average of these results is used as the final result.

Time v.s. n The error threshold k is set to 9. As Fig. 6 illustrated, The filtering time is linear with the size of data set. And even when the size of data set is 5.5×10^4 , the worst filtering time is 0.63s and 0.28s of S_1 and S_2 respectively. $2TI_{merged}$ has the best performance, it is because the alphabet reorder rules and merger reduce the number of nodes in the $2TI_{merged}$. The filtering time of $2TI_{merged}$ and $2TI_{ordered}$ is almost same in Fig. 6(a) is mainly because the merger cause of false positives which raise the time cost of the second tier of $2TI_{merged}$.

Time v.s. k The size of string collection n is fixed to 55000. The filtering time of those indices all quickly grow with the increasing of threshold k as illustrated by Fig. 7, it is because larger k causes more nodes needed to be tested. It is also shown that $2TI_{merged}$ has the best performance again. But differently, $2TI_{full}$ performs better than $2TI_{ordered}$ at S_2 when k is small, it may be because some

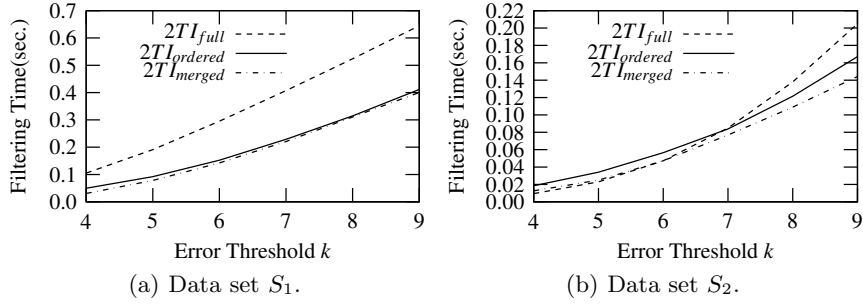


Fig. 7. Filtering time v.s. error threshold.

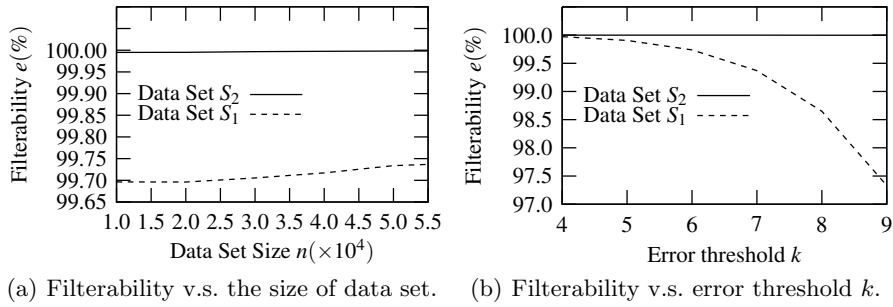


Fig. 8. Filterability.

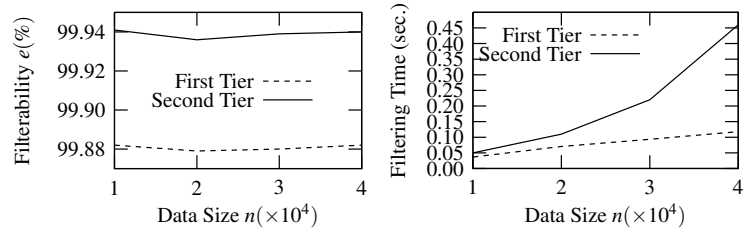
characters at the top of $2TI_{full}$ do not occur in the query string, and the searches on $2TI_{full}$ terminal earlier.

Filterability We use Equation 2 to evaluate the filterability of our index structure.

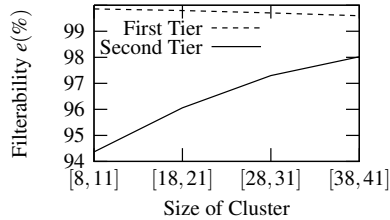
$$e = 1 - \frac{|S_C|}{|S|}, \quad (2)$$

where S_C denotes the candidate set generated by filtering with our indices, and S denotes the string collection.

Fig. 9 shows the results. Fig. 8(a) shows the filterability when we fixed threshold $k = 6$ and varied the size of data set n . Fig. 8(b) shows the filterability when we fixed varying $n = 55000$ and varied k . We can see the proposed two-tier index structure provided good filterability. In Fig. 8(a), e was closed to 1 for the data set S_2 because the frequency distance between query and S_2 are more different than between query and S_1 . When the size of data set increased, e increased slowly for the data set S_1 . The result also shows that our indices are scalable. For similar reason, in Fig. 8(b), e kept stable, i.e. near to 1, for S_2 . Although e for S_1 decreases quickly when k grew, its value is still larger than 0.97 when k was less than 9.



(a) Filterability v.s. data set size. (b) Filtering time v.s. data set size.



(c) Filterability of cluster and q -gram.

Fig. 9. Cluster Trie v.s. Q -gram Index.

6.3 Cluster Trie v.s. Q -gram Index

In this section, we show the performance of using cluster index and q -gram index separately. The data set used is S_2 , and k is fixed to 1.

As Fig. 9(a) illustrated, with the growth of data set, the filterability of both cluster trie and q -gram index dose not change so much, the filterability of cluster trie is nearly 99.88%, and is nearly 99.94% for q -gram index. Obviously, the filterability of q -gram is much powerful than cluster trie's, i.e. using q -gram may generate a smaller candidate set.

However the filtering time of q -gram increases sharply with the growth of data set size, as illustrated in Fig. 9(b). But the filtering time of cluster trie only slightly increases, i.e. cluster trie has a better response time.

So first use cluster trie to generate a candidate set, then use q -gram to refine the candidate set can benefit both the advantage of cluster trie and q -gram and avoid the disadvantage.

The filterability of q -gram in Fig. 9(c) is based on the candidate set generated by cluster trie. As Fig. 9(c) illustrated, with the increase of cluster's size, the filterability of cluster trie is descendent, i.e. the size of its candidate set is increased. And the increase of q -gram's filterability is mainly because that the size of final candidate generated by q -gram dose not change.

7 Conclusion

In this paper we have developed a two-tier index structure, called $2TI$, to reduce candidate strings for approximate string queries with block moves. The first

tier of index is based on the idea of utilizing the advantages of the existing two mature filter strategies: frequency distance and positional q -gram inverted lists to enhance the filterability of non-candidate strings. We extend the idea of frequency distance to build up the first tier index and classify strings into small size of non-overlapped strings, then for each clusters, we use q -gram based inverted lists as second tier index to get tight lower bound of block edit distance. We present our experiments on real data sets to evaluate our technique and show the proposed index structure can provide a good performance.

References

1. Dana Shapira and James A. Storer: Edit distance with move operations. In: CPM(2002)
2. Dana Shapira and James A. Storer: Edit distance with move operations. Journal of discrete algorithms, Vol. 5, 380-392(2007)
3. Tamer Kahveci and Ambuj K. Singh: An Efficient Index Structure for String Databases. In: VLDB(2001)
4. Esko Ukkonen: Approximate String-matching with q -grams and Maximal Matches. Theoretical Computer Science, Vol.92, 191-211(1992)
5. Luis Gravano, Panagiotis G. Ipeirotis and H. V. Jagadish: Approximate String Joins in a Database (Almost) for Free. In: VLDB(2001)
6. Maxime Crochemore and Wojciech Rytter: Text Algorithms. Oxford University Press, UK(1995)
7. Dan Gusfield: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational. Cambridge University Press, UK(1997)
8. Maxime Crochemore and Wojciech Rytter: Jewels of Stringology. World scientific, Singapore(2002)
9. Gonzalo Navarro. A Guided Tour to Approximate String Matching. ACM computing surveys(CSUR), Vol.33, 31-88(2001)
10. Daniel Lopresti and Andrew Tomkins: Block Edit Models for Approximate String Matching. Theoretical computer science, Vol.181, 159-179(1997)
11. Graham Cormode and S. Muthukrishnan: The String Edit Distance Matching Problem with Moves. In: ACM-SIAM symposium on Discrete algorithms(2002)
12. Graham Cormode and S. Muthukrishnan: The String Edit Distance Matching Problem with Moves. ACM Transactions on Algorithms(TALG), Vol.3, 2-21(2007)
13. Haim Kaplan and Nira Shfir: The Greedy Algorithm for Edit Distance with Moves. Information Processing Letters, Vol.97, 23-27(2006)
14. Gonzalo Navarro, Ricardo Baeza-yates, Erkki Sutinen and Jorma Tarhio: Indexing Methods for Approximate String Matching. IEEE Data Engineering Bulletin, Vol.24, 19-27(2001)
15. Gonzalo Navarro and Ricardo Baeza-yates: A Hybrid Indexing Method for Approximate String Matching. Journal of Discrete Algorithms, Vol.1, 205-239(2000)
16. Min-soo Kim and Kyu-young Whang and Jae-gil Lee and Min-jae Lee: n-Gram/2L: A Space and Time Efficient Two-Level n-Gram Inverted Index Structure. In: VLDB(2005)
17. Chen Li, Bin Wang, Xiaochun Yang: VGRAM: Improving Performance of Approximate Queries on String Collections Using Variable-Length Grams. In: VLDB(2007)
18. Xiaochun Yang, Bin Wang, Chen Li: Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In: SIGMOD(2008)