

Approximate String Matching by Position Restricted Alignment *

Manish Patil
School of EECS
Louisiana State University
USA
mpatil@csc.lsu.edu

Rahul Shah
School of EECS
Louisiana State University
USA
rahul@csc.lsu.edu

Xuanting Cai
Department of Mathematics
Louisiana State University
USA
xcai1@math.lsu.edu

Seung-Jong Park
School of EECS
Louisiana State University
USA
sjpark@csc.lsu.edu

Sharma V. Thankachan
School of EECS
Louisiana State University
USA
thanks@csc.lsu.edu

David Foltz
Department of Biological
Sciences
Louisiana State University
USA
dfoltz@lsu.edu

ABSTRACT

Given a collection of strings, goal of the approximate string matching is to efficiently find the strings in the collection that are similar to a query string. In this paper, we focus on edit distance as measure to quantify the similarity between two strings. Existing q -gram based methods to address this problem use inverted indexes to index the q -grams of given string collection. These methods begin by generating the q -grams of query string (disjoint or overlapping) and then merge the inverted lists of these q -grams. Several filtering techniques have been proposed so as to segment inverted lists to relatively shorter lists thus reducing the merging cost. We use a filtering technique which we call as “position restricted alignment” that combines well known length filtering and position filtering to provide more aggressive pruning. We then provide an indexing scheme that integrates the inverted lists storage with the proposed filter thus enabling us to auto-filter the inverted lists. We evaluate the effectiveness of the proposed approach by thorough experimentation.

1. INTRODUCTION

Strings form a fundamental data type in computer systems and string searching has been extensively studied since the inception of computer science. Approximate string matching or string similarity search takes a set of strings and a query string as input, and outputs all the strings in the set

*This work is supported in part by US NSF Grant CCF-1017629, CCF-1218904 (Rahul Shah, Sharma V. Thankachan), US NSF MRI Grant 0821741 (Seung-Jong Park, Manish Patil, Xuanting Cai) and US NSF Grant 1036358 (David Foltz, Xuanting Cai)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EDBT/ICDT '13, March 18 - 22 2013, Genoa, Italy
Copyright 2013 ACM 978-1-4503-1599-9/13/03 ...\$15.00.

that are similar to the query string. It is a central problem to many real world applications in the field of information retrieval, bioinformatics, data cleaning etc. There are many functions to quantify the similarity of two strings, such as Jaccard similarity, Cosine similarity, and Edit distance. In this paper, our focus is on edit distance. The edit distance of two strings is the minimum number of single-character edit operations (i.e., insertion, deletion, and substitution) needed to transform one string to another. For example, the edit distance between “masachusatts” and “massachusetts” is 2. Edit distance is useful when a search engine is required to perform spelling corrections and/or needs to handle half-word or phrase queries. It is also widely used in many biological tasks like genome matching, alignment, mass spectrometry and DNA sequencing.

Existing methods to address this problem can be broadly classified into two categories. The first one is based on the use of trie (suffix tree) data structure for indexing the set of input strings. These methods rely on the fact that edit distance between two strings is bounded by the edit distance between their prefixes. This allows us to filter out strings having prefix with edit distance higher than the required threshold with respect to some prefix of the query string by systematically navigating trie (suffix tree). However, these methods are usually inefficient for long strings as they have a small number of shared prefixes. Moreover navigation cost of trie (suffix tree) suffers from exponential dependence on the pattern length as well as the edit distance threshold in worst case.

A more common approach to tackle the approximate string matching is q -gram based and makes use of inverted indexes to index the q -grams. A q -gram is a consecutive substring of a string with size q that can be used as a signature of the string. The key idea these methods exploit is that two strings are similar only if their q -gram sets share enough common grams. A lower bound on the number of common grams depends on the length of the grams i.e., q as well as the edit distance threshold. Thus, given a query string, these methods first generate its q -grams, retrieve the corresponding inverted lists, and then merge the lists to find strings similar to the query string. These methods also use various filtering techniques to prune strings (*length filtering*

and *position filtering* being the most common), effectively reducing the size of lists to be merged, and thus reducing query time. However these methods have following limitations:

- Most of the existing methods use “one-for-all” principle and fix q at the time of index construction. As known from the literature, a larger value of q results in a smaller size of inverted lists, which may reduce the cost of merging, thus improving the query performance [12].
- Applying filters to prune out the candidate strings during query execution can be expensive in terms of computational cost.

We observe that the ability to store grams for multiple values of q can allow us to select the appropriate value of q based on the given edit distance threshold and achieve good query performance over wide range of edit distance threshold. Moreover, overhead of applying filters during query execution can be avoided if inverted lists of the q -grams are maintained so as to efficiently retrieve only those strings which satisfy the filter conditions to be applied.

Our contributions: We first propose the use of suffix tree leafs for encoding inverted lists of grams for all values of q . We then use “Position Restricted Alignment” that combines well known length filtering and position filtering to provide more aggressive pruning. Finally we show how the leafs of the suffix tree can be rearranged so that only the strings satisfying the position restricted alignment are retrieved. This enables us to auto-filter the inverted lists by integrating their storage with the filter application.

Outline: We review related work in Section 2. Section 3 gives the preliminaries. Section 4 presents the proposed framework for answering approximate string matching queries whereas Section 5 focuses on “Position Restricted Alignment” based filtering. Section 6 discusses practical improvements to the framework. We have conducted extensive experiments to evaluate the proposed techniques and the results are reported in Section 7.

2. RELATED WORK

There are many studies on approximate string search, in which given a set of strings, a query string, a similarity function, and a threshold, goal is to find all the strings having similarities to the query string within the threshold. Several algorithms [2, 12, 11] have been proposed for answering approximate string queries efficiently. Their main strategy is to use various filtering techniques to improve the performance. Traditionally, fixed length q -grams are widely used for answering edit similarity queries to utilize the effectiveness of count filtering in pruning candidates. In [12] authors have proposed to preprocess the string collection to obtain a dictionary of high-quality grams of variable lengths based on gram frequencies. Query partitioning using such a dictionary can help to achieve better performance than using a fixed length q -grams partitioning. However most of the existing algorithms assume a static q determined at the index construction, whereas we make an attempt to adaptively select the appropriate gram length based on the required edit distance threshold on the fly during query execution. Moreover, applying various filters is an independent step in the existing algorithms. We alleviate the overhead of applying filters during query execution by integrating the filter

conditions with inverted lists storage during index construction itself. As these algorithms need to merge the inverted lists of grams generated from the strings, efficient merging techniques have also been developed [19, 11]. We use a simple “ScanCount” method [11] for merging the lists, which is known to achieve a good performance when combined with various filtering techniques.

Due to the difficulties in selecting appropriate edit distance threshold while querying, lot of research has been devoted to the problem of top- k string similarity search recently. Given a collection of strings and a query string, top- k string similarity search returns the top- k most similar strings to the query string. Kahveci et al. [9] proposed the solution which first converts a set of contiguous substrings into a Minimum Bounding Rectangle (MBR) and then use it to estimate the edit distance threshold of top- k answers. Yang et al. [22] proposed a gram-based method that increments the edit threshold in steps and adaptively selects the gram length to be used. Though the intuition behind using different gram lengths is similar to our approach, we maintain a unified index as opposed to multiple inverted indexes maintained in [22]. Recent studies on this problem also includes B^+ -tree based approach by Zhang et al. [23] and trie based approach by Deng et al. [5]. The former traverses the B^+ -tree nodes iteratively and computes a lower bound of edit distances between the query and strings under the node. This bound is then used to update the edit distance threshold. Whereas the later traverses the trie and progressively computes the edit distance between query string and strings grouped by a common prefix.

A closely related and extensively studied problem is “string similarity joins” [18, 7, 3, 1, 10, 21, 13, 6]. Given two sets of strings, a similarity join finds all similar string pairs. The approximate string searching problem could be treated as a special case of similarity join. It is known that behavior of an algorithm could be very different while answering approximate string matching queries from that of answering join queries. Therefore, though the algorithms developed for similarity joins can be adapted for edit similarity queries, they might not be efficient. Approximate string matching is an important problem and needs to be investigated separately, which is the focus of this paper.

In the literature, “approximate string matching” also refers to the problem of finding a pattern string approximately in a text [15, 4] i.e., given a query string and a text string, goal is to find all substrings of the text that are similar to the query. The problem studied in this paper is different, as we want to report the strings similar to a query string from a given collection of strings. As most of the techniques introduced for the former problem rely on suffix tree navigation they lead to poor performance when adapted to our problem.

3. PRELIMINARIES

Let Σ be an alphabet. For a string s of the characters in Σ , we use “ $|s|$ ” to denote the length of s , “ $s[i]$ ” to denote the i -th character of s (starting from 1), and “ $s[i..j]$ ” to denote the substring from its i -th character to its j -th character.

3.1 Suffix Trees, Compressed Suffix Trees

Given a string $s[1..n]$, a substring $s[i..n]$ with $1 \leq i \leq n$ is called a suffix of s . The lexicographic arrangement of all n suffixes of s in a compact trie is called the *suffix*

tree of s [20], where the i th leftmost leaf represents the i th lexicographically smallest suffix. Each edge in the suffix tree is labeled by a character string and for any node u , $\text{path}(u)$ is the string formed by concatenating the edge labels from root to u . For any leaf v , $\text{path}(v)$ is exactly the suffix corresponding to v . Given a gram Q , a node u is defined as the *locus node* of Q if it is the closest node to the root such that Q is a prefix of $\text{path}(u)$. Therefore all distinct strings in which given gram i.e. Q appears as a substring along with the position of occurrence can be reported by simply scanning the leaves in the range $[l \dots r]$, where l and r represent the left-most and right-most leaves in the subtree rooted at node u respectively. The range $[l \dots r]$ is called suffix range of Q . Similarly *generalized suffix tree* (GST) is a compact trie which stores all suffixes of all strings in a given collection \mathcal{S} of strings. Large space overhead of suffix tree prohibits its use in practice. Instead we use a space efficient version of suffix tree known as Compressed suffix tree (CST). Several variants of CSTs have been proposed till date, we use the one by Ohlebusch et al. [16].

3.2 Wavelet Tree

Let $A[1..n]$ be an array of length n , where each element $A[i]$ is a symbol drawn from a set Σ . The *wavelet tree* (WT) [8] for A is an ordered balanced binary tree on Σ , where each leaf is labeled with a symbol in Σ , and the leaves are sorted alphabetically from left to right. Each internal node u represents an alphabet set Σ_u , and is associated with a bit-vector B_u . In particular, the alphabet set of the root is Σ , and the alphabet set of a leaf is the singleton set containing its corresponding symbol. Each node partitions its alphabet set among the two children (almost) equally, such that all symbols represented by the left child are lexicographically (or numerically) smaller than those represented by the right child. For the node u , let A_u be a subsequence of A by retaining only those symbols that are in Σ_u . Then B_u is a bit-vector of length $|A_u|$, such that $B_u[i] = 0$ if and only if $A_u[i]$ is a symbol represented by the left child of u . Indeed, the subtree from u itself forms a wavelet tree of A_u . Note that B_u is augmented with a data structure with small overhead [17] to support constant-time bit-rank and bit-select operations. Moreover we do not store A_u 's explicitly to reduce the space requirement. The following is a useful lemma on wavelet trees.

LEMMA 3.1. *The wavelet tree of A can be maintained in $n \log |\Sigma|(1 + o(1))$ bits, such that given a range $[l, r]$ and a symbol $\pi \in \Sigma$ as the input, all those $i \in [l, r]$ with $A[i] = \pi$ can be reported in $O((1 + \text{output}) \log |\Sigma|)$ time.*

By using multiple wavelet trees and the above lemma, we shall answer more sophisticated orthogonal range searching queries as follows:

LEMMA 3.2. *A given set of n 3-dimensional points in an $[0, n-1] \times [0, \alpha-1] \times [0, \beta-1]$ grid can be maintained in $n(\log \alpha + \log \beta)(1 + o(1))$ bits such that all those points with its x -coordinate within $[l, r]$, and $\pi \in [0, \alpha-1]$ and $\pi' \in [0, \beta-1]$ as its y and z coordinates respectively can be reported in $O((1 + \text{output})(\log \alpha + \log \beta))$ time, where l, r, π and π' are input parameters.*

Id	Strings	Length
s_1	AAACTGTGC	9
s_2	AACTGTC	7
s_3	CTAATCT	7
s_4	GCGTC	5
s_5	GCGTCGT	7
s_6	TCAACCGTACG	11
s_7	TCCTATAAA	9

Table 1: A collection \mathcal{S} of strings

3.3 Approximate String Matching

Given a string r and a collection of strings \mathcal{S} , an approximate string query finds all strings in \mathcal{S} similar to r . In this paper, we use edit distance to quantify the similarity between two strings. Formally, the edit distance between two strings r and s , denoted by $\text{ed}(r, s)$, is the minimum number of single-character edit operations (i.e., insertion, deletion, and substitution) needed to transform r to s . In this paper two strings are similar if their edit distance is not larger than a specified edit-distance threshold τ . We formalize the problem of approximate string matching as follows.

DEFINITION 3.3. *Given a non-negative integer τ , a string s and a collection of string \mathcal{S} , an approximate string query finds all pairs (r, s) with $s \in \mathcal{S}$ such that $\text{ed}(r, s) \leq \tau$.*

For example, consider the strings in Table 1. Suppose threshold $\tau = 2$. Then strings s_1 and s_2 are similar to query string $r = \text{"AACTGTGC"}$ as their edit distance is not larger than τ .

4. A FRAMEWORK FOR APPROXIMATE STRING QUERY

This section gives overview of the proposed framework and describes the naive way of applying the known length and position filters.

We begin by partitioning the given query string r , into $\tau+1$ disjoint segments. Here for simplicity we assume $|r| \geq \tau+1$. The idea behind such a partitioning is that, if a string s has no substring that matches a segment of r , s cannot be similar to r . Following lemma formally summarizes this idea.

LEMMA 4.1. *Given a string r with $\tau+1$ segments and a string s , if s is similar to r within threshold τ , s must contain a substring which matches a segment of r .*

A common partitioning technique is to divide the query string into $\tau+1$ segments each with length $\lfloor |r|/(\tau+1) \rfloor$ except the last $|r| \bmod (\tau+1)$ segments which have length $\lceil |r|/(\tau+1) \rceil$. For example, consider a string $r = \text{"AACTGTGC"}$ and suppose $\tau = 2$. We partition it into 3 segments, with first segment of length $\lfloor 8/3 \rfloor = 2$ and last 2 segments having length $\lceil 8/3 \rceil = 3$. Thus r has three segments $\{\text{"AA"}, \text{"CTG"}, \text{"TGC"}\}$. Since strings s_4, s_5 have no substrings matching segments of r , they are not similar to r . We introduce the notation $r(i)$ to represent the i th segment of r after partitioning. We refine this partitioning technique later in Section 6.1.

To be able to efficiently search the segments of the query string, we build a GST on the string collection \mathcal{S} . Let n be the total length of the strings in the collection \mathcal{S} . In addition

to GST, we maintain two arrays s_ids and s_pos each of length n . $s_ids[i]$ stores the identity of the string to which the i th lexicographically smallest suffix belongs to i.e. suffix corresponding to i th leftmost leaf of GST. Whereas the starting position of that suffix in the corresponding string i.e. string with id $s_ids[i]$ is maintained in $s_pos[i]$. These two arrays essentially stores the information about the leafs of GST and thus eliminate the need to compute the same during query answering. A straight forward approach to find candidate strings that are potentially similar to r is to enumerate all the strings which have at-least one of the segments of r as its substring. This can be achieved by simply scanning the array $s_ids[l_i \dots r_i]$, where $[l_i \dots r_i]$ represents the suffix range of i th segment of r i.e. $r(i)$ for $1 \leq i \leq \tau + 1$.

We can reduce the number of potential candidates by applying well known length filtering and position filtering:

- Length filtering: The length of a string s that is within edit distance τ from query string r is bounded by the equation: $||r| - |s|| \leq \tau$
- Position filtering: Let s be the string which has edit distance less than or equal τ with respect to string r . Without loss of generality let s contains a substring $s(i)$ that matches segment $r(i)$. By Lemma 4.1, there is at-least one such segment since $ed(r, s) \leq \tau$. Also let segment $r(i)$ has starting position $r(i)^{sp}$ in r and substring $s(i)$ has starting position $s(i)^{sp}$ in s . As noted in [13], if alignment of two strings produced by matching $s(i)$ and $r(i)$ gives edit distance less than or equal to threshold τ then $|r(i)^{sp} - s(i)^{sp}| \leq \tau$.

The above filters can be easily applied by scanning the s_ids and s_pos arrays simultaneously. While scanning for the suffix range $[l_i \dots r_i]$ for segment $r(i)$, we ignore the string $s_ids[j]$ if its length is not in the range $[max(0, |r| - \tau) \dots |r| + \tau]$ to apply length filtering. Similarly we ignore the string $s_ids[j]$ if its corresponding starting position i.e. $s_pos[j]$ is not in the range $[max(0, r(i)^{sp} - \tau) \dots r(i)^{sp} + \tau]$ to apply position filtering. Here we note that a particular string s may appear multiple times in the suffix range $[l_i \dots r_i]$ i.e. there can be more than one possible alignments of r and s based on matching of segment $r(i)$. A string s becomes a possible candidate due to segment $r(i)$ if at-least one of its alignment satisfies the position filtering. Out of the strings listed in Table 1 that have substring matching with at-least one of the segments of query string $r = \text{"AACTGTGC"}'$, string s_6 can be pruned using length filtering whereas string s_7 can be pruned using position filtering. Thus, we are now left with only s_1, s_2 and s_3 as candidate strings.

Finally, each candidate string that satisfies both filters described above is subjected to verification that involves computing its actual edit distance with the query string. Though number of interesting optimizations to the verification process have been proposed so far [13], we use verification algorithm by Ukkonen along with simple early termination criteria. We defer more details about optimizing verification step to Section 6.3.

5. POSITION RESTRICTED ALIGNMENT

In this section, we describe the position restricted alignment, which provides more aggressive filtering than applying both position and length filtering independently. Recall the terminologies from the previous section. Let us assume that

we have two strings r and s with $r(i)$ and $s(i)$ as their substrings respectively such that $r(i) = s(i)$. Further $r(i)^{sp}$ and $s(i)^{sp}$ represents the starting positions of $r(i)$ and $s(i)$ within r and s respectively. Now we partition the string s into $\overleftarrow{s}(i), s(i), \overrightarrow{s}(i)$, where $\overleftarrow{s}(i)$ and $\overrightarrow{s}(i)$ are the parts of s respectively on the left and right side of the segment $s(i)$. Similarly r is partitioned into $\overleftarrow{r}(i), r(i), \overrightarrow{r}(i)$, where $\overleftarrow{r}(i)$ and $\overrightarrow{r}(i)$ are the parts of r respectively on the left and right side of the segment $r(i)$. Then position restricted alignment filtering is based on the following observations:

$$ed(r, s) \leq \tau$$

if and only if,

$$ed(\overleftarrow{r}(i), \overleftarrow{s}(i)) + ed(\overrightarrow{r}(i), \overrightarrow{s}(i)) \leq \tau$$

We note that $ed(\overleftarrow{r}(i), \overleftarrow{s}(i))$ captures the essence of position filtering whereas $ed(\overrightarrow{r}(i), \overrightarrow{s}(i))$ captures the essence of length filtering. Continuing the example from previous section, we are left with candidate strings $\{s_1, s_2, s_3\}$ after applying length and position filtering. If we apply position restricted alignment we can decide to prune string s_3 , which satisfies both length as well position filter. We note that position restricted alignment is more tight filtering condition and will filter out any string that can be filtered by either length filtering or position filtering.

By expanding the above equation using simple length filtering we derive the following results.

$$||\overleftarrow{r}(i)| - |\overleftarrow{s}(i)|| + ||\overrightarrow{r}(i)| - |\overrightarrow{s}(i)|| \leq \tau$$

$$|r(i)^{sp} - s(i)^{sp}| + ||r| - r(i)^{sp}| - (|s| - s(i)^{sp})| \leq \tau$$

By solving the above inequality, we can obtain $O(\tau^2)$ solutions in the form of $(s(i)^{sp}, |s|)$ pair. Let C be the set of all such possible pairs. Therefore, our task is now reduced to finding the strings such that pair $(s_pos[j], |s_ids[j]|) \in C$ and $j \in [l_i \dots r_i]$, where $[l_i \dots r_i]$ is the suffix range of partition segment $r(i)$. To answer such a query efficiently we use the data structure described in Lemma 3.2. For us to be able to use the data structure, we simply map the i th left-most leaf of GST to a 3 dimensional point $(i, s_pos[i], |s_ids[i]|)$.

Though the idea behind position restricted alignment is similar to the one proposed in [13], there are primarily two distinctions with respect to our work: (1) In [13], authors goal is to answer similarity join queries assuming fixed edit distance threshold (τ), whereas our indexing technique is independent of τ . (2) The algorithm in [13] needs to access multiple inverted lists and then apply the filtering condition, whereas we do not need to apply the filter at the time query execution.

6. PRACTICAL IMPROVEMENTS

This section describes how the framework proposed in earlier sections can be extended to incorporate more filtering techniques to improve query performance. We also briefly discuss the verification process that our framework uses towards the end of the section.

6.1 Incorporating count filtering

Instead of partitioning the string r into $\tau + 1$ segments, we can decide to partition it into $\tau + k$ segments for $k \geq 1$.

As a consequence, a string s qualifies as a candidate only if it has substrings matching at-least k segments of the query string r . Requirement to share $k \geq 1$ segments of the query string can help us achieve more effective pruning than simply restricting k to be 1. Lemma 4.1 can now be rewritten to reflect the generalized count filtering as follows.

LEMMA 6.1. *Given a string r with $\tau + k$ segments and a string s , if s is similar to r within threshold τ , s must contain substrings that match at-least k segments of r for $k \geq 1$.*

Before we describe the partitioning that incorporates the count filtering as summarized in the lemma above we highlight the necessary changes required to obtain candidate strings based on count filtering. We use a simple ‘‘Scan-Count’’ algorithm proposed in [11], so as to select only those strings for verification that satisfy the count filtering. We maintain an array of counts for all the string ids in \mathcal{S} . For each segment $r(i)$ we first obtain the candidate strings resulting due to alignment of $r(i)$ and increment the count corresponding to each of the candidate string by 1. Then the string ids that appear as a candidate due to at least k segments can be reported.

Here restriction that each segment $r(i)$ can contribute only once towards a string s in count array poses overhead since string id can appear in the suffix range $[l_i \dots r_i]$ multiple times and more than one alignment can satisfy the position restricted alignment condition as well. Though such a restriction can be easily handled theoretically by using chaining idea in [14], it has the potential to offset any advantage obtained by splitting the query strings into $k > 1$ partitions. Therefore, we decide to enforce the uniqueness restriction in reporting candidate strings per segment $r(i)$ selectively. Otherwise, we let the candidate string to be reported multiple times per segment $r(i)$, thus resulting in inflated count value and possibly qualifying a string as a candidate incorrectly. We note that a string that incorrectly qualifies as a candidate will be pruned during final verification and will not be incorrectly reported as an answer.

Dilemma of choosing k : On one hand, by increasing the length of segments, we can hope to make the segment distinct enough so that it does not appear multiple times in the same string. This reduces the number of strings incorrectly reported as a candidate thus saving the expensive verification operation. On the other hand by decreasing k , we will have a lower threshold on the number of segments shared by similar strings, causing a less selective count filter to eliminate dissimilar string pairs. We use a heuristic technique that initializes $k = 1$. We then increment k till following condition is satisfied: $\lfloor |r|/(\tau + k) \rfloor = \lfloor |r|/(\tau + k + 1) \rfloor$. This simple technique tries to maximize the selected value of k while ensuring minimum length of the segments of r remain unchanged and thus offers a useful middle ground for selection of k .

Dynamic partitioning: Encouraged by the research efforts in variable length partitioning so far [12], we form a dictionary of strings based on which an informed decision can be made for query string partitioning. We construct this dictionary by navigating GST in depth first search manner. Inline with the existing approaches, we assume the availability of two length bounds q_{min} and q_{max} to limit the dictionary size as well its construction time. We assign a weight to

each node u in GST given by $dist(u)/(r - l + 1)$, where $[l \dots r]$ represents the suffix range of node u and $dist(u)$ represents the number of distinct strings that have $path(u)$ as one of its suffix. In another words, $dist(u)$ is the number of distinct string ids in the subarray $s_ids[l \dots r]$. Intuitively, the weight of the node tries to estimate the overhead involved in reporting only the unique candidate strings if u is the locus node of a segment of string r . Along with two length bounds we assume a user defined threshold $0 \leq UQ_{min} \leq 1$ is given. Then string represented by $path(u)$ is added to the dictionary while navigating the GST if following two conditions are satisfied.

- $dist(u)/(r - l + 1) < UQ_{min}$
- $q_{min} \leq |path(u)| \leq q_{max}$
- string corresponding to node v i.e. $path(v)$ does not exists in the dictionary such that v is a proper ancestor of node u

Based on such a dictionary we now follow the procedure described below to obtain the proposed dynamic partitioning of the query string r . It is a greedy algorithm that initializes the segment with the minimum length of $\lfloor |r|/(\tau + k) \rfloor$ and keeps incrementing it by one character at a time till it does not belong to dictionary or it is not possible to extend this segment any further without reducing the length of yet to produce segments below desired minimum length i.e. $\lfloor |r|/(\tau + k) \rfloor$.

Input: Dictionary \mathcal{D} , string r , segment count $(\tau + k)$
Output: Set R of partitioned segments of r

$R = \text{empty set}$
 $i = 1, pos = 1$
 $len = \lfloor |r|/(\tau + k) \rfloor, rem = |r| \bmod (\tau + k)$
while ($pos \leq |r| - len + 1$) **do**
 $r(i) = r[pos \dots pos + len]$
 $j = 0$
 while ($rem > 0$ AND $r(i) \in \mathcal{D}$) **do**
 $j = j + 1$
 $rem = rem - 1$
 $r(i) = r[pos \dots pos + len + j]$
 end while
 $i = i + 1$
 $pos = pos + len + j$
end while

Other than directing the query string partitioning, the dictionary also allows us to selectively enforce the uniqueness restriction in reporting candidate strings for segment $r(i)$. Each segment $r(i)$ in the final partitioned set R that also belongs to the dictionary, implies that the suffix range of $r(i)$ contain multiple occurrences of same string ids and hence overhead of applying the uniqueness restriction can pay off by reducing the number of incorrectly reported candidate strings and thereby avoiding their verification cost.

We note that the choice of user defined parameter UQ_{min} greatly affects the quality of partitioning as well as play an important role in balancing the overhead of applying uniqueness restriction with the verification cost of incorrectly reported candidate strings. We leave the strategy of selecting good value for UQ_{min} as a future work and decide its value empirically for the work in this paper.

6.2 Filtering based on frequency distance

The frequency distance based filtering was first introduced by Kahveci and Singh [9]. The intuition behind this filtering is that if two strings are similar, then the frequency of the alphabet symbols in two strings should also be similar. For the formal application, we first define the frequency vector. Given a string s from the alphabet Σ , frequency vector $f(s)$ is defined as $f(s) = [c_1, \dots, c_{|\Sigma|}]$, where c_i is the count of i th alphabet of Σ . Below, we first define the edit distance and then Theorem 6.3 captures the relation between frequency distance and edit distance as established in [9].

DEFINITION 6.2. Let r and s be two strings from the same alphabet Σ . Let $f(r)$ and $f(s)$ be the frequency vectors of r and s respectively. The frequency distance of r and s to be

$$fd(r, s) = \max\{posDistance, negDistance\},$$

where

$$posDistance = \sum_{f(r)_i > f(s)_i} f(r)_i - f(s)_i$$

and

$$negDistance = \sum_{f(r)_i < f(s)_i} f(r)_i - f(s)_i$$

THEOREM 6.3. Let r and s be 2 strings from the same alphabet Σ . Then we have

$$fd(r, s) \leq ed(r, s).$$

Frequency distance based filter can be particularly useful for long strings with small alphabets e.g. DNA strings. In addition to the various index components described earlier we also maintain the frequency vector for each string in the collection \mathcal{S} . Such a storage does not result in too much of space overhead with restricted alphabet size and relatively long strings. Also applying this filter can be much faster than the verification process even with the optimizations.

6.3 Improving the verification

The classic dynamic programming algorithm of edit distance of strings r, s takes $O(|r| \times |s|)$ times and space. This algorithm computes a matrix M , whose $[i, j]$ th entry records the edit distance between substrings $r[0..i]$ and $s[0..j]$. As we only need to determine whether $ed(r, s) \leq \tau$, computing the entire matrix M is not necessary. We use verification algorithm by Ukkonen with time complexity of $O((\tau + 1) \times \min(|r|, |s|))$. It relies on the following theorem.

THEOREM 6.4. In order to check inequality $ed(r, s) \leq \tau$, it is enough to compute the entries on diagonal of the matrix satisfying $-\Delta \leq j - i \leq |r| - |s| + \Delta$ if $|r| \leq |s|$ or $|r| - |s| - \Delta \leq j - i \leq \Delta$ if $|s| > |r|$, where $\Delta = (\tau - ||r| - |s||)/2$.

A straightforward early-termination method is to check if all elements in one row are larger than τ . Then by dynamic programming algorithm all the values in the rows yet to be computed must be larger τ . This simple technique do not add much computational overhead and was found to be effective during experimentation.

7. EXPERIMENTAL RESULTS

We have implemented our method and conducted an extensive set of experimental studies on the Human genome data set provided by ‘‘String Similarity Search/Join Competition 2013’’. We use a set of 5000 queries (provided along with the data set) and perform experiments with edit distance threshold values in the set $\{4, 8, 12, 16\}$. All the algorithms were implemented in C++ and compiled using GCC 4.4 with -O3 flag. Public code libraries at <http://www.uni-ulm.de/in/theo/research/sdsl.html> and <http://pizzachili.dcc.uchile.cl/indexes.html> are used to develop some of the components in the proposed index. All the experiments were run on a Ubuntu machine with an Intel core i5 (quad core) 1.6GHz processor and 8GB RAM. We consider four variants of the the proposed indexing scheme as follows:

- *I-GST*: This is the index as described in Section 4 and makes use length and position filtering independently.
- *I-PRA*: This is the index as described in Section 5. It uses aggressive position restricted alignment for filtering out candidate strings and also make use the wavelet tree based index storage so that cost of applying filters at the query time can be avoided.
- *I-CF⁻*: This is the index as described in Section 5 and utilizes the count filtering introduced in Section 6.1. It also employs the dictionary to dynamically partition the query string. However, it applies uniqueness constrain while reporting the candidate strings of each segment $r(i)$ of query string. We set the parameter $UQ_{min} = 0.85$ for this index.
- *I-CF⁺*: This index is exactly same as the previous one i.e. *I-CF⁻* with one modification. *I-CF⁺* selectively applies the uniqueness constrain while reporting the candidate strings of a segment based on the dictionary.
- *I-FDF*: This is the index that further improves *I-CF⁺* index by incorporating the frequency distance filtering.

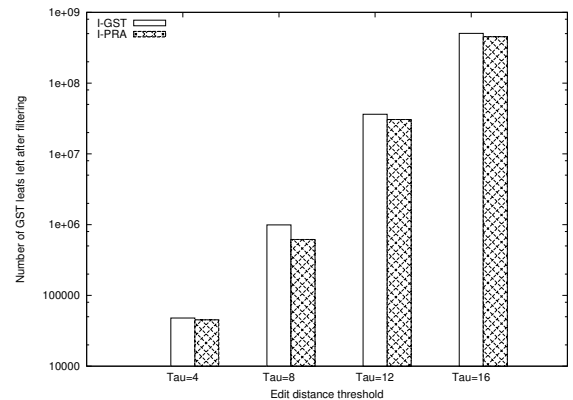


Figure 1: Effect of position restricted alignment

7.1 Effect of position restricted alignment

As a part of this experiment, we evaluate the effectiveness of ‘‘position restricted alignment’’ against the length

and position filter applied independently. For this purpose we compare index *I-GST* with variant *I-PRA* and use the number of GST leaves remained after applying the filtering conditions by each of them as measure of performance. As shown in the Figure 1, position restricted alignment is able to filter out up to 40% of the leaves that could not be filtered out using either length or position filtering. We also highlight that *I-PRA* do not have to apply the filter during execution and hence also improves the query time.

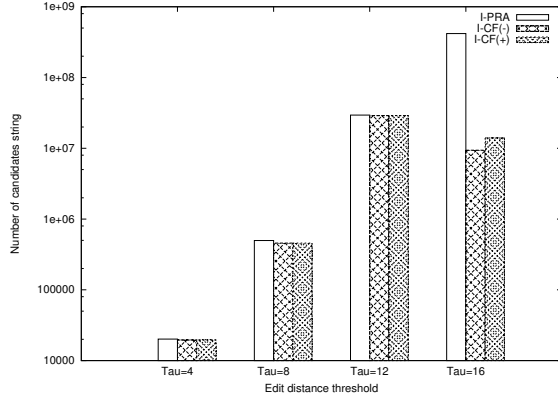


Figure 2: Effect of count filtering: Number of candidate strings

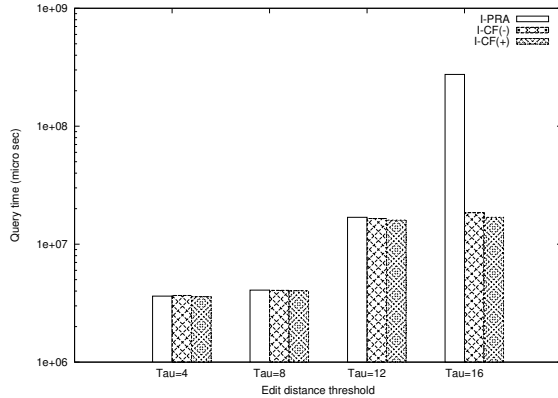


Figure 3: Effect of count filtering: Total query time

7.2 Effect of count filtering

In this experiment we evaluate the effectiveness of applying count filtering by partitioning the query string into $\tau + k$ segments for $k > 1$. As noted in the Section 6.1, applying count filtering requires candidate strings due to each partition of query r to be reported uniquely. To isolate the effect of overhead due to the requirement of unique reporting, we compare three variants *I-PRA*, *I-CF⁻* and *I-CF⁺*. We use number of candidate strings subjected to verification as well as the total query time for comparing these three index variants. Figure 2 and 3 show the benefits offered by count filtering along with the dynamic partitioning technique as *I-PRA* needs to verify more strings and hence higher query time as compared to other two index variants. Comparison of *I-CF⁻* and *I-CF⁺* reveals that in general applying

uniqueness constrain requires lot of overhead as a result savings achieved by reducing the number of candidate strings to be verified might not be translated in query time improvement. Therefore we need to be judicious while making a decision to report only unique candidate strings for a particular segment partition of query r .

7.3 Effect of frequency distance filtering

This experiment is intended to weigh the benefits of applying frequency distance filtering against the overhead of applying the same. We compare index variant *I-CF⁺* with *I-FDF* as they differ only in one aspect; former does not employ the frequency distance filtering whereas the later does. Figure 4 reveals that though such filtering can effectively reduce the number of candidate strings, the overhead involved makes it an attractive option only for large values of edit threshold τ . We observed that, for small values of τ , most of the candidates are filtered out by position restricted alignment and count filtering beforehand. As a result overhead of applying frequency distance filtering does not help to improve the query performance in such scenario.

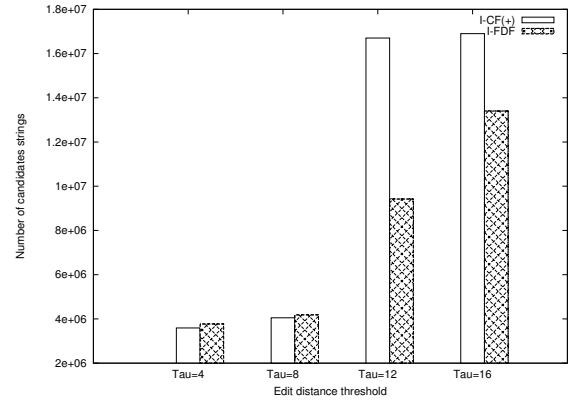


Figure 4: Effect of frequency distance filtering

8. REFERENCES

- [1] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.
- [2] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD Conference*, pages 313–324, 2003.
- [3] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.
- [4] R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don’t cares. In *STOC*, pages 91–100, 2004.
- [5] D. Deng, G. Li, and J. Feng. Top-k string similarity search with edit-distance constraints. In *ICDE*, 2013.
- [6] J. Feng, J. Wang, and G. Li. Trie-join: a trie-based method for efficient string similarity joins. *VLDB J.*, 21(4):437–461, 2012.
- [7] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, 2001.

- [8] R. Grossi, A. Gupta, and J. S. Vitter. High-Order Entropy-Compressed Text Indexes. In *Proceedings of Symposium on Discrete Algorithms*, pages 841–850, 2003.
- [9] T. Kahveci and A. K. Singh. Efficient index structures for string databases. In *VLDB*, pages 351–360, 2001.
- [10] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD Conference*, pages 802–803, 2006.
- [11] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.
- [12] C. Li, B. Wang, and X. Yang. Vgram: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, pages 303–314, 2007.
- [13] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.
- [14] S. Muthukrishnan. Efficient Algorithms for Document Retrieval Problems. In *Proceedings of Symposium on Discrete Algorithms*, pages 657–666, 2002.
- [15] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1), 2001.
- [16] E. Ohlebusch, J. Fischer, and S. Gog. Cst++. In *SPIRE*, pages 322–333, 2010.
- [17] R. Raman, V. Raman, and S. S. Rao. Succinct Indexable Dictionaries with Applications to Encoding k -ary Trees and Multisets. In *Proceedings of Symposium on Discrete Algorithms*, pages 233–242, 2002.
- [18] K. Ramasamy, J. M. Patel, J. F. Naughton, and R. Kaushik. Set containment joins: The good, the bad and the ugly. In *VLDB*, pages 351–362, 2000.
- [19] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, pages 743–754, 2004.
- [20] P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of Symposium on Switching and Automata Theory*, pages 1–11, 1973.
- [21] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.
- [22] Z. Yang, J. Yu, and M. Kitsuregawa. Fast algorithms for top-k approximate string matching. In *AAAI*, 2010.
- [23] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *SIGMOD Conference*, pages 915–926, 2010.