2013

Approximate sequence alignment

Xuanting Cai
Louisiana State University and Agricultural and Mechanical College, xcai1@math.lsu.edu

Approximate Sequence Alignment

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science

in

The Department of Computer Science

by
Xuanting Cai
M.S, Louisiana State University, 2008
B.S., Peking University, 2007
August 2013

Acknowledgements

This thesis would not be possible without several contributions. First of all, I am extremely grateful to my advisor, Professor Shah, for his constant encouragement and guidance throughout this work.

Meanwhile, I really wish to express my sincere thanks to all the committee members, Prof. Foltz, Prof. Park, for their help and suggestions on the corrections of this dissertation, and to Manish and Sharma for helpful discussion during the research.

I also take this opportunity to thank the department of Computer Science of Louisiana State University for providing me with a pleasant working environment and all the necessary facilities.

This dissertation is dedicated to my parents and my wife for their unceasing support and encouragement.

Table of Contents

Abstract

Given a collection of strings and a query string, the goal of the approximate string matching is to efficiently find the strings in the collection, which are similar to the query string. In this paper, we focus on edit distance as a measure to quantify the similarity between two strings. Existing q-gram based methods use inverted lists to index the q-grams of the given string collection. These methods begin with generating the q-grams of the query string, disjoint or overlapping, and then merge the inverted lists of these q-grams. Several filtering techniques have been proposed to segment inverted lists in order to obtain relatively shorter lists, thus reducing the merging cost. The filtering technique we propose in this thesis, which is called position restricted alignment, 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. It enables us to auto-filter the inverted lists. We evaluate the effectiveness of the proposed approach by experiments.

Chapter 1
Introduction

1.1 Motivation

String matching plays an important role in computer science and related area. It is the key step in many applications. For example, Bioinformatics, search engine and shopping website.

Bioinformatics is a cross field that retrieves information from biological data, for example, DNA sequences. DNA sequences could be simply modelled as sequence of specific alphabets A, C, G, T. These sequences encode the genetic information of all known living organisms. Searching a particular segment among other sequences is a fundamental operation in Bioinformatics. Because of mutation and evolution, the same genetic sequence appears slightly different in two organisms of the same species, even two individuals in the same species. Therefore, approximate string matching is the operation used in Bioinformatics. Moreover, there are more than 100 gigabases of DNA and RNA sequences at the end of 2005 thanks to the new generation sequencing technology. Thus, a fast approximate pattern matching is critical in the research.

Another application of the approximate pattern matching is from business, such as searching engine and shopping sites. The back end servers of searching engine and shopping sites have numerous information stored. Users will query the site for knowledge or products. In reality, users may misspell words or only have parts of the words. In those cases, exact pattern matching will not give meaningful results. For example, users searching for "Massachusetts" may just enter in a word like

"Masachusets". In order to improve user experience, the system need return the result quickly. So a fast approximate string matching method is necessary.

1.2 Problem Definition

In this section, we give the formal definition of approximate string matching problem. 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 thesis, we use edit distance or Levenshtein distance to quantify the similarity between two strings.

Definition 1.2.1. Given two strings $A = a_1 \cdots a_m$ and $B = b_1 \cdots b_n$, the edit distance or Levenshtein distance is the minimal number of following operations needed to transform string A to string B:

1. deleting a symbol from position i to get $a_1 \cdots a_{i-1} a_{i+1} \cdots a_m$;
2. inserting a symbol b at position i to get $a_1 \cdots a_i b a_{i+1} \cdots a_m$;
3. changing a symbol at position i to symbol b to get $a_1 \cdots a_{i-1} b a_{i+1} \cdots a_m$.

We denote the edit distance between strings r, s by $ed(r, s)$.

In this thesis, 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 1.2.2. Given a non-negative integer τ, a string r and a collection of string $\mathcal{S}$, an approximate string query finds all pairs (r, s) with $s \in \mathcal{S}$ such that $ed(r, s) \leq \tau$.

For example, consider the strings in Table 1.1. Suppose threshold $\tau = 2$. Then strings s_1 and s_2 are similar to query string $r =$ "$AACTGTGC$" as their edit distance is not larger than 2.

TABLE 1.1. A collection $\mathcal{S}$ of strings

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

1.3 Common Approach

Existing methods to address this problem can be broadly classified into two categories based on the way of indexing string database. The methods in first category use the suffix tree data structure to index string database. These methods rely on the fact that edit distance between two strings is bounded below by the edit distance between their prefixes. This allows us to filter out strings, which have a prefix with edit distance larger than the required threshold with respect to prefixes of the query string. However, these methods are usually inefficient for long strings as they have a small number of shared prefixes. Moreover navigation cost of suffix tree suffers from exponential dependence on the pattern length as well as the edit distance threshold in worst case. The methods in second category are q-gram based and make use of inverted lists 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 reduc-

ing 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 [13].

- Applying filters to filter out the candidate strings during query execution can be expensive in terms of computational cost.

1.4 Related Work

There are many studies on approximate string matching. Several algorithms [3, 13, 12] 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 [13] 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 [20, 12]. We use a simple "ScanCount" method [12] 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, lots 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. [10] 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. [23] 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 [23]. Recent studies on this problem also includes B^+-tree based approach by Zhang et al. [24] and trie based approach by Deng at al. [6]. 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" [19, 8, 4, 2, 11, 22, 14, 7]. 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 [16, 5] 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.

1.5 Thesis Organization

In Chapter 2, we present the fundamental data structures we will use in our algorithm, i.e. suffix tree and wavelet tree. In Chapter 3, we summarize a general framework to address approximate string matching problem and introduce the new technique we invented to eliminate non-candidate strings in a fast way. In Chapter 4, we talk about a few technique in the implementation to filter out more strings and speed up matching process. In Chapter 5, we run some experiments on Sea Star genomes and compare our method to existing methods.

Chapter 2
Suffix Tree And Wavelet Tree

2.1 Suffix Trees

The suffix tree is one of the most important data structures in string processing applications. It is a tree of all suffixes of a string. So first we introduce the definition of the suffix.

Definition 2.1.1. Given a string $s = s_1 \cdots s_n$, the ith suffix of s is the substring $s_i \cdots s_n$.

In order to construct the suffix tree of a string, we require each string terminated with a special character. Let Σ be the alphabet of strings. Let $\$ \notin \Sigma$ be a unique character, which terminates each string. We define the suffix tree following the definition in [1].

Definition 2.1.2. Given a string $s = s_1 \cdots s_{n+1}$ with $s_{n+1} = \$$, the suffix tree of s, denoted by $ST(s)$, is a compacted trie of all suffixes of string s satisfying:

1. $ST(s)$ has n leaves labeled $1, \cdots, n$.

2. Each internal node has at least 2 children.

3. Each edge in the tree is labeled with a substring of s.

4. The concatenation of edge labels from the root to the leaf labeled i is the suffix starting from s_i.

5. The labels of the edges connecting a node with its children start with different characters.

6. The tree is lexicographically arranged such that the ith leftmost leaf corresponding to the ith suffix in lexicographic order.

To illustrate the definition, we show an example of the suffix tree.

Example 2.1.3. Let $s = BANANA\$$. Then s has suffixes:

0. $BANANA\$$
1. $ANANA\$$
2. $NANA\$$
3. $ANA\$$
4. $NA\$$
5. $A\$$

They form the suffix tree of s in Figure 2.1.

It is easy to see that except the edge labels, the size of the suffix tree is $O(n)$. In order to obtain linear space implementation of suffix tree, we represent the edge label by two integers. The first integer denotes the starting position of the substring for that edge and the second integer denotes the ending position of the substring for that edge.

We also can construct suffix tree for a collection of strings. We call it the generalized suffix tree.

Definition 2.1.4. Given a collection of strings $\mathcal{S} = d_1, \cdots, d_m$, the generalized suffix tree of $\mathcal{S}$, denoted by $GST(\mathcal{S})$, is the suffix tree of string $d_1 \cdots d_m$, where $d_1 \cdots d_m$ is the concatenation of d_i's.

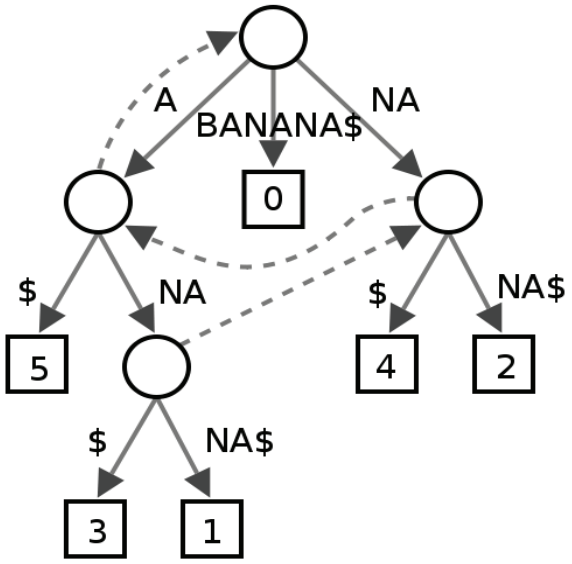

FIGURE 2.1. Suffix tree for the string BANANA. Each substring is terminated with special character \$. The six paths from the root to a leaf (shown as boxes) correspond to the six suffixes $BANANA\$$, $ANANA\$$, $NANA\$$, $ANA\$$, $NA\$$ and $A\$$. The numbers in the leaves give the start position of the corresponding suffix. Suffix links, drawn dashed, are used during construction.

2.2 Wavelet Trees

In our algorithm, rank query is required to find the position of match in suffix array. The wavelet tree is a standard choice of rank query problem.

Definition 2.2.1. Given a string s drawn from alphabet Σ, the wavelet tree of s, denoted by $WT(s)$, is an ordered balance binary tree on Σ satisfying

1. Each leaf is labeled with a symbol in Σ, and the leaves are sorted alphabetically from left to right.

2. 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.

3. Each node partitions its alphabet set among the two children (almost) equally, such that all symbols represented by the left child are lexicographically smaller than those represented by the right child.

4. 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.

Here is an example of wavelet tree.

Example 2.2.2. Let $s = AGTAGACTAGTGGATTACCATCACT$ and $\Sigma = \{A, C, G, T\}$. The wavelet tree of s is drawn in Figure 2.2.

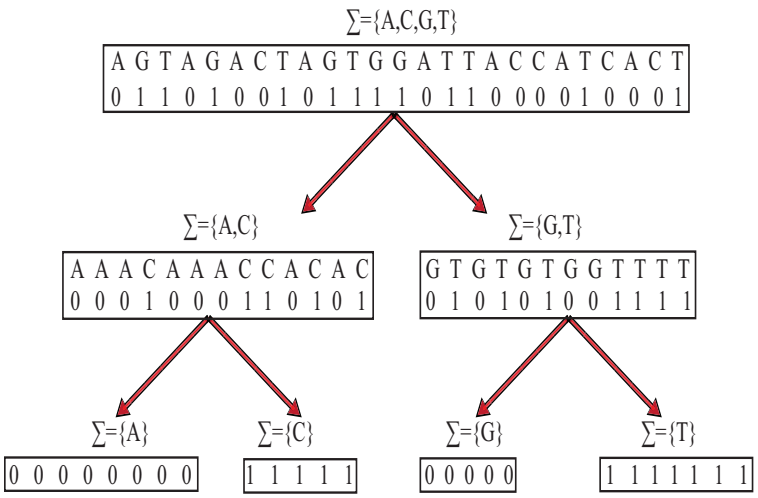

FIGURE 2.2. The wavelet tree of s on alphabets $\Sigma = \{A, C, G, T\}$.

The following is a useful lemma on wavelet trees.

Lemma 2.2.3. *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 $O((1 + output) \log |\Sigma|)$ time.*

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

Lemma 2.2.4. *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 $O((1+output)(\log \alpha + \log \beta))$ time, where l, r, π and π' are input parameters.*

Chapter 3

A General Framework And A New Technique

3.1 A General Framework

This section gives an 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, where τ is the threshold edit distance. Here for simplicity we assume $|r| \geq \tau + 1$. The idea behind such a partitioning is: if a string s has no substring that matches any segment of r, then s cannot be similar to r. Lemma 3.1.1 formally summarizes this idea.

Lemma 3.1.1. *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.*

This partitioning technique divides the query string into $\tau + 1$ segments each with length $\lfloor |r|/(\tau+1) \rfloor$ except the last $|r|$ mod $(\tau+1)$ segments, which have length $\lceil |r|/(\tau+1) \rceil$. For example, consider a string $r = AACTGTGC$ 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 $\{$"AA'', "CTG'', "$TGC''\}$. 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 ith segment of r after partitioning. We refine this partitioning technique later in Section 4.1.

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* with each of length n as follows.

Definition 3.1.2. Suppose the ith leftmost leaf of GST belongs to string $x \in \mathcal{S}$. Then we set $s\_ids[i]$ to be x and set $s\_ids[i]$ to be the starting position of this suffix in x.

These two arrays essentially stores the information about the leaves of GST. They help to eliminate the need to compute the same information during query answering. A straight forward approach to find candidate strings those 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 $s\_ids[l_i...r_i]$, where $[l_i...r_i]$ represents the suffix range of ith 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 to τ with respect to string r. Without loss of generality let s contains a substring $s(i)$ that matches segment $r(i)$. By Lemma 3.1.1, 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 [14], 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 $[l_i...r_i]$ for segment $r(i)$, we ignore the string $s\_ids[j]$ if its length is not in the range $[max(0, |r| - \tau])...|r| + \tau]$ 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 $[max(0, r(i)^{sp} - \tau)...r(i)^{sp} + \tau]$ to apply position filtering. Here we note that a particular string s may appear multiple times in the suffix range $[l_i...r_i]$, that is, 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 1.1 that have substring matching with at least one of the segments of query string $r =$ "$AACTGTGC$", 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 [14], we use verification algorithm by Ukkonen along with simple early termination criteria. We defer more details about optimizing verification step to Section 4.3.

3.2 A New Technique

In this section, we describe the new technique, 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 $s(j)$ as their substrings respectively such that $r(i) = s(j)$. Further $r(i)^{sp}$ and $s(j)^{sp}$ represents the starting positions of $r(i)$ and $s(j)$ within r and s respectively. Now we partition the string s into $\overleftarrow{s}(j), s(j), \overrightarrow{s}(j)$, where $\overleftarrow{s}(j)$ and $\overrightarrow{s}(j)$ are the parts of s respectively on the left and right side of the segment $s(j)$. 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:

Lemma 3.2.1.

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

if and only if

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

We note that $ed(\overleftarrow{r}(i), \overleftarrow{s}(j))$ captures the essence of position filtering whereas $ed(\overrightarrow{r}(i), \overrightarrow{s}(j))$ 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 length as well position filter. We note that position restricted alignment is a tighter 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.

Theorem 3.2.2. *If*

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

then we have

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

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

By solving the above inequality, we can obtain $O(\tau^2)$ solutions in the form of $(s(j)^{sp}, |s|)$ pair. Let C be the set of all such possible pairs. Therefore, our task is now reduced to find the strings such that pair $(s\_pos[j], |s\_ids[j]|) \in C$ and

$j \in [l_i ... r_i]$, where $[l_i ... r_i]$ is the suffix range of partition segment $r(i)$. To answer such a query efficiently we use the data structure described in Lemma 2.2.4. For us to be able to use the data structure, we simply map the $j$the leftmost leaf of GST to a 3 dimensional point $(j, s\_pos[j], |s\_ids[j]|)$.

Though the idea behind position restricted alignment is similar to the one proposed in [14], there are primarily two distinctions with respect to our work: (1) In [14], 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 [14] 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.

Chapter 4
Implementation

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.

4.1 Incorporating count filtering

Instead of partitioning the string r into $\tau+1$ segments, we can 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 3.1.1 can now be rewritten to reflect the generalized count filtering as follows.

Lemma 4.1.1. *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 "ScanCount" algorithm proposed in [12], 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 $[l_i...r_i]$ 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 [15], 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 [13], 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 $[l...r]$ 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 $s\_ids[l...r]$. 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 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 𝒟, string r, segment count (τ + k)
Output: Set R of partitioned segments of r
R = empty set
i = 1, pos = 1
len = ⌊|r|/(τ + k)⌋, rem = |r| mod (τ + k)
while ( dopos ≤ |r| − len + 1)
    r(i) = r[pos...pos + len]
    j = 0
    while ( dorem > 0 AND r(i) ∈ 𝒟)
        j = j + 1
        rem = rem − 1
        r(i) = r[pos...pos + len + j]
    end while
    i = i + 1
    pos = pos + len + j
end while
```

FIGURE 4.1. Psudocode.

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.

4.2 Filtering based on frequency distance

The frequency distance based filtering was first introduced by Kahveci and Singh [10]. The intuition behind this filtering is that if two strings are similar, then the fre-

quency 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 $f(s) = [c_1, ..., c_{|\Sigma|}]$, where c_i is the count of ith alphabet of Σ. Below, we first define the edit distance and then Theorem 4.2.2 captures the relation between frequency distance and edit distance as established in [10].

Definition 4.2.1. 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

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

Theorem 4.2.2. *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.

4.3 Improving the verification

The classic dynamic programing 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 4.3.1. *In order to check inequality $ed(r, s) \leq \tau$, it is enough to computer 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 computed must be larger τ. This simple technique do not add much computational overhead and was found to be effective during experimentation.

Chapter 5
Experimental Analyses

5.1 Experimental Setup

We have implemented our method and conducted an extensive set of experimental studies on the sea star transcriptomic sequences provided by David Foltz Laboratory at `http://www.foltzlab.biology.lsu.edu`.

- We use a set of 100,000 DNA sequences with length $100 - 200$, a set of 5,000,000 DNA sequences with length $100 - 200$.

- We choose edit distance threshold values in $\{0, 4, 8, 12, 16\}$.

5.2 Platform

All the algorithms were implemented in C++ and compiled using GCC 4.7 with -O3 flag. All the experiments were run on a Ubuntu machine with an Intel core i5 quad core 1.6GHz processor with $256K$ L2 cache and $6144K$ L3 cache and 8GB RAM.

5.3 Source Code

The components in index, i.e. suffix tree is available in public. We obtain these libraries from

- Compressed Suffix Tree: `http://www.uni-ulm.de/in/theo/research/sdsl.html`, version 0.9.8.

5.4 Experimentation

We consider four variants of the the proposed indexing scheme as follows:

- *I-GST*: This is the index as described in Chapter 3, Section 3.1 and makes use length and position filtering independently.

- *I-PRA*: This is the index as described in Chapter 3, Section 3.2. 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-CFA*: This is the index as described in Chapter 3, Section 3.2 and utilizes the count filtering introduced in Section 4.1. While we use count filtering technique, we need a data structure to record how many grams of data string appears in the query string. We use an array of size of number of data strings for *I-PRA*. It also employes the dictionary to dynamically partition the query string.

- *I-CFH*: This is the index as described in Chapter 3, Section 3.2 and utilizes the count filtering introduced in Section 4.1. However, we use a hash table to record the counting. It also employes the dictionary to dynamically partition the query string.

- *I-FDF*: This is the index that further improves *I-CFH* index by incorporating the frequency distance filtering.

5.5 Effect of position restricted alignment

In this experiment, we evaluate the effectiveness of position restricted alignment against the length and position filter applied independently. We compare index *I-GST* with variant *I-PRA* and use the number of GST leafs remained after applying the filtering conditions by each of them as measure of performance. As shown in the Figure 5.1, position restricted alignment is able to filter out up to 44% of the

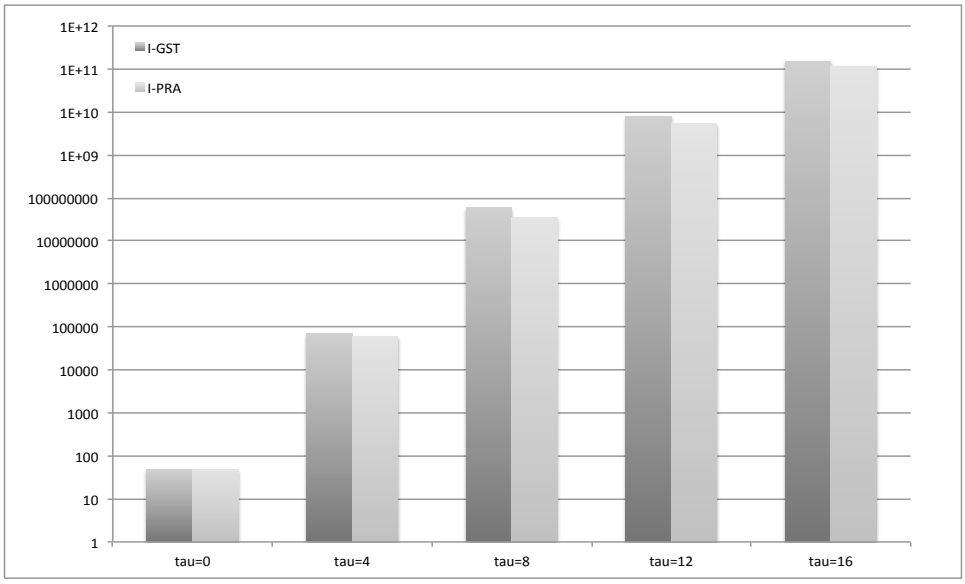

FIGURE 5.1. Effect of position restricted alignment. The black columns are the number of leafs left after applying position filter and length filter separately. The grey columns are the number of leafs left after applying position restricted filter.

leafs that could not be filtered out using either length or position filtering. We also highlight that *I-PRA* do not have to apply the the filter during execution and hence also improves the query time.

5.6 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 $k > 1$. We compare the number of strings left after applying the count filtering with the number of strings left without applying the count filtering. As shown in Figure 5.2, count filtering along with the dynamic partitioning technique can reduce the string number up to 74%. However, the average query time after applying the count filtering could not be reduced up to the same scale as shown in Figure 5.3. This is because the time is increased by the overhead of applying the counting filtering.

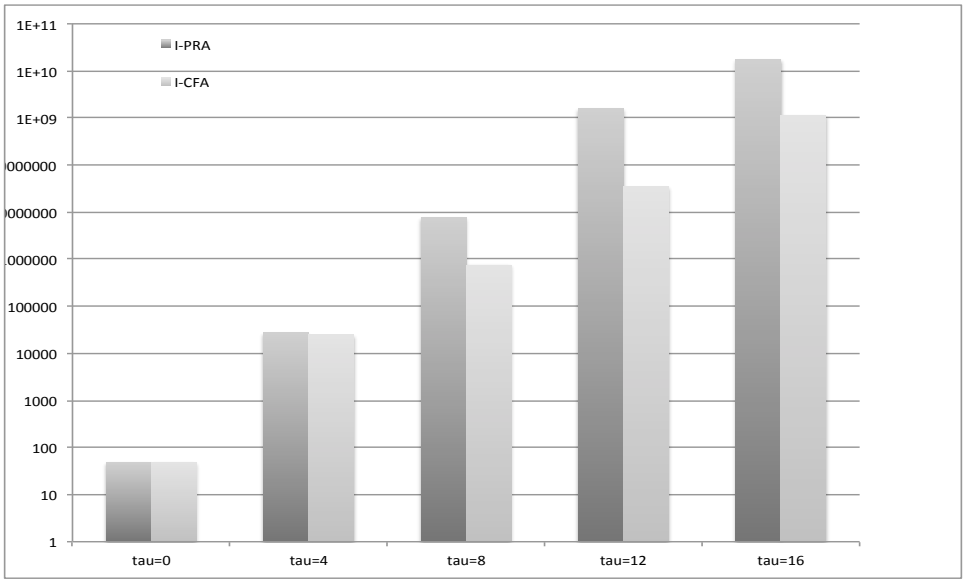

FIGURE 5.2. Effect of count filter in number of strings left. The blue columns are the numbers of strings left without count filtering. The red columns are the numbers of strings left with count filtering.

5.7 Effect of using hash table

In this experiment, we evaluate the effectiveness of using hash table to apply the counter filtering instead of using an array with size of number of data strings. We compare the average query time by using hash table with the average query time by using an array. As shown in Figure 5.4, when τ is small, we need count a few strings. In this case, using a hash table could save a lot of time. When τ is large, we need count a lot strings, which is approximately the same as using an array. However the overhead of calling function of hash table is large, which reduces the benefit of using hash table is reduced.

5.8 Effect of frequency distance filtering

This experiment is intended to evaluate the effectiveness of applying frequency distance filtering. We compare index variant $I\text{-}CFH$ with $I\text{-}FDF$ as they differ only in one aspect. The former does not employ the frequency distance filtering

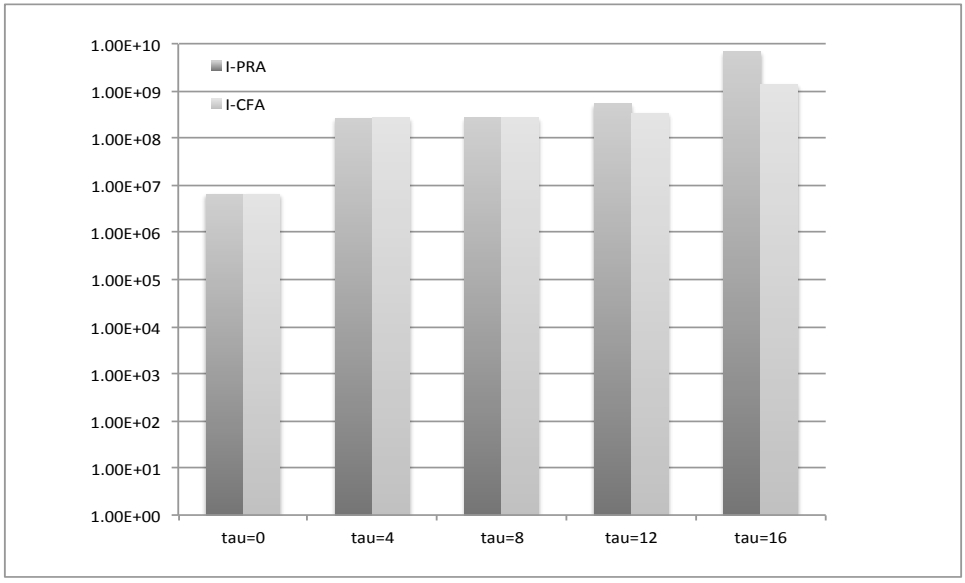

FIGURE 5.3. Effect of count filter in query time. The blue columns are the average query time without count filtering. The red columns are the average query time with count filtering.

whereas the later does. Figure 5.5 reveals that such filtering can effectively reduce the number of candidate strings.

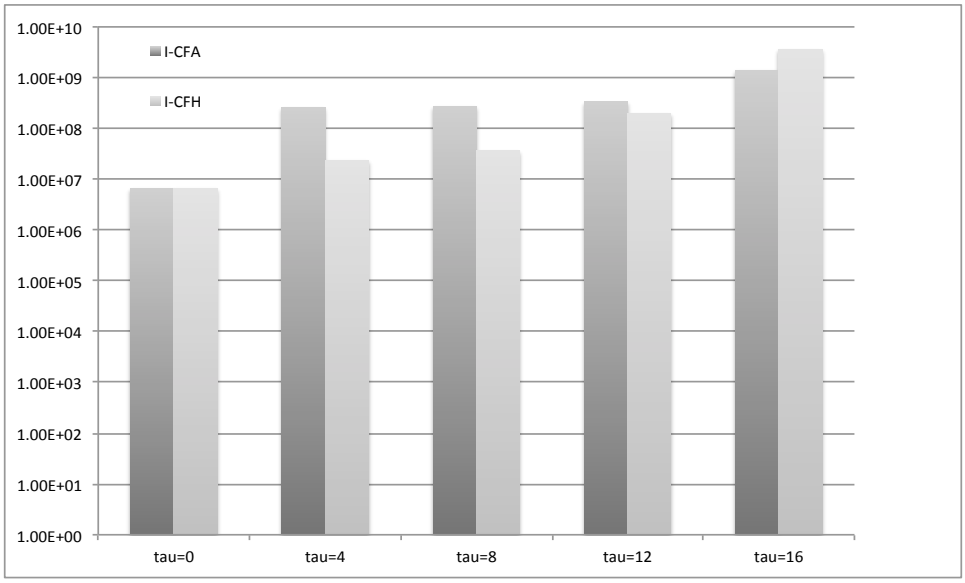

FIGURE 5.4. Effect of using hash table to count. The blue columns are the average query time of using an array. The red columns are the average query time of using hash table.

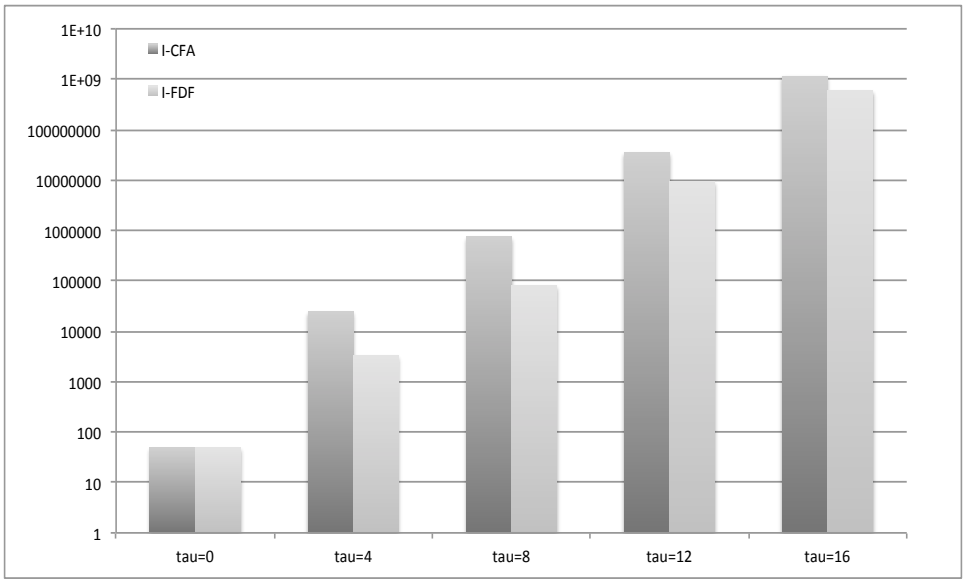

FIGURE 5.5. Effect of frequency distance filtering. The blue columns are the numbers of strings left before applying frequency filtering. The red columns are the numbers of strings left after applying frequency filtering.

References

[1] Srinivas Aluru. Suffix Trees And Suffix Arrays. *Note*, 2001.

[2] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[3] 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.

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

[5] R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *STOC*, pages 91–100, 2004.

[6] D. Deng, G. Li, and J. Feng. Top-k string similarity search with edit-distance constraints. In *ICDE*, 2013.

[7] 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.

[8] 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.

[9] 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.

[10] T. Kahveci and A. K. Singh. Efficient index structures for string databases. In *VLDB*, pages 351–360, 2001.

[11] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD Conference*, pages 802–803, 2006.

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

[13] 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.

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

[15] S. Muthukrishnan. Efficient Algorithms for Document Retrieval Problems. In *Proceedings of Symposium on Discrete Algorithms*, pages 657–666, 2002.

[16] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1), 2001.

[17] E. Ohlebusch, J. Fischer, and S. Gog. Cst++. In *SPIRE*, pages 322–333, 2010.

[18] 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.

[19] 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.

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

[21] P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[22] 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.

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

[24] 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.

Vita

Xuanting Cai was born in 1984, in Zhuji City, Zhejiang Province, China. He finished his undergraduate studies at Peking University of China in July 2007. In August 2007, he came to Louisiana State University to pursue Ph.D degree in Mathematics. He is currently a candidate for the degree of master in System Science, which will be awarded in August 2013.