Approximate Search Engine Optimization for Directory Service

Kai-Hsiang Yang and Chi-Chien Pan and Tzao-Lin Lee

Department of Computer Science and Information Engineering,
National Taiwan University, Taipei, Taiwan, R.O.C.
E-mail: { f6526004, d5526001, tl_lee}@csie.ntu.edu.tw

Abstract

Today, in many practical E-Commerce systems, the real stored data usually are short strings, such as names, addresses, or other information. Searching data within these short strings is not the same as searching within longer strings. General search engines try their best to scan all long strings (or articles) quickly, and find out the places that match the search conditions. Some great online search algorithms (such as "agrep" is used inside glimpse, or "cgrep" is used inside compressed indices, or "NR-grep") are proposed for searching without any indices in the sub-linear time O(n). However, for short strings (n is small), the practical performances of algorithms of O(n) and O(n^2) are much the same. Therefore, suitable indices are necessary to optimize the performance of search engine.

On the other hand, Directory services are more and more important because of its optimization for searching data. The data stored in Directory servers are almost short strings. The approximate search engine for Directory service must take the properties of short strings into considerations.

In our previous research, we have designed one approximate search engine especially for short strings by using filters to filter out the possible short strings, and then checking for the answers. However the performance of previous search engine needs to be enhanced. In this paper, we propose new architecture and algorithm to optimize the performance of searching for Directory service.

1. Introduction

With the tremendous growth of the E-Commerce, various systems have been widely deployed. The data stored in such E-Commerce system are almost short strings, such as the customer's name, address, telephone number or other information. These short strings are usually stored in commercial databases or Directory servers, and are searched for some targets by many applications very often.

However, searching data within these short strings is not the same as searching within longer strings. There are still many factors need to be concerned. General search engines try their best to scan all long strings (articles) quickly, and find out the places that match the search conditions. Some great online search algorithms [8, 10, 4] are proposed for searching without any indices (online) in the sub-linear time O(n). However, for these short strings (n is very small), the practical performance of algorithms of O(n) and O(n^3) is much the same. Even though the fastest algorithm is used, we still have to process all short strings (retrieve each string, and then scan it). The time for retrieving each string is fixed. The enhancement of performance for scanning string is limited. Therefore the search time is almost the same. Under this scenario, suitable indices are necessary for the search performance optimization.

On the other hand, approximate string matching is a recurrent problem in many branches of computer science, with applications to text searching, computational biology, pattern recognition, signal processing, etc. In many E-Commerce systems, the data usually are stored in databases or Directory servers. However, the quality of the string information residing in various databases can be degraded due to a variety of reasons, including human typing errors and flexibility in specifying string attributes. Hence the results of the integration based on exact matching of string attributes are often of lower quality than expected. Therefore, the approximate string

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

matching is very important for integration or searching within data.

The formal approximate string matching problem is: given a large data set D, and a search pattern S of length m (|S| = m), retrieve all the data whose edit distance to the pattern is at most given k. The edit distance between two strings x and y, denoted d(x, y), is defined as the minimum number of character insertions, deletions and replacements to convert x into y. More detailed definition is in section 2.2.

In the on-line version of the problem, the pattern S can be preprocessed but the data set D cannot (no index). The classical solution uses dynamic programming and is O(|x|*|y|) time [7]. Figure 1 shows an example.

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	2	2	2

Figure 1. The dynamic programming matrix to search the pattern "survey" and the string "surgery"

As shown in Figure 1, the algorithm, Levenshtein Distance method [7], to compute the edit distance d(x, y) is based on dynamic programming. Imagine that we need to compute d(x, y). A matrix $C_{0\ldots|x|,0\ldots|y|}$ is filled, where $C_{i,j} = d(x_{1..i}, y_{1..j})$, so $C_{|x|,|y|} = d(x, y)$. This is computed as:

$C_{i,0} = i$, $C_{0,j} = j$,
$C_{i,j} =$ if $(x_i = y_j)$ then $C_{i-1,j-1}$
else $1 + \min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1})$

The dynamic programming method is O(|x|*|y|) time, and is relatively expensive. A number of algorithms improved this result (more details in [5]). The lower bound is O(n(k + log m) / m), which is of course O(n) for constant m. However, if the data set is large and each string is short, the fastest on-line algorithms are not practical because it is necessary to scan all data.

Another trend is that of "filtration" algorithms: some filters are applied over the data set to retrieve the possible candidate strings in a short time, and then verify these candidate strings with the expensive (time-consuming) algorithm. Some simple filtration approaches are proposed [3], but the performance is not well. Especially, in [1], they proposed three filter conditions based on n-gram index to solve the problem of approximate string joins in a database, however such techniques have to be supported by the database management system. To prevent this limitation, we had proposed one new independent algorithm [2] based on the same conditions. This algorithm could be applied in any systems without any special database engine support.

There are also some filtration approaches using n-gram index. In [6], they make n-gram index every h characters, and use the "pigeon-hole" principal to prove that there must be one n-gram with at most k/j errors where the answer is. However, the filter is designed for long strings (articles), and not suitable for short strings. When the data strings are short, this method couldn't split the short strings into j disjoint pieces. The filter method will lose its functionality for filtering.

In this paper, we focus on the optimization for the performance of approximate search engine. Based on the original algorithm, we firstly use two new filter conditions to enhance the filter result. Furthermore, we design one new algorithm with our signature method to optimize the performance more. We also implement the new algorithm and experiment with a large amount of real trademark data. Finally, we show the experimental results and analyze the effects.

This paper is organized as follows: Section 2 lists some basic concepts about approximate string matching, section 3 reviews our previous algorithm and data structure, section 4 outlines the new algorithm, section 5 presents the analysis and experimental results, and the last section is the conclusion.

2. Basic Concepts

2.1 Symbol Definition

We define some symbols used in this paper:
S: the search pattern we given.
L: the length of S. (L = |S|).
D: a very large database with short strings.
RID: the unique record identifier in the database D.
D_s: one string in the database D with its unique RID in D.
G_s: the set of all n-grams of D_s . (see the followings for more detail)
$G_{s,i}$: the positional n-grams of D_s starting at the i-th position.
K: the distance error threshold.

2.2 N-grams: Indices for Approximate String Matching

In our algorithm, we choose the n-grams of one string as its indices for the approximate string matching. Firstly, we should define the n-grams of one string:

For a given string s, its n-grams are obtained by "sliding" a window of length n over the characters of s. Since n-grams at the beginning and the end of the string have fewer than n characters from s, we introduce new characters "#" and "$", and conceptually extend the string by prefixing it with occurrences of "#" and suffixing it with occurrences of "$". Thus, each n-gram contains exactly n characters.

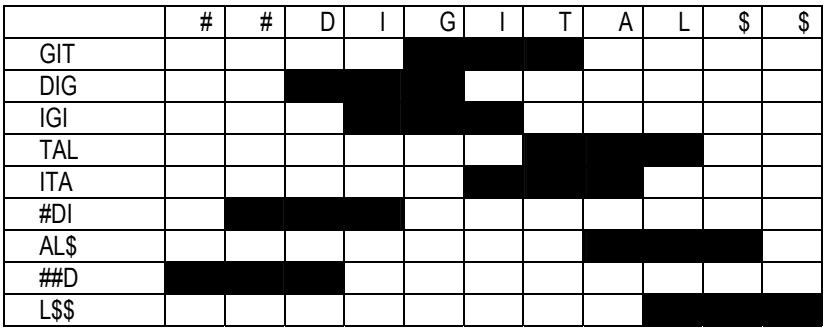

Figure 2. The 3-grams for the string "DIGITAL". The number of 3-grams: 9 = 7 (length) + 3(n) – 1

Another important concept is the "positional n-gram", and here is its definition:

Definition 2.1 [Positional n-gram]: A positional n-gram of a string s is a pair (i, k), where k is one n-gram of s that starts at the position i, counting on the extended string. Symbol G_s is the set of all positional n-grams of a string s, and it contains |s|+n-1 positional n-grams of s. □

The concept behind using n-grams is that when two strings x, y are within a small edit distance, they must share a large number of n-grams in common. Therefore we choose the positional n-grams as the indices to capture the property for one string.

2.3 Number of the n-grams

For one string s (its length is |s|), the number of its n-gram is |s| + n –1. Figure 2 shows the example.

3. Our previous work and data structure

In this section, we will quickly review our previous algorithm [2] and data structures for approximate string matching.

3.1 Filtering technique using n-gram

For one very large database D with short strings, three basic filter conditions are used to filter out impossible strings with edit distance less then k. These three filtering conditions are based on the n-gram and edit distance situation. The key objective here is to efficiently identify candidate answers for our search target, and to prevent computing each real distance by using the "expansive" distance function. Here is the definition of the three filter conditions [2]:

[Count Filtering] For strings x and y, of lengths |x| and |y|, respectively. If the equation d(x, y) <= k holds, then the two strings must have at least (max(|x|, |y|) – 1 – (k – 1) * n) number of the same n-grams. □

[Position Filtering] For strings x and y are within k edit distance, a positional n-gram in x cannot correspond to a positional n-gram in y that differs from it by more than k positions. □

[Length Filtering] String length provides useful information to quickly prune impossible strings. For two strings x and y are within k edit distance, their lengths cannot differ by more then k. □

3.2 Index Architecture

In our algorithm, the positional n-grams are used as the indices for strings. Firstly, we should compute all positional n-grams (G_s) for each string in database, and then store them into one large table (called "Index Set"). The Index Set has four fields: 1: N-gram. 2: String length (denote L). 3. Position (the position which n-gram appears). 4. RID (the unique number of string in database). All indices stored in Index Set are sorted by the order from 1 to 4. Example 3.1 shows all stored positional n-grams for one string "HELLO" in Index Set.

Example 3.1 [Index Set] Assume that one string s = "HELLO", Length(s) = | s | = 5. We use 3-gram as index (n = 3), and then we get the following 3-grams: $G_{s,1}$="##H", $G_{s,2}$="#HE", $G_{s,3}$="HEL", $G_{s,4}$="ELL", $G_{s,5}$="LLO", $G_{s,6}$="LO$", $G_{s,7}$="O$$". All the indices are stored in Index Set as figure 3.

N-grams	Length	Position	RID
##H	3	1	00001
#HE	3	2	00001
HEL	3	3	00001
ELL	3	4	00001
LLO	3	5	00001
LO$	3	6	00001
O$$	3	7	00001

Figure 3. The indices for string Ds using 3-grams

3.3 Approximate Searching Processes

Our approximate searching processes are listed as follow:

Step 1. For each string D_s, we firstly produce all the positional n-grams of D_s.
Step 2. For one n-gram, we quickly retrieve all records in Index Set with the same n-gram as one filter list corresponding to this-gram. We have to retrieve all filter lists corresponding to all n-grams.
Step 3. For all filter lists, we simply count the records with the same RID. If the number is greater then the Count Filtering, the record with the RID maybe is the answer. Then we check it by using the Length Filtering, and then add it into the last result list if it passes the filter.
Step 4. Use the distance function to compute the real distance of the records in the last result list.

Some problems arise during these processes, especially when the amount of record in filter lists is very large. Therefore we need an efficient method for these merge processes. We sort the records in each filter list by record id (RID) field, and then perform one process like the merge-sort algorithm. The following j iterations illustrate the process:

List 1 => Result List (initiation).
List 2 + Previous Result List => new Result List (remember that we sort the records by record id (RID), we could finish the counting linearly in time O(n)).
List j + Previous Result List => new Result List

During these merge iterations, we could observe that the records in preceding lists also appear in the latter lists, and the space and time used for counting increases quite substantially. For the purpose to reduce the space and time, we sort all lists by its size beforehand. The goal of Directory service is to optimize searching. Therefore we design all data structures to reduce searching time by using reasonable space. Figure 4 illustrates our searching processes and data structures.

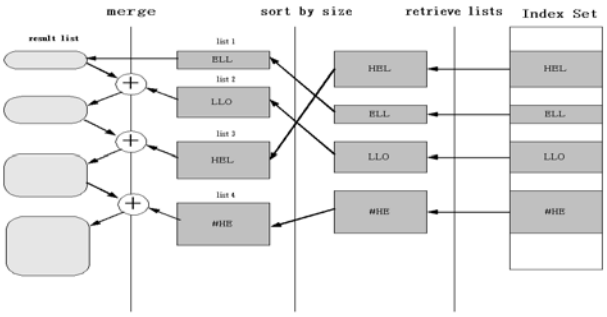

Figure 4. Lists contain fields (length, id, pos) and are sorted by each field. Result lists contain fields (length, id, count) and are the candidate set for each filtering process

4. Improvements

In this section, we will propose two additional filter conditions based on the previous architecture and algorithm to optimize the performance of filtering.

4.1 Minimum Difference

The first filter condition is "Minimum Difference" (denoted by MD). The idea of MD is very simple, that is, for any string s in the list during the searching processes, we want to compute the possible error characters between s and our search pattern. In other words, for example, if one string s doesn't have three n-grams constructed from search pattern, could we know the minimum number of mismatched characters? That is the lower bound of edit distance. If we could compute the minimum edit distance between s and search pattern, the impossible strings will immediately be dropped off during search processes when the minimum edit distance is greater then k (the error threshold).

We define the "minimum difference" of s, md(Gs), that is the minimum number of possible error characters in s so let s mismatches previous n-grams. Figure 5 shows one example. If one string "DIGITAL" doesn't match the first four n-grams, then maybe the characters "G" and "T" are different with the search pattern. Therefore the four n-grams constructed from search pattern can't match the string "DIGITAL". The minimum number of possible error characters is 2, that is, the edit distance between "DIGITAL" and search pattern at least is 2. If we search the pattern with approximate error k = 1, then the string "DIGITAL" could be im

mediately dropped off after the fourth iteration.

	#	#	D	I	G	I	T	A	L	$	$
GIT					♦	♦	♦				
DIG			♦	♦	♦						
IGI				♦	♦	♦					
TAL							♦	♦	♦		
ITA						♦	♦	♦			
#DI		♦	♦	♦							
AL$								♦	♦	♦	
##D	♦	♦	♦								
L$$									♦	♦	♦

Figure 5. The minimum difference is 2 for n-gram subset {GIT,DIG,IGI,TAL}

The basic idea of the minimum difference is to compute the low bound of edit distance when one string doesn't have some n-grams of pattern string S. We could use the property to filter impossible strings during the search processes.

4.2 Filter Condition

In this section, we will introduce two new filter conditions for approximate string matching based on the edit distance. The key objective here is to efficiently identify candidate answers by using the properties of the n-grams.

Condition 1:

We firstly define the minimum difference cut line (D-cut): During the search processes (1 to j), each process uses one n-gram to match all strings. In our algorithm, we first choose the order of filter lists corresponding to the n-grams constructed from search pattern, and then follow the iterations to filter out strings. Now we could firstly compute the minimum distance of previous k-1 n-grams at k-th iteration. For example, in fig. 5, the order of n-gram is {**GIT, DIG, IGI, TAL**}. If we are in the fourth iteration, any string that doesn't appear in the result list will have the edit distance >= 2 (computed in the fig. 5). For the approximate error threshold k, the D-cut is which iteration that minimum distance of previous n-grams is over k.

For condition 1, we compute the D-cut of each searching in advance. When the algorithm reach the D-cut, it could stop to add new candidate strings in the following iterations. All new strings that don't exist in the result list have the edit distance > k. Therefore, the algorithm could stop adding new candidate strings into the result list.

On the other hand, the original "Count filtering condition" also has its cut-line, called C-cut. It says that there are at least ($\max(|S_1|,|S_2|)–1–(k-1)*n$) the same n-grams. Therefore if the number of the rest unprocessed n-grams < ($\max(|S_1|,|S_2|)–1–(k-1)*n$), the algorithm could immediately stop adding new candidate strings. The new strings are impossible to satisfy the Count filtering condition.

The D-cut is better than original C-cut in some situation, and is the same as C-cut in the rest situation. The following example illustrates the situation.

Example: S= "DIGITAL" and edit distance k = 2, and n = 3. We choose the order of n-grams {GIT, #DI, AL$, DIG, IGI, TAL, ITA, ##D, L$$}, the md({GIT, #DI})=2, md({GIT, #DI, AL$})=3, therefore the D-cut = 3. On the other hand, the C-cut = 7 (because 7-1-(2-1)*3= 3). Figure 6 shows the two cut-lines. In this example, the D-cut is better then C-cut, and the algorithm could stop early adding new strings, such as the string "VITALL". Although it contains three the same n-grams "TAL", "ITA", "L$$" and also passes the Count filtering condition, the algorithm stops to add "VITALL" into the result list because of the D-cut line. The D-cut condition will enhance the performance of filtering.

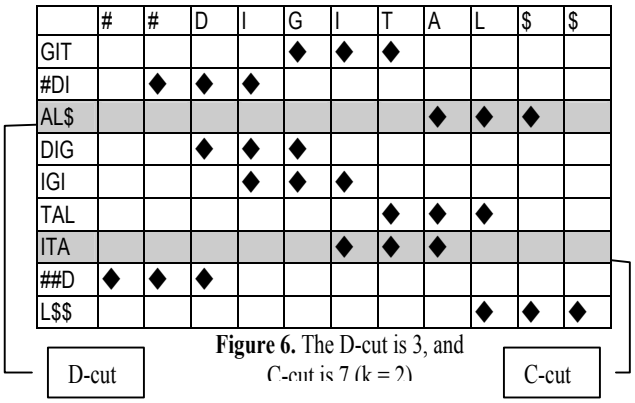

Figure 6. The D-cut is 3, and C-cut is 7 (k = 2)

The condition 1 is the "D-cut" which could be computed before the search, and then improves the performance. Differently with the static "D-cut", we will introduce one dynamic condition into the algorithm.

Condition 2:

This time we will focus on each string in the i-th result list. For each candidate string in the result list, we dynamically compute the minimum distance of some previous n-grams that the string doesn't match. Each string has different mismatched n-grams, and the minimum distance of them is different. The same as condition1, if the minimum distance > k, the candidate string will have edit distance at least k+1, and impossible to be the answer. We should drop off the string early during search processes. We call this minimum distance as "accumulate difference".

To apply the condition 2, we add one bit mask array in the result list to identify which n-gram the string doesn't match. If any candidate string has the accumulate difference > k, it will be marked "DISABLE" at the bit. We don't need to process it in the following processes.

Condition 2 needs to compute each accumulate difference dynamically, and will take much time then condition 1. On the other hand, the condition 1 could compute the D-cut line in advance, called "static", and the algorithm stops adding new candidate strings into result list when D-cut is reached. However, condition 2 focuses on the candidate strings already in the result list, and compute the accumulate difference dynamically to verify them. This is the main different between the two conditions.

4.3 Signature Filter

For more optimization the performance, we combine the filtration algorithm with the signature method. The signature method is making properties indices for each string beforehand, and then using them to filter strings.

As we know, the classical signature method has to scan all signature data for retrieving candidate strings. The drawback of it is taking too much time. However, in our hybrid method, we first apply the n-gram index to quickly filter, and then apply the signature filter. The search time will be substantially decreased. After the first

filter process, the number of candidate strings is relatively small. We just spend a little time to apply the signature filter, and it could filter out more strings than n-gram method does. Now, we first prove the lemma we used, and show our signature architecture later.

For a string $S = c_1c_2..c_n$, where c_i is one character. We define the signature array G_S: for i = 1 to n, if $c_i = j$, then G(j) = 1. The signature array is used to denote which character contained in the string.

Lemma 4.1: Assume that x and y are two strings with length L_x and L_y, and $G = G_x$ XOR G_y, that is, G is from the bitwise xor operation between G_x and G_y. If $d(x, y) < k$, the number of "1" in G array at most is: $2 * k - abs(L_x-L_y)$, where $abs(L_x-L_y)$ is the absolute value of L_x-L_y.

Explanation:

By the definition of edit distance, insertion and deletion at most change one "1" in G (0 -> 1 or 1 -> 0), and the replacement at most changes two "1"s in G (one character disappears and another character appears). Therefore, we could analyze that: (Suppose that x and y are with k edit distance)

1. When $abs(L_x-L_y) = k$, there must be k insertions or deletions. Therefore, G has at most k bits with '1'.
2. When $abs(L_x-L_y) = k-1$, there must be k-1 insertions or deletions, and at most 1 replacement. Therefore, G has at most $(k-1) + 2 * 1 = k+1$ bits with '1'.
3. When $L_x = L_y$, there must at most be k replacements. Therefore, G has at most 2*k bits with '1'.

Therefore, G has at most $2 * k - abs(L_x-L_y)$ bits with '1'□

Signature Architecture

In order to save time and space, we use one long type (32 bits) as our signature, and it could be easily loaded into memory for filtering. The following is its architecture:

Basic section (bit 0 ~ 26)	Extended section (bit 27~31)
ABCDE…YZ, space	AEIOU (happen twice or above)

We divide the signature into two parts. The first 27 bits denote the existence of A to Z and space character. The extended 5 bits (27~31 bits) denote whether the five characters ('A','E','I','O','U') happen twice or above.

In our experiment, the first part is the basic signature filter. The extended part is to enhance the performance.

5. Implementation and experimental results

In this section, we present the results of an experimental comparison. We start in section 5.1 by describing the data sets used in our experiments. In section 5.2 we discuss the programs developed to compare different approaches for approximate string matching. Finally, in section 5.3 we report some experimental results and analysis.

5.1 Data Set

All data used in our experiments are the strings of real trademarks. The data set contains about 510,000 short strings that generate more then 5,000,000 n-gram data. In order to experiment different n-grams, we generate four kinds of n-grams (2-grams, 3-grams, 4-grams, and 5-grams).

5.2 Programs

Our programs use the "Levenshtein" distance algorithm [7] to compute the edit distance. They could search under different situations, such as edit distance or n-gram.

In our experiments, the programs randomly choose 100 strings of each length 5, 8, 10, and 15 for testing, totally 400 string as our search patterns. To experiment every different situation, we search the 400 strings under different combination of k (2, 3, 4) and n (2, 3, 4, 5) situations. Finally, programs output the average results.

Our programs implement four different algorithms:

(1) Original algorithm just uses the original three filter conditions.
(2) Including (1) and two new filter conditions.
(3) Including (2) and signature (without extended) algorithm.
(4) Including (2) and signature (with extended) algorithm

5.3 Experimental results and Analysis

In our experiments, we search strings with different filter method ((1), (2), (3), (4)), under different combination of n-gram (2, 3, 4, 5), and k (2, 3, 4, 5). These reports are divided into five parts:

1. In the first experiment, we want to know about the effect of the four different algorithms upon the final candidate size. We default use 2-gram (n = 2), and k = 2, 3, 4 to search strings with different lengths (L = 5, 8, 10, and 15). The figure 7 lists the results. For showing the results easily, we choose the log value of candidate size on the axis y in all figures. There are some unsuitable (L, n, k) combinations for the Count filtering condition, and therefore, we don't list these unsuitable results in our figures.

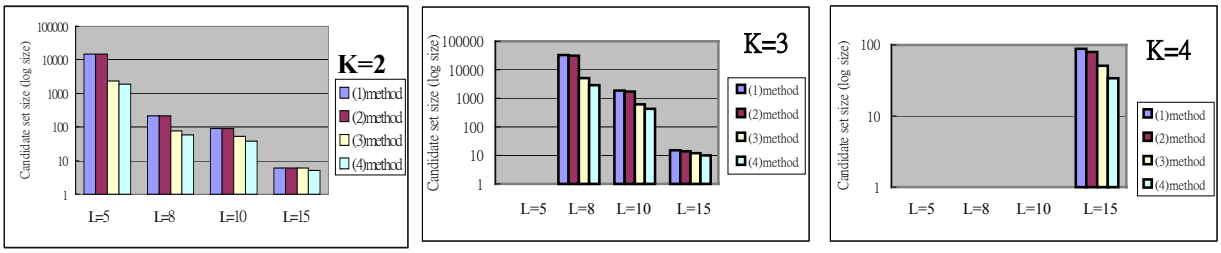

Figure 7. The first results under situations n = 2, and k = 2, 3, 4, for the four algorithms.

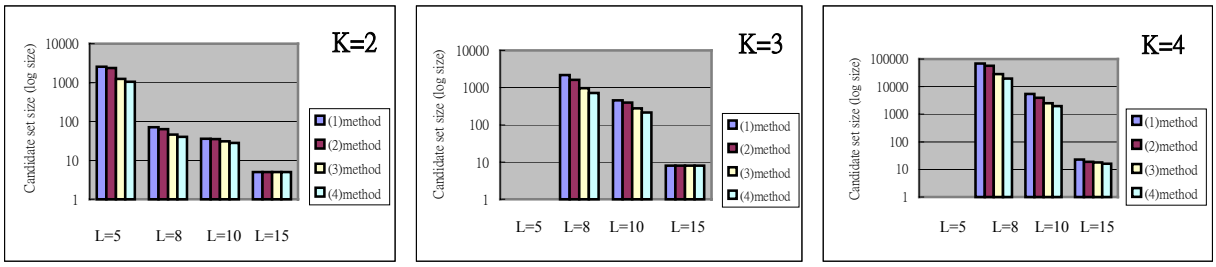

Figure 8. The second results. (N = 3, K = 2, 3, 4)

According to the results shown in Fig.7, we could find out that, no matter under different L and k situation, the best is algorithm (4), and then is algorithm (3), and algorithm (2). Finally algorithm (1) is the worst.

2. In the second experiment, we use 3-gram (n=3) to search strings for observing the influence of different k and L. The figure 8 shows the results.

According to the results shown in Fig.8, we could find out that, especially when L is large and K is small, the error ratio (K/L) is relatively small, the size of candidate strings are smallest. The filtering performance is better under the small error ratio.

3. In the third experiment, we want to know about the effect of different n-gram upon the candidate size. We set the k = 2 by default, and search strings with different lengths (L=5, 8, 10, and 15). The figure 9 is the results.

As the fig. 9 shows, when n is small, the accuracy of filtering is higher, but the size of candidate list increases. Therefore, the comparison time during merge processes also increases. There exists a trade-off between the comparison time and the filter accuracy. When the accuracy increases, we save the time to compute real edit distance for candidate strings; however, the comparison time also increases.

According to these results, there are some facts we discover:

1. We just get a small promotion for filtering accuracy when we apply two new filter conditions (algorithm (2)). The hybrid method with n-gram and signature (algorithm (4)) causes a great improvement on the accuracy.
2. The hybrid method has a great deal of promotion for filtering accuracy, especially when both K and N are large, and that is the worst situations for algorithm (1).

6. Conclusion

For the optimisation for approximate search engine, we successfully combine the signature and n-gram method for improving the filtering accuracy and performance. According to our experimental results, the algorithm (4) is better than the original algorithm. However, some questions still should be more considered:

1. We should consider the time and space trade-off, and by using the properties of data set to adjust the signature for special needs. In our experiment, the extended part of the signature supports a great effect on the accuracy.

2. For the two new filter conditions, the D-cut value depends on the n-gram order as well as the candidate lists. The trade-off between comparison time and cut-line needs to be studied more.

Reference

[1] L. Gravano and P. G. Ipeirotis and H. V. Jagadish and N. Koudas and S. Muthukrishnan and D. Srivastava. Approximate String Joins in a Database (Almost) for Free. In *Proceedings of the 27th VLDB Conference, 2001.*

[2] Chi-Chien Pan and Kai-Hsiang Yang and Tzao-Lin

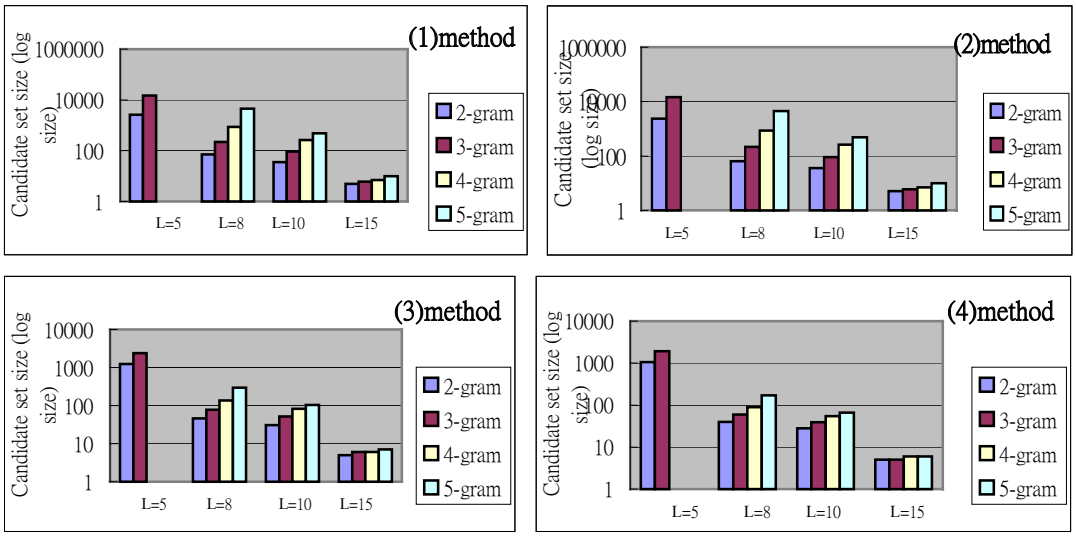

Figure 9. The third results. (K = 2)

Lee. Approximate String Matching in LDAP based on edit distance. In *Proceedings of the IPDPS2002 Conference, 2002*.

[3] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proceedings of ESA'95, 1995*. LNCS 979.

[4] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. In *Journal of Molecular Biology, 147: pages 195-197, 1981*.

[5] U. Manber and S. Wu. Glimpse: A tool to search through entire file system. In *Proceedings of USENIX Technical Conference, pages 23-32, 1994*.

[6] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proceedings of Third Annual European Symposium, pages 327-340, 1995*.

[7] Levenshtein Distance method to compute the real distance. (http://www.merriampark.com/ld.htm).

[8] P. Sellers. The theory and computation of evolutionary distances. In *pattern recognition. Journal of Algorithms, 1:359-373, 1980*.

[9] W. Masek and M. Paterson. A faster algorithm for computing string edit distances. In *Journal of Computer and System Sciences, 20:18-31, 1980*.

[10] G. Landau, E. Myers, and J. Schmidt. Incremental string comparison. In *SIAM Journal on Computing, 27(2):557-582, 1998*.

[11] R. Cole and R. Hariharan. Approximate string matching: a simpler faster algorithm. In *Proceedings of ACM-SIAM SODA'98, pages 463-472, 1998*.

[12] G. Myers. Incremental alignment algorithms and their applications. In *Technical Report 86-22, Dept. of Computer Science, University of Arizona, 1986*.

[13] W. Chang and E. Lawler. Sublinear approximate string matching and biological applications. In *Algorithmica, 12(4/5):327-344, 1994. Preliminary version in FOCS'90, 1990*.

[14] T. Bozkaya and Z. M. Ozsoyoglu. Distance based indexing for high dimensional metric spaces. In *Proceedings of String Processing and Information Retrieval Symposium (SPIRE'99), pages 16-23, 1999*.

[15] Gonzalo Navarro, Erkki Sutinen, Jani Tanninen and Jorrna Tatbio. Indexing Text with Approximate q-grams. In *Proceedings of CPM'2000,LNCS 1848. P. 350-363,2000*.

[16] G. Navarro and R. Baeza Yates. A new indexing method for approximate string matching. In *Proceedings of CPM'99,LNCS 1645, pages 163-186, 1999*.

[17] Gonzalo Navarro. A Guided Tour to Approximate String Matching. In *ACM Computing Surveys 33(1):31-88, 2001*.

[18] Gonzalo Navarro and Ricardo Baeza-Yates. Improving an Algorithm for Approximate String Matching. *Algorithmica 30(4):473-502, 2001*.