# Approximate String Matching in LDAP based on edit distance

Chi-Chien Pan and Kai-Hsiang Yang and Tzao-Lin Lee
Department of Computer Science and Information Engineering,
National Taiwan University, Taipei, Taiwan, R.O.C.
E-mail: {d5526001, f6526004, tl_lee}@csie.ntu.edu.tw

**Abstract**

*As the E-Commerce rapidly grows up, searching data is almost necessary in every application. Approximate string matching problems play a very important role to search with errors. Against these problems "Edit distance" and "Soundex" are two common techniques, especially the latter one is a "sound-like" method and had been applied to the LDAP server. Nevertheless, it is not adequate for certain situations especially when we perform the symbol matching (as in DNA); it doesn't make sense to use the "sound-like" method. On the other hand, "Edit distance" has a clear definition and also is widely used in many fields of application. Since the design of LDAP server is optimized for reading, applying edit distance technique to LDAP server has the problem of lowering speed. In this paper we design efficient data structures and an algorithm to solve the speed problem, and furthermore we use three filter conditions [1] based on the n-gram technique to achieve a well filter performance. Finally we also demonstrate experimentally the benefits of applying our algorithm and its limitations.*

## 1. Introduction

With the explosive deployment of the E-Commerce, more and more companies update their application systems to be used in the Internet. No matter the application system is web-based or not, it needs a large number of various applications and services to organize and manage information efficiently in distributed environments. Especially the directory related services are very important for those distributed systems. It could be applied to Operating System, asset management systems, security systems, etc. Furthermore, The Gartner Group [2], a market research firm, predicts that 40% to 90% portion of new software and hardware will be directory related products at end of year 2001 to 2003.

LDAP (Lightweight Directory Access Protocol) is an IETF standard for accessing directory information, and it is the most advanced, popular method for the directory service. It was originally designed to be just a gateway between X.500 directory server agents. LDAP version 1, RFC1478 [3], is a lightweight alternative to the X.500 Directory Access Protocol (DAP). It is simpler and easier to implement than DAP, and uses TCP/IP stack versus the overly complex OSI stack.

LDAP services provide a variety of searching functions, and use "Soundex" method [16] to perform the approximate string matching function. However, it is not adequate for certain applications such as the DNA symbol matching. Traditional approximate string matching models use the edit distance as the measurement of similarity, and they could achieve different similar levels by setting different error threshold values. When we applied edit distance to LDAP server containing a huge amount of data, the performance decreased substantially because it spends a lot of time for computing edit distance of each data. This does not agree with the LDAP server design goal for quick searching, therefore we need efficient methods to reduce the searching time. Two approaches could be considered: one is to optimize the algorithm of edit distance, and the other is to filter out lots of records impossible to be the answers before the computing of edit distance. There are already many papers about the former [8,9,10,11], but in this paper we focus on the latter approach. The goal of our research is to filter LDAP database records efficiently, and to design an algorithm with suitable filter conditions in order to improve the performance. During each filter process, we use the sort and merge methods (like merge-sort algorithm) to reduce the filter time in O(n), this makes our algorithm a good candidate to be incorporated by the LDAP server.

This paper is organized as follows: Section 2 presents related work, section 3 lists some basic

Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)
1530-2075/02 $17.00 © 2002 IEEE

concepts about our algorithm, section 4 outlines the algorithm and data structures, section 5 presents experimental performance results, and the last section is the conclusion.

## 2. Related Work

A lot of researches have been published about the "approximate string matching" problem. For two strings of length n and m, there exists a dynamic programming algorithm to compute the edit distance of the strings in O(nm) time and space [4], and improvements to the average and worst case have appeared [11,12,13].

Indexed approximate string matching is a relatively new problem where it is possible to build some indices beforehand in order to answer queries later. Indices such as built by Glimpse [5] store a dictionary and use an algorithm to obtain a set of words to retrieve. These approaches are limited in scope due to the static dictionary, and they are not suitable for dynamic environments. For the Netscape LDAP servers, they use the "sound-like" method to reduce each word into a short form (such as "Washington" is coded "W252") [16] in order to perform the approximate searching. However, each language may need its own particular sound-like algorithm.

In [1], they solve the problem of approximate string joins in a database, using n-gram as index stored in database and using three filter conditions for quickly joins. In the field of database, several indexing techniques proposed for the "approximately string matching" problem, however such techniques have to be supported by the database management system [14,15].

## 3. Basic Concepts

In this section, we describe some basic concepts about our algorithm.

### 3.1 Approximate String Matching

For any string s, we denote its length as |s|. The problem of approximate string matching is to find all the data strings that match a given pattern with up to k errors. The k value is the threshold of the edit distance.

### 3.2 Distance function

The edit distance d(x, y) between two strings x and y is the minimal cost of a sequence of operations that transform x into y. The cost of a sequence of operations is the sum of the costs of the individual operations. In this paper we use the three standard operations of cost 1 such as follows.

- Insertion: inserting the letter a.
- Deletion: deleting the letter a.
- Replacement: for $a \neq b$, replacing a by b.

### 3.3 N-grams: Indices for Approximate String Matching

For a given string s, its positional 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.

**Definition 3.1 [Positional n-gram]:** A positional n-gram [6] of a string s is a pair ( i , k ) ,where k is the q-gram of s that starts at the position i ,counting on the extended string. The set Gs of all positional n-grams of a string s is the set of all the |s|+n-1 pairs constructed from all n-grams of s. □

The concept behind using n-grams is that when two strings a, b are within a small edit distance of each other, they must share a large number of n-grams in common [6].

### 3.4 Number of the n-grams

For any string s (its length is |s|), we can easily find out the number of its n-gram is |s| + n –1. (Figure 1)

### 3.5 Filtering technique using n-gram

In a very large string database, we use three filter conditions to filter out strings which is impossible having edit distance less then k with a given target string A. In this section, we present the three filtering conditions [1] based on the n-gram and edit distance. The key objective here is to efficiently identify candidate answers to our problem before we use the "expansive" distance function to compute the real distance. The three filtering conditions are as follows:

| | # | # | D | I | G | I | T | A | L | $ | $ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| GIT | | | | | ◆ | ◆ | ◆ | | | | |
| DIG | | | ◆ | ◆ | ◆ | | | | | | |
| IGI | | | | ◆ | ◆ | ◆ | | | | | |
| TAL | | | | | | | ◆ | ◆ | ◆ | | |
| ITA | | | | | | ◆ | ◆ | ◆ | | | |
| #DI | | ◆ | ◆ | ◆ | | | | | | | |
| AL$ | | | | | | | | ◆ | ◆ | ◆ | |
| ##D | ◆ | ◆ | ◆ | | | | | | | | |
| L$$ | | | | | | | | | ◆ | ◆ | ◆ |

**Figure1: The 3-grams for the string "DIGITAL". The number of 3-grams: 9 = 7 (length) + 3(n) – 1.**

**Count Filtering:** Consider strings s1 and s2, of lengths |s1| and |s2|, respectively. If the equation $d(s_1,s_2) <= k$ holds, then the two strings must have at least ( max(|S1|,|S2|)–1–(k-1)*n ) the same n-grams. □

**Position Filtering:** If strings s1 and s2 are within an edit distance of k, then a positional n-gram in one cannot correspond to a positional n-gram in the other that differs from it by more than k positions. □

**Length Filtering:** The last condition is that string length provides useful information to quickly prune strings that are not within the desired edit distance. If two strings s1 and s2 are within edit distance k, their lengths cannot differ by more then k. □

## 4. Algorithm and data structures

In this section we introduce our algorithm and data structures for the three filtering conditions.

### 4.1 Symbol Definition

We define some symbols below used in this paper:

S: the string pattern we want to search.

L: the length of S. (L = |S|).

D: a very large string database.

RID: the unique recode identifier in the string database D.

Ds: one string in the string database D, which has unique RID in D.

Gs: the set of n-grams of Ds.

Gs,i: the positional n-grams of Ds starting at the i-th position.

K: the distance error threshold.

### 4.2 Index Architecture

During the existent implementations of the LDAP server, some of them (such as OpenLDAP) use n-grams as the indices of each string. We also use the set of n-gram (Gs ) as the original indices. For each string Ds in D, we put the indices Gs into a large table (called "Index Set"). The Index Set contains four fields: 1.n-gram 2.string length (denote L) 3.position (the position which n-gram appears) 4.RID.

**Example 4.1 [Index Set]** Assume that string Ds = "HELLO", Length(Ds)= |Ds| = 5, we use the 3-grams as indices(n = 3), then we get the following 3-grams: $G_{3,1}$="##H", $G_{3,2}$="#HE", $G_{3,3}$="HEL", $G_{3,4}$="ELL", $G_{3,5}$="LLO", $G_{3,6}$="LO$", $G_{3,7}$="O$$". We collate all the indices into a table as figure 2 shows:

| 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 2: the indices for string Ds using 3-grams.**

### 4.3 Searching Processes and Structures

The approximate searching processes using n-gram is as follows:

1. For each string Ds, we produce all the n-grams of Ds.
2. Retrieve each filter list in the Index Set corresponding to each n-gram.
3. In all the filter lists, we sum the records which have the same RID. If the sum is greater then the Count Filtering, the record with the RID maybe is the answer. Then we check it for the Length Filtering, and add it into the last result list when it passes the condition.
4. Use the distance function to compute the real distance for 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 records in each filter list by record id (RID) field, like the merge-sort algorithm. The following j iterations present the method:

List 1 => Result List (initiation).
List 2 + Previous Result List => new Result List (because we sort the records by record id (RID) in lists, we can do the counting linearly in time O(n) ).
List j + Previous Result List => new Result List

During the Merge iterations, we can observe that the preceding list records 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 size beforehand, and the first list has the smallest size. The goal of LDAP server is for searching quickly, therefore we design all data structures to reduce searching time by using reasonable space. Figure 3 is our searching process and data structure.

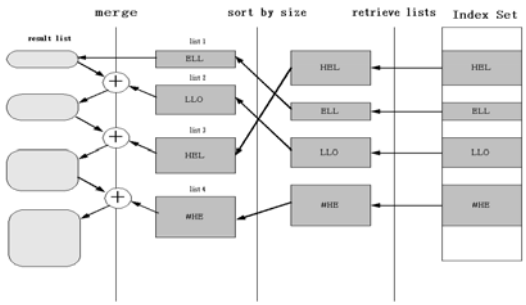


**Figure 3. Lists contain fields (length, id, pos) and are sorted by each field. Result lists contain fields (length, id, count) and are the candidate set for each filtering process.**

### 4.4 Algorithm Design

In this section, we propose our efficient algorithm using the three conditions to filter data efficiently. We use the example string "HELLO" and 3-grams to explain our algorithm.

**Algorithm 4.1 [Algorithm using n-grams with sorted Index Set]**

1. We produce the 3-grams of the S = "HELLO", and easily retrieve lists corresponding to its 3-grams in the sorted Index Set. Then we sort lists by size. For example, we retrieve the "HEL" list containing 10,000 records, and the "LLO" list containing 5,000 records, and "ELL" list containing 100 records, and go on. Then we sort the lists by size, the smallest list is called List1, and go on. In general, the n-grams "##H" and "O$$" have a lot of records, and are sorted to be the last ones.
2. From List1 to List 7(because S has 7 3-grams), we cut each list by the Length Filtering to a small one. For example, S = "HELLO", L = |S| = 5, assume K (distance threshold) = 2, then we have the cut condition ( 3 <= L <= 7) to filter each list .
3. During merge processes, we filter out many strings by using Count Filtering and Position Filtering to avoid computing real distance because of its expensive cost. These processes check two conditions: one is to check whether the distance of corresponding n-grams (called positional n-grams) is less then or equal to K = 2, and the other is to check whether total number of same n-grams passes the Count Filtering, Especially we can filter out any record immediately when we are sure it is impossible to pass the Count Filtering. For example: S = "HELLO", L = |S| = 5, k = 2, n = 3, when string length |Ds| = 7, it must have at least 7-1-(2-1)*3 = 3 same 3-grams with S.
4. After the merge processes we use distance function to compute real distance for the records in the last result list.

**Algorithm 4.1 [The formal definition]**

1. Retrieve all lists corresponding to each n-grams of the search pattern S.
2. Filter each list by the Length Filtering.
3. Sort all lists by size in ascend order.
4. Prepare a null result list, and two pointers for the result list and list 1.
5. For j = 1 to k
   5.1 Move the pointer to next record in the list j
   If there is no data, go to (3), else check the Position Filtering, if it passes the filter, then add

record score by 1, score = score + 1.
Check whether the next record has the same pair (L, RID). If it is true, check the condition again, and update the score of the record. Then we can get the triple (L, RID, score).

5.2 Check the record in the result list:

5.2.1 If (L, RID) result < (L, RID) List j: check whether the score of the record in the result list plus the number ((L+n-1)-j) is less then ( ), if it is true, we delete the record immediately by the Count Filtering, else we insert the record of the result list into new result list.

5.2.2 If (L, RID) result = (L, RID) List j : sum the two score in two lists, and check whether the score plus (L+n-1)-j is less then , if it is true, we delete the record, else we insert the record into new result list.

5.2.3 If (L, RID) result > (L, RID) List j: check whether the score of the record in the list j plus the number ((L+n-1)-j) is less then the value ( ), if it is true, we delete the record, else we insert the record of the list j into new result list. Go to (1).

5.3 Check the remnant records for the condition above, if it is true, we insert it into new result list.

5.4 Change the new result list as result list.

Next j

6. For the last result list, we compute the real distance for each record. Then we get the answer set.

# 5. Experimental performance

According to our algorithm and data structures, we developed programs and used a lot of practical data to verify the effectiveness of the filter. Furthermore, we experiment on different n-gram, string length (L), and distance value (k) to evaluate their relation. In this section, we start in Section 5.1 by describing the implementation environment. In Section 5.2, we evaluate the performance and list their limitations.

## 5.1 Environment

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.

Our platform is the "OpenLDAP" system which is developed by LDAP community and is an open source (http://www.openldap.org), and we also use the DB library developed by the University of Berkeley.

Our programs contain two parts:

**1. Index Generation**

The part is responsible for generating n-grams and making the sorted Index Set mentioned above, and sorting lists, etc. In order to experience different n-grams we generate four different grams (2-grams, 3-grams, 4-grams, 5-grams).

**2. Filtering and Searching**

Programs could search for different parameters such as edit distance or n-gram. Furthermore, we also add some parameters like "sort-order" for the purpose of experiments.

Our programs use the "Levenshtein" distance algorithm [7] to compute the real distance between two strings.

## 5.2 Performance Analysis

In this section, we perform three experiments on the relations between L, k, and n. Programs chose dynamically 30 strings of length 5, 8, 10, 15 respectively to search, then reported the average results. The experiments are:

5.2.1 First, we want to evaluate the effect of different L and k values upon the filter performance. Under fixed n-grams (n=2, 3, 4, 5) we searched for different (L, K) pairs, and recorded the amount of real answers ("answer" in the figure 4) and the amount of candidate records passed the three filter conditions ("filter" in the figure 4). Figure 4 shows the results.

**Analysis:** As the figure 4 shows, we could discover that:

(1) Some (L, K) pairs are not suitable for the filter, because the triple (L, k, n) makes the record to be an answer even it has 0 same n-grams. Therefore, the filter loses its functionality.

(2) The k value increases with the decrease in filter performance under all n-grams. Furthermore, as the increase of L, we could still have good performance for a bigger k.

(3) The algorithm filtered candidate strings from 510,000 to hundreds, and therefore it could search very quickly. These results proof the effectiveness of the algorithm.

5.2.2 In the second experiment, we want to understand the effect of the n-grams and sort order upon the amount of comparisons during the merge processes. Therefore under fixed k value (k=1, 2, 3, 4), we searched for different (L, n) pairs and uses three different kinds of sort order (S: Sequential, A: Ascend, D: Descend). The results are presented in Figure 5.

**Analysis:** As the figure 5 shows, the comparison count is smallest under the ascend sort order. Besides,

the comparison count decreases as we use bigger n-grams under the ascend sort order.

5.2.3 In the third experiment, we want to know about the effect of different (L, n) pairs upon the filter performance.

**Analysis:** In the results showed in the figure 6, we can discover that:

(1) The case (K=1) is a special situation. Different n-grams have the same performance, because the Count Filtering condition is independent of n when k=1. However for k>1, the filter performance decreases rapidly as the n value increases. Another fact is that, the performance increases with the L value.

5.2.4 According to the results from (1) to (3), we know the limitation for the three parameters is from the Count Filtering: $\max(|S1|,|S2|) - 1 - (k-1) * n > 0$.

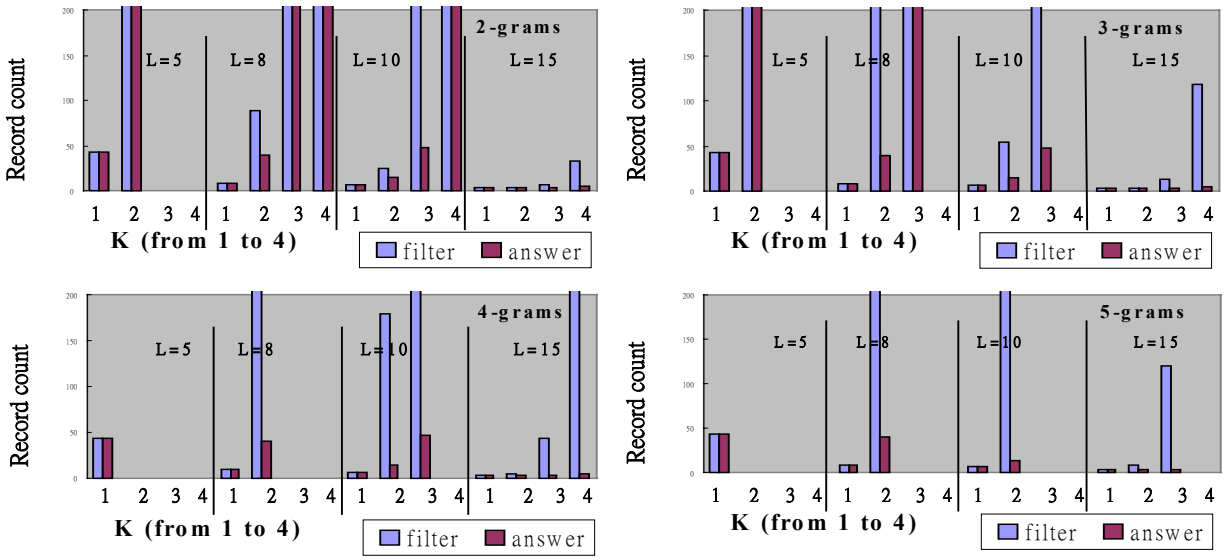


**Figure 4: the results of (1).**

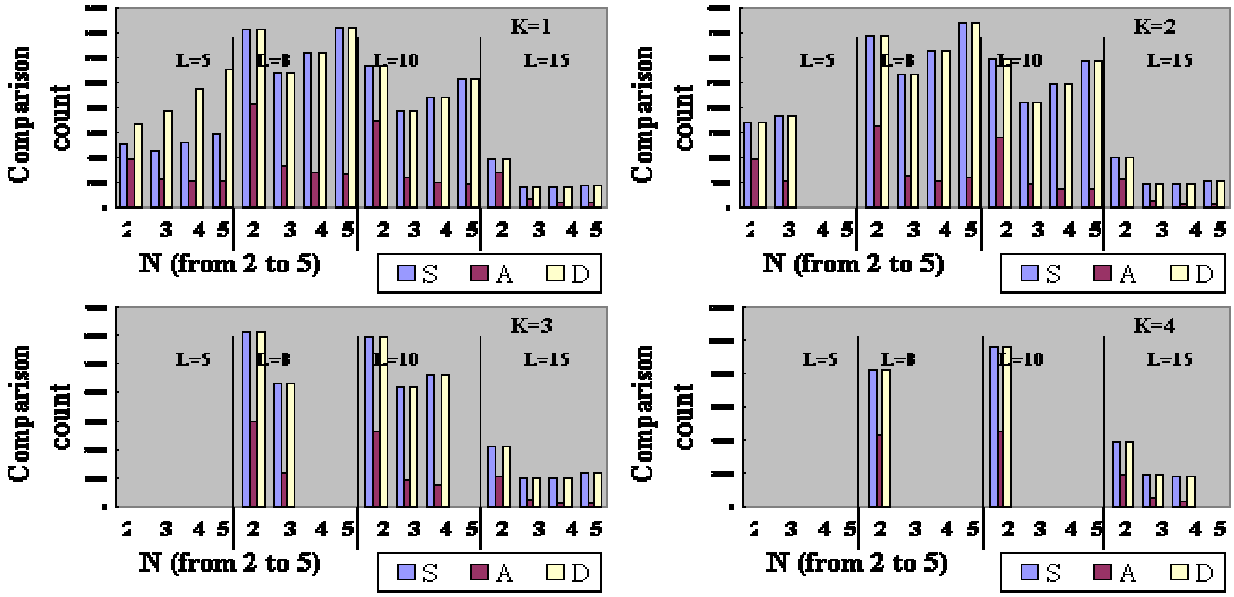


**Figure 5: the result of (2).**

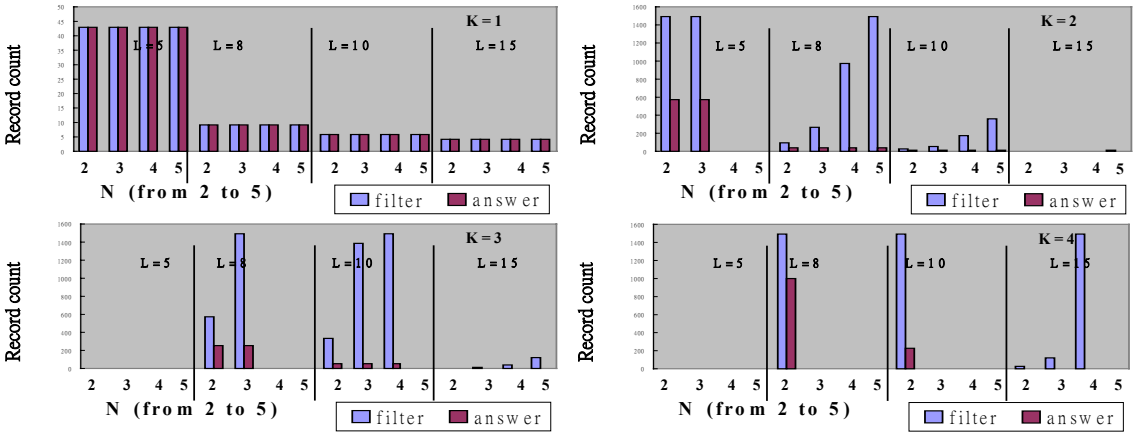


**Figure 6: the result of (3).**

## 6. Conclusions

We successfully apply the n-gram technique to LDAP server for approximate string searching. By sorting the n-grams into the Index Set we improve the search performance, especially when the program retrieves all the lists correspond to n-grams, and when it merges the lists, like a "merge-sort" process. In other words, we develop an efficient algorithm for searching the approximate strings in the LDAP server. And the algorithm with the sorted Index Set brings the three filter conditions into a full play. Furthermore we list the limitation between the pattern length (L), error threshold (K), and n-gram. The limitation could help programs to choose suitable n-grams for different searching in order to improve the performance.

## References

[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] Gartner's site: (http://gartner4.gartnerweb.com/).

[3] Lightweight Directory Access Protocol (V2) RFC 1777 (V3) RFC2251.

[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] S.Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21st International Conference on Very Large Databases (VLDB'95), pages 574-584, 1995.*

[16] The Soundex Indexing System. (http://www.nara.gov/genealogy/soundex/soundex.html).

[17] "iPlanet" Directory Server. (http://www.iplanet.com/products/iplanet_directory/home_2_1_1z.html).

[18] 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.*

[19] 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.*

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

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