

# Efficient Edit Distance based String Similarity Search using Deletion Neighborhoods

Shashwat Mishra  
Dept. of Computer Science  
and Engineering,  
Indian Institute of Technology,  
Kanpur, India  
shashm@cse.iitk.ac.in

Tejas Gandhi  
Dept. of Computer Science  
and Engineering,  
Indian Institute of Technology,  
Kanpur, India  
tgandhi@cse.iitk.ac.in

Akhil Arora  
Dept. of Computer Science  
and Engineering,  
Indian Institute of Technology,  
Kanpur, India  
aarora@cse.iitk.ac.in

Arnab Bhattacharya  
Dept. of Computer Science  
and Engineering,  
Indian Institute of Technology,  
Kanpur, India  
arnabb@cse.iitk.ac.in

## ABSTRACT

This paper serves as a report for the participation of *Special Interest Group In Data (SIGDATA)*, Indian Institute of Technology, Kanpur in the *String Similarity Workshop, EDBT, 2013*. We present a novel technique to efficiently process edit distance based string similarity queries. Our technique draws upon some previously conducted works in the field and introduces new methods to tackle the issues therein. We focus on achieving minimum possible execution time while being rather liberal with memory consumption. We propose and support the use of deletion neighborhoods for fast edit distance lookups in dictionaries. Our work emphasizes the power of deletion neighborhoods over other popular finger print based schemes for similarity search queries. Furthermore, we establish that it is possible to reduce the large space requirement of a deletion neighborhood based finger print scheme using simple hashing techniques, thereby making the scheme suitable for practical application. We compare our implementation with the state of the art libraries (Flamingo) and report speed ups of up to an order of magnitude.

## Categories and Subject Descriptors

H.3 [Information Storage and Retrieval]: Indexing Methods; D.4 [Performance]: Metrics—*complexity measures, performance measures*

## General Terms

Performance, Measurement, Experimentation

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

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

## Keywords

String Similarity, Deletion Neighborhood, Edit Distance

## 1. INTRODUCTION

Amount of textual data present in the cyber-sphere is growing at a rapid rate. Due to the advent of social networking and micro-blogging sites like Facebook, Twitter etc. production rate of textual data has gained a major boost. Twitter produces more than 5 billion tweets every 5 days. Textual data also feature in several research sub-domains like bio-informatics, information retrieval, data cleaning etc. Conducting analysis on such large amounts of textual data encapsulates several problems. One of these problems is to be able to quantify the similarity between strings and subsequently use the quantification to find all strings similar to a given query string. There have been several attempts to define the notion of distance between two strings. Some of these like Jaccard Index or Cosine Similarity were drawn from the set theory domain. Perhaps the most popular and widely used notion of distance between strings is given by the Levenshtein or the Edit Distance. The Edit Distance between two strings is the minimum number of single character edit operations (insert, replace, delete) required to transform one string to the other. The ability to run fast edit distance based similarity queries on huge databases is a prime requirement for all domains dealing with textual data. Much research has been devoted to developing techniques for answering edit distance based similarity queries which are fast, scalable and memory efficient. This is a challenging problem because edit distance computation is a costly operation and therefore for large dictionaries, naively computing edit distance of the query string with each string in the dictionary is not an efficient method. Most modern strategies use some preprocessing schemes to index the dictionary which facilitates fast evaluation of the queries.

To further the cause of development and more specifically to get a measure of the current state of the art in string similarity queries, a String Similarity workshop was organized as a part of the 16<sup>th</sup> *International Conference on Extending Database Technology (EDBT)*. The competition presented

the participants with the challenge to develop fast methods for answering ‘Similarity Search’ and ‘Similarity Join’ queries over a large string dataset. In this paper we describe the details of the system we developed for the ‘String Similarity Search’ problem. Our system uses the power of deletion neighborhoods to answer threshold based edit distance queries. More specifically, given a dictionary  $D$  and 2-tuple  $\langle q, \tau \rangle$ , it lists all tuples from  $D$  whose edit distance from the query string  $q$ , is less than or equal to  $\tau$ . Our contributions are the following:

- We present a new method which enables practical usage of deletion neighborhoods for answering edit-distance based similarity search queries on large string datasets with moderately large values of the edit distance threshold.
- We propose the use of novel techniques of suffix-based hashing and bucketing schemes for reducing the space requirements of the resulting index structure and decreasing the average query processing time.
- We compare our algorithm with one of the previous state of the art methods and report significant reduction in average query processing time.
- We mention several important avenues which can be pursued to make the algorithm even faster.

The remainder of this article is concerned with showing the basics, the implementation and the results of our system. Section 2 presents a formal description of the problem and mentions the related work. In Section 3 we introduce the notion of deletion neighborhoods and emphasize their power for edit-distance queries. In Section 4 we give the implementation details of the system built to leverage the power of deletion neighborhoods. We compare our system with the state of the art Flamingo Project library over the supplied dataset and present the results in Section 5. Finally we give directions for future work and conclude.

## 2. FORMAL PROBLEM STATEMENT AND RELATED WORK

Consider a dictionary  $D$  containing a large number of strings. The  $j^{th}$  tuple in  $D$ ,  $t_j$ , has the signature  $\langle id_j, s_j \rangle$  where  $s_j$  denotes the string and  $id_j$  denotes the unique identifier associated with the string. Define a query instance,  $Q$ , as a 2 tuple  $\langle q, \tau \rangle$ , where  $q$  denotes a (query) string and  $\tau$  denotes an edit distance threshold. Consider a list of  $k$  such query instances (denoted as  $L$ ), where the  $i^{th}$  instance  $Q_i$ , is given as  $\langle q_i, \tau_i \rangle$ . For every query instance  $Q_i$ , define an answer set  $A_i$  as  $A_i = \{id_j \mid edit\_distance(s_j, q_i) \leq \tau\}$ . Let  $T_L$  denote the total time taken to obtain  $A_i$  for every  $Q_i$  in  $L$ . Given  $D$ , what is the best way to obtain  $A_i$  so as to minimize  $T_L$  for any given  $L$ ?

The problem definition above is very similar to the problem of *Named Entity Recognition (NER)* [4, 8]. In *NER*, it is required to match entities from a document  $d$  with a given dictionary  $D$ . A document is essentially a sequence of characters. A matching can be from any collection of contiguous characters in  $d$  to an entity in  $D$ . Good *NER* algorithms typically use smart strategies to avoid matching every possible contiguous sequence of characters [2, 4, 8]. This typically involves either creating some sort of index structure on the

document  $d$  or continually maintaining information which allows one to skip ahead on a mismatch.

Another related problem is to preprocess the dictionary  $D$  so as to minimize the average computation time per query [3, 7]. Almost all methods that answer such problem use some form of finger print based scheme to index the entities in  $D$  in inverted indices. Associated with every finger print scheme is a ( $\tau$  dependent) filtering criteria which is essentially a *necessary* condition for any two strings lying within  $\tau$  edit distance of each other to satisfy. An example of such a scheme would be the  $q$ -gram signature scheme. A  $q$ -gram signature of a string  $s$  is the set of all contiguous subsequences of length  $q$  that can be obtained from  $s$ . Let  $G_x$  denote the  $q$ -gram signature of string  $x$ , then, if edit distance between  $s_1$  and  $s_2$  is less than  $\tau$  then one can show that  $|G_{s_1} \cap G_{s_2}| \geq T$  where  $T$  is a function of  $|s_1|$ ,  $|s_2|$ ,  $\tau$  and  $q$ . Specifically,  $T = \max(|s_1|, |s_2|) + 1 - (\tau + 1).q$ . One can create inverted index for each  $q$ -gram and use this filtering criteria to answer an edit distance query in the following way. During dictionary preprocessing phase, generate  $q$ -gram signature for every string  $s$  in the dictionary. For each  $b \in G_s$  make an entry of  $id_s$  in the inverted index of  $b$ . Given a query string  $m$ , generate  $q$ -gram of  $m$ ,  $G_m$ . The inverted index of each  $b' \in G_m$  gives a list of strings that also generated  $b'$  as a  $q$ -gram during their own  $q$ -gram generation step. Let  $z = |G_m|$ . Then as per the filtering criteria, the qualified candidates would be those that appear in at least  $T$  of these  $z$  lists. The last statement refers to a very commonly occurring problem known as the *T-occurrence problem*. Several efficient strategies exist to solve this problem [3] and very efficiently obtain the candidate list which is guaranteed not to miss any of the answers. Once the candidate list is obtained one can verify (using  $\tau$ -threshold edit-distance) for each candidate if the edit distance is indeed less than or equal to  $\tau$ . It is expected that higher values of  $q$  would be more selective and hence, generate shorter candidate lists. However arbitrarily increasing  $q$  has a problem. For a query string  $m$ ,  $T$  is initialized as  $|m| + 1 - (\tau + 1).q$ . It is obvious that for the filtering criteria to be meaningful, the value of  $T$  must be  $\geq 1$ . This implies that  $|m| \geq (\tau + 1).q$ . This puts a serious restriction on the usability of this algorithm for high values of  $\tau$  or  $q$ . For example, for  $q = 2$  and  $\tau = 3$ , the algorithm is only useful for queries of length  $\geq 8$  and for  $q = 3$ , it is only useful for queries of length  $\geq 12$ .

A different class of solutions that aim to minimize the average query time involves mapping strings to integer domain by imposing one (or more) *global order* on the strings. Simple examples of such order could be the dictionary order. All strings in the dictionary are sorted according to the dictionary order and the rank of a string forms its integer mapping. In [9] the authors construct a B+ tree on the strings using the integer embedding. A query is then processed by efficiently traversing the tree while ensuring the completeness of the solution.

Our approach for this competition is based on *deletion-neighborhoods* and draws upon the techniques described in [1, 6, 8]. The core idea in all of these studies have been to generate deletion neighborhoods for the query and the dictionary string. Deletion neighborhoods being longer than  $q$ -grams, describe an alternative signature scheme which is more selective than the  $q$ -gram scheme. Unlike the  $q$ -gram scheme which requires the query string to have a minimum length  $L_{min}$ , deletion neighborhood as a finger-print scheme

**Table 1: Terminology**

$s$	original string
$\tau$	edit distance threshold
$s'$	$k$ -neighbor of $s$
$F_k(s)$	$k$ -neighborhood of $s$
$H_\tau(s)$	$\tau$ -variant family of $s$

is applicable for queries of any length and for any value of  $\tau$ . The caveat with deletion neighborhood based schemes is that they have large space requirements. In the next section, we introduce the relevant terms and give the details of a generic *deletion neighborhood* based string similarity answering system.

### 3. DELETION NEIGHBORHOODS

Consider a string  $s$  of length  $l_s$ . Assume we are to delete  $k$  characters from  $s$ . We denote  $r$  as a bit vector of length  $l_s$  with a 1 in each of the  $k$  positions which are selected for deletion in  $s$ . Clearly, for a fixed  $k$ , multiple  $r$ 's are possible. For a fixed  $k$ , we define  $D_k$  as the set of all possible  $r$ 's. Clearly,  $|D_k| = \binom{l_s}{k}$ . Consider some  $r \in D_k$ . Deleting the  $k$  characters as specified in  $r$  will result in a unique string  $s'$ . We say  $s'$  is a  $k$ -neighbor of  $s$ . Qualitatively, a string  $s'$  is a  $k$ -neighbor of  $s$  if  $s'$  can be generated by deleting  $k$  characters from  $s$ . We define  $k$ -neighborhood (denoted  $F_k(s)$ ) as the set of all  $k$ -neighbors of  $s$ . It is possible that a  $k$ -neighbor  $s'$  generated via  $r$  is also generated by some other  $r' \in D_k$ ,  $r' \neq r$ . For ex. the string  $AK$ , which is a 2-neighbor of string  $AKKA$  will result for  $r = 0011$  and  $r = 0101$ . We now define the  $\tau$ -variant family of a string  $s$ , denoted  $H_\tau(s)$  as the following.

*Definition 1.* For a given string  $s$  and a given value of  $\tau$ , define  $H_\tau(s)$  as  $H_\tau(s) = \{s' : s' \in F_k(s), 0 \leq k \leq \tau\}$

Consider any two strings  $s_1$  and  $s_2$ . If  $ED(s_1, s_2) \leq \tau$  it can be shown that  $H_\tau(s_1) \cap H_\tau(s_2) \neq \emptyset$ . The proof is simple and we refer the readers to [6] for the same. The underlying idea is that if the total number of single edit operations required to transform  $s_1$  into  $s_2$  is  $\leq \tau$ , then the two strings should be reducible to a common form after performing at the max  $\tau$  deletions. Since  $H_\tau(s)$  is qualitatively the set of all possible subsequences of characters in  $s$  that can be generated by deleting  $k$  characters from  $s$  (where  $0 \leq k \leq \tau$ ) the common form should be present in both  $H_\tau(s_1)$  and  $H_\tau(s_2)$ .

*Theorem 1.* If  $ED(s_1, s_2) \leq \tau$  then  $H_\tau(s_1) \cap H_\tau(s_2) \neq \emptyset$

Using  $H_\tau(s)$  as a finger-print scheme and Theorem 1 as a filtering condition we can now construct an algorithm which leverages the power of deletion neighborhoods for efficiently answering string similarity search queries.

It can be easily inferred that the selectivity of a  $q$ -gram finger-print scheme is not as high as that of the *deletion-neighborhood* finger-print scheme. To illustrate this, consider the following example. Say we need to find all strings in the dictionary with in 2 edit distance of the string *placating*. Lets say that the  $q$ -gram signature scheme uses  $q = 3$ . Then for  $\tau = 2$ ,  $T = 9 + 1 - (2 + 1) \cdot 3 = 1$ . Therefore the candidate list includes any string that appears in the inverted index of any 3-gram of *placating*. Notice that *ing* will be

a  $q$ -gram of the string. Any string in the dictionary which ends in an *ing* will be in the inverted index of *ing*. All such strings will be valid candidates as per the filtering criteria ( $T = 1$ ). It is clear that the candidate list will be rather large. Next consider how *deletion-neighborhood* finger-print scheme would handle the same query string. We would first generate the 2-variant family of *placating*. Notice that every string in  $H_\tau(s)$  will be at least 7 characters long. Thus all occurrences of *ing* in the  $\tau$ -variant family will be preceded by at least 4 characters. This would drastically reduce the total number of strings ending in *ing* that would appear in the candidate list. Notice that the size of  $\tau$ -variant family,  $|H_\tau(s)|$ , would be much larger,  $O(\binom{l}{\tau} + \binom{l}{\tau-1} \dots \binom{l}{0})$ , than the total number of  $q$ -grams,  $O(l - q + 1)$ . The number of lists that need examining would therefore increase for *deletion-neighborhoods* scheme but every string in it will be highly selective. Thus the total number of candidates would, expectedly, be very low.

It is clear that *deletion neighborhoods* offer a very selective finger-print scheme for strings. However, they have not been incorporated into modern systems. The reason, as mentioned previously, is the amount of space required to index every single string in  $H_\tau(s)$  for every string  $s$  in the dictionary for a range of values of  $\tau$ . For a fixed edit distance threshold  $\tau$ , alphabet set size  $\Sigma$ , average string length  $l$  and total number of string in the dictionary  $n$ , the space requirement (if one were to index every string in  $H_\tau(s)$  for every string  $s$ ) would be  $O(nl^\tau \Sigma^\tau)$  [8]. The large space requirement makes the algorithm impractical for real world applications. It would be very lucrative to reduce the space requirements. This is precisely the point that previous studies involving *deletion neighborhoods* have addressed. In [1] the authors reduce the space requirement per string to  $O(l^\tau)$ . Despite the significant improvement, the algorithm still has heavy space requirements. In [8] the authors have further reduced the space requirement to  $O(l_p \tau^2)$  ( $l_p$  being a design parameter) making the system practical for general values of  $\tau$ . The idea in [8] is essentially to generate and index  $\tau$ -variant family for partitions of strings instead of the entire string. This significantly reduces the space complexity. The authors further impose a condition on the  $\tau$ -variant family of the partitions of the strings. The condition is derived from the main filtering criteria for deletion neighborhoods (Theorem 1) and is *weaker* than the original filtering criteria.

We take a different approach to reduce the space requirements. We make use of suffix hashing scheme to reduce the over all size of the index structure whilst maintaining the correctness of the algorithm. The reduction in space is on the cost of admitting *false positives* in the candidate list which are later discarded during the verification step. We further use novel bucketing schemes to reduce unwanted collisions in our hash-table. Our focus during this endeavor has been to reduce the space requirements just enough to fit the index structure in the memory. The next section describes the details of our index structure.

### 4. IMPLEMENTATION

The issue in using *deletion neighborhoods* as a finger-print scheme is the large size of the  $\tau$ -variant family,  $H_\tau(s)$ . In a string  $s$  of length  $l_s$  if no character occurs twice, then  $|H_\tau(s)| = \binom{l}{\tau} + \binom{l}{\tau-1} \dots \binom{l}{0}$ . The size of the  $\tau$ -variant family is still large even if there are character repetitions. For a dic-

tionary containing  $n$  strings ( $s_1$  to  $s_n$ ), if  $G = \bigcup_{i=1}^n H_\tau(s_i)$ , then the total number of inverted index lists required will be  $|G|$  which can be very large. To guarantee that we do not miss any answer  $s_i$ , it is necessary to make an entry of  $i$  ( $id$  of  $s_i$ ) in the inverted index of each  $s' \in H_\tau(s_i)$ . Hence, one cannot omit any string in  $H_\tau(s_i)$ . However, notice that this analysis about large space requirement only applies if we decide to index the string itself, i.e., if an inverted index were to be created for each  $s' \in G$ , and an entry of the  $id$  of  $s_i$  were to be made for each  $s' \in H_\tau(s_i)$ , for each  $s_i$ . If it were possible to hash the strings in  $H_\tau(s_i)$ , intentionally introducing collisions, then one could create an inverted index structure over the hash-keys of the strings in  $H_\tau(s_i)$  instead of the strings themselves. Because of enforced collisions the total number of hash-keys would be much lower than  $|H_\tau(s_i)|$ . This would reduce the space requirement substantially. However, since we would have to hash each  $s_i$  in the dictionary, there would also be collisions across the  $\tau$ -variant families for two different strings in the dictionary i.e. if we had two strings  $s_i$  and  $s_j$  in the dictionary, such that  $H_\tau(s_i) \cap H_\tau(s_j) = \phi$ , then it is possible that for some  $s' \in H_\tau(s_i)$  and  $s'' \in H_\tau(s_j)$ ,  $h(s') = h(s'')$  where  $h$  denotes the hash function. Notice that hashing ensures the completeness of the solution, i.e. there are no *false dismissals*. The proof is the following. Assume the query  $\langle q, \tau \rangle$ . Let  $s \in D$  be an answer of the query. Then by Theorem 1,  $\exists s'$  s.t.  $s' \in H_\tau(q)$  and  $s' \in H_\tau(s)$ . Because  $s' \in H_\tau(q)$ , all entries in the inverted index of  $h(s')$  will be taken as valid candidates. And since  $s' \in H_\tau(s)$ , an entry of  $id_s$  will be made in the inverted index of  $h(s')$ . Thus  $s$  will be present in the candidate list and will be subsequently picked up as an answer post verification.

We propose a system which uses this idea to reduce the space requirement. Further we use bucketing schemes to reduce the number of *unwanted* collisions in the hash-table. We create a dedicated index structure for each value of  $\tau$ . We denote these structures as  $I_\tau$  for  $\tau = \{0, 1, 2, 3, 4\}$ . All queries with edit distance  $\tau$  are handled by  $I_\tau$ . Each  $I_\tau$  has the same basic structure, the main features of which are described below.

## 4.1 Hash Table

We use a simple hashing scheme,  $h$ , where  $h(s)$  is the suffix of the string  $s$ . The size of the suffix,  $L_s$ , is a design parameter. The strings generated by applying the hash function  $h$  on strings in  $H_\tau(s_i)$  form the keys of the hash table. Hereafter we also refer to these keys as hash-strings. Associated with every key in the hash-table is a linked list. Let  $U_{s_i}$  denote the set of *hash-strings* generated by hashing the strings in  $H_\tau(s_i)$ . To index the string  $s_i$ , for each *hash-string*  $w \in U_{s_i}$ , we enter the  $id$  of  $s_i$ , denoted  $id_{s_i}$ , into the linked list associated with  $w$ .

Notice that if the alphabet set is denoted as  $\Sigma$ , then the total number of keys in the hash-table would be upper bounded by  $|\Sigma|^{L_s}$ . While this bound is high for large alphabet sets and high values of  $L_s$ , our experiments reveal that higher values of  $L_s$  can still be practical for large alphabet sets. This is because the different strings in the dictionary will tend to generate many common hash-strings. Thus the expected number of unique keys in the hash-table would be significantly smaller than the maximum number of unique keys possible. We denote  $HK$  as the set of all hash-strings, then figure 1 shows the generic structure of our hash-table.

The keys  $h_1$  to  $h_{|HK|}$  are the unique hash-strings generated by hashing  $H_\tau(s_i) \forall s_i$  in the dictionary  $D$ . Associated with every hash-string  $h_j$  is a list of *ids* of all strings  $s_i \in D$  s.t.  $h_j \in U_{s_i}$ .

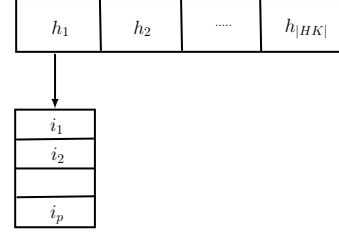


Figure 1: Bucket Overview

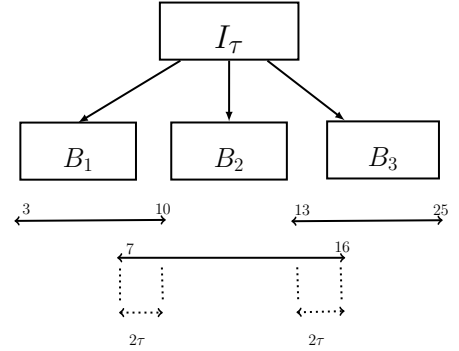


Figure 2: Index structure ( $I_\tau$ ) overview for  $\tau = 2$

## 4.2 Bucketing Scheme

Instead of creating a single hash-table for all the strings in the dictionary, we create  $k$  small hash-tables (hereafter referred to as buckets). Each of the  $k$  buckets is responsible for indexing the strings of length within a specified range. This presented us with the choice to either use overlapping ranges or non-overlapping ranges for adjacent buckets. Overlapping ranges have higher space requirement since dictionary strings in the overlapped regions would have to be indexed in both the buckets. Non-overlapping ranges have the exact space requirement as a single hash-table. However, with non-overlapping ranges, for every query  $\langle q, \tau \rangle$  such that the range  $[l_q - \tau : l_q + \tau]$  spans over two buckets, one would inadvertently have to lookup both the buckets to ensure completeness of the solution. We chose the overlapping strategy with an overlap of  $2\tau$  within adjacent buckets which ensures that every query can be answered by a single bucket alone. Notice that if  $s$  is an answer of the query  $\langle q, \tau \rangle$  then  $|l_s - l_q| \leq \tau$ . This condition effectively describes a length filter which can be used for pruning the candidate set.

The key motivation behind bucketing is to apply the length filter at an early stage. This results in smaller hash-tables and hence speeds up the lookup. The following example emphasizes the power of bucketing scheme. Suppose one were to use a single hash-table along with the suffix size parameter  $L_s = 4$ . Suppose further that we had the string *placating* with *id*  $i$  in the dictionary. Clearly, the inverted index of the string *ting* would contain  $i$ . Now, suppose the query  $\langle \text{bating}, 2 \rangle$  arrives. Hashing the strings present in

the  $\tau$ -variant family of the string *bating* will result in the hash-string *ting*. Thus, *i* will be present in the candidate set. Notice that one can indeed use a length filter at this stage and discard *placating* but this operation would still require some finite computation. And the same would have to be done for any other *ting*-ending string in the dictionary. Instead, if a bucketing scheme were used and say the  $i^{th}$  bucket was responsible for the range [4:8], then only those *ting*-ending strings would have to be examined which have a length  $l_s$  within the range [4:8]. This would rule out *placating* and a number of other strings. Figure 2 shows an overview of the bucketing scheme for  $\tau = 2$ . In Figure 2 we divide the length spectrum of dictionary strings into 3 overlapping ranges. Bucket 1,  $B_1$ , will index every string  $s_i \in D$  s.t.  $1 \leq l_{s_i} \leq 10$ . Bucket 2,  $B_2$ , would index every string  $s_i \in D$  s.t.  $7 \leq l_{s_i} \leq 10$ .  $B_3$  would index  $s_i \in D$  s.t.  $l_{s_i} \geq 13$ . Notice that we allow an overlap of  $2\tau$  between adjacent buckets. This ensures that any query  $\langle q, \tau \rangle$  can be completely answered using a single bucket. In Figure 2  $B_1$  would be responsible for all queries  $\langle q, \tau \rangle$  s.t.  $l_q \leq 8$ .  $B_2$  would handle all queries with length in the range [9:14].  $B_3$  would handle all queries with  $l_q \geq 15$ .

In our implementation, for each  $\tau$  we use different number of buckets. For  $\tau = \{1, 2, 3\}$ , we keep the number of buckets in  $I_\tau$  fixed at 4. For  $\tau = 0$ ,  $I_\tau$  consists of one single bucket. For  $\tau = 4$  we use 3 buckets. The range for each bucket was decided manually after inspecting the length distribution of the query and dictionary strings. However, notice that if the two length distributions are indeed known, smarter automated strategies (employing some empirical measures) can be used to decide both, the number of buckets and the range for each bucket. However, we refrain from doing that in our current implementation.

### 4.3 Length and Threshold Aware Edit Distance Computation

When a query  $\langle q, \tau \rangle$  arrives it is forwarded to the appropriate  $I_\tau$ . It is further routed to the appropriate bucket within  $I_\tau$ . A lookup in the bucket is subsequently done which results in a candidate set which is guaranteed to contain all the answers. We need to check each string in the candidate set and verify if it is an answer. Until now we have only focussed on strategies that help us efficiently construct a candidate set that is also small. However, for high values of  $\tau$  and low values of query length  $l_q$ , the candidate set can still be huge. For ex.  $\langle \text{Manora}, 4 \rangle$  results in a candidate set of size 58000. Therefore there is a clear need for a fast verification method. Notice that exact edit distance computation is not required. For every string in the candidate set we only need to answer if edit distance between the query and the candidate is less than the specified threshold  $\tau$ . We implemented the idea suggested in [5] into our system, which helps us answer this in  $O((\tau + 1) \min(l_q, l_s))$  where  $l_q$  and  $l_s$  denote length of the query and the candidate strings respectively. The idea in [5] is essentially to exploit the length and threshold information to avoid computing all  $l_q \times l_s$  values in the matrix during the computation of the levenshtein distance.

We make some competition specific changes into our system. The first change is the use of multiple threads to process several queries simultaneously. After index construction, our system first reads and classifies all queries on the basis of the edit distance threshold ( $\tau$ ) in each query. It then

---

#### Algorithm 1 Index construction

---

**Input:**  $D, \tau$   
**Output:**  $I_\tau$   
**for**  $s \in D$  **do**  
     $l_s \leftarrow \text{length}(s)$   
     $B \leftarrow \{B_i^\tau : l_s \in \text{range}(B_i^\tau)\}$   
     $U_s \leftarrow h(H_\tau(s))$   
    **for**  $h_j \in U_s$  **do**  
        **for**  $b \in B$  **do**  
             $\text{insert}(id_s, h_j, b)$   
        **end for**  
    **end for**  
**end for**  
**return**  $I_\tau$

---



---

#### Algorithm 2 Query execution

---

**Input:**  $s, \tau$   
**Output:**  $A = \{s : s \in D, ED(s, q) \leq \tau\}$   
     $U_q \leftarrow H_\tau(q)$   
     $A \leftarrow \emptyset$   
     $b \leftarrow B_i^\tau : l_s \in \text{queryRange}(B_i^\tau)$   
    **for**  $h_j \in U_q$  **do**  
         $C_j \leftarrow \text{getList}(h_j, b)$   
    **end for**  
     $C_{id} = \bigcup_j^{U_q} C_j$   
     $C_{string} = \{s : s \in D, id_s \in C_{id}\}$   
    **for**  $s' \in C_{string}$  **do**  
         $d \leftarrow ED(q, s')$   
        **if**  $d \leq \tau$  **then**  
             $A \leftarrow A \cup \{s'\}$   
        **end if**  
    **end for**  
**return**  $A$

---

uses 8 threads to process all queries with  $\tau = 0$ . Thereafter it moves to all queries with  $\tau = 1$  and so on. The second change is using a buffered disk-I/O for producing the result file. By using buffered disk-I/O we have observed a significant improvement in total query execution time as seen in Figure 14. The improvement is significant for the values of  $\tau \geq 2$  as the answer set is usually large for such queries.

## 5. RESULTS

In this section we evaluate the performance of the proposed algorithm. For comparison, we used the state of the art Flamingo Project<sup>1</sup> which is a library that offers fast, efficient implementation of several string similarity search algorithms. We implemented the proposed algorithm in C++ and compiled in g++ 4.7.0 using the -O2 flag. The test environment was an Intel core i7 – 2600 machine with 3.4 GHz CPU and 24 GB RAM running Linux Fedora Core 17 with kernel 3.3.4 – 5.fc17.x86\_64. We perform the comparison only against a  $q$ -gram based approach [3]. We used the experimentation dataset supplied by the organizers of the *String Similarity Workshop, EDBT, 2013*<sup>2</sup>. The dictionary contains names of geographical locations from across

<sup>1</sup><http://flamingo.ics.uci.edu>

<sup>2</sup><http://www2.informatik.hu-berlin.de/~wandelt/searchjoincompetition2013/>

the globe. The dictionary size is 400K and the average string length is 10. The distribution of string length is shown in Figure 7. The query file consists of a total of 5000 queries. Every entry in the query file is essentially a 2 tuple containing the query string and the required edit distance threshold. The range of edit distance threshold  $\tau$  varies from 0 to 4. The exact number of queries for each threshold is shown in Table 2. The average query length is 9.7.

To ensure even comparison, we do not make use of any multi-threading in our implementation i.e. for a given edit distance threshold,  $\tau$  we use only a single thread to process all queries with the threshold  $\tau$ , sequentially. As per the competition specifications, the total time spent from reading the query file to writing the result to a disk is measured as the score. However, writing the query results to the disk can be an obvious bottleneck in the performance of both the algorithms and hence is avoided to ensure a clear comparison.

Before comparing our algorithm with the state of the art methods, we show the time and memory consumption of our algorithm. First, we show the variation in the space requirements of the index structure with changes in the dictionary size  $N$  and the suffix size  $L_s$ . Figure 3 shows the complete size of the hash table as it varies with change in  $N$  for different values of  $\tau$ . The value of  $L_s$  is kept fixed at 5. It must be mentioned here that for every  $\tau$ , shown size is the size of the index structure constructed specifically for that value of  $\tau$ , i.e size of  $I_\tau$ . It can be seen from Figure 3 that the size of the index structure has better than linear dependence on the dictionary size,  $N$ . This reason is that several strings will generate the same hash-key. The following example better illustrates this point. Suppose we have indexed  $n$  strings into the hash table ( $s_1$  to  $s_n$ ) and are now inserting the  $(n+1)^{th}$  string  $s_{n+1}$ . The number of hash-keys in the hash table at this stage will be the total number of unique hash-keys generated by the first  $n$  strings. Let this set of unique hash-keys be denoted as  $HK$ . To index the  $s_{n+1}$ , we would generate  $H_\tau(s_{n+1})$ . If  $n$  is sufficiently high, it will be highly unlikely that  $H_\tau(s_{n+1})$  generates a hash-key which is not already present in  $HK$ . Thus inserting  $s_{n+1}$  will not increase the number of hash-keys in the hash table. The only other cost associated with inserting  $s_{n+1}$  will involve inserting the id of  $s_{n+1}$  for every unique hash-key produced by strings in  $H_\tau(s_{n+1})$ . If the expected length of a string in the dictionary is  $L_{avg}$  then the expected number of unique hash-keys produced by  $H_\tau(s_{n+1})$  will be bounded by  $\binom{L_{avg}}{\tau} + \binom{L_{avg}}{\tau-1} \dots \binom{L_{avg}}{0}$ , which can be approximated as a constant  $c$ , for a fixed  $\tau$ . Thus inserting  $s_{n+1}$  will only require an additional space of  $O(c \times \text{size of id})$ . It is clear that higher values of  $\tau$  will have higher space requirements as is evident from Figure 3 and 4. The reason is that a string  $s$  will generate more number of hash-keys for higher values of  $\tau$  and this will require more number of insertions (an entry in the inverted index of each of the hash-keys generated by  $s$ ).

Figure 4 shows the variation in space requirement with change in suffix size  $L_s$  for the different values of  $\tau$ . The dictionary size ( $N$ ) is fixed to 400K. Increasing the suffix size  $L_s$ , as expected, dramatically increases the space requirement. This is because  $L_s$  is essentially the size of the hash-key and increasing  $L_s$  increases the size of the hash string. The effect is more pronounced for higher values of  $\tau$  (see Figure 4) because as mentioned previously higher  $\tau$

**Table 2: Number of queries and optimal  $L_s$  value**

$\tau$	number of queries	optimal $L_s$ value
0	1673	8
1	854	8
2	827	7
3	839	7
4	807	6

implies more number of hash-keys per string.

We show the dependence of index construction time on dictionary size ( $N$ ) and suffix size ( $L_s$ ) in Figure 5 and 6. In Figure 5 we keep the suffix size  $L_s$  fixed at 5 and vary  $N$  from 100K to 400K for  $\tau = \{0, 1, 2, 3, 4\}$ . As mentioned previously, the reported time for each  $\tau$  is the time taken to build an index specifically for that value of  $\tau$ . It can be seen that the index construction time varies linearly with increase in the dictionary size  $N$ . In Figure 6, we keep the dictionary size fixed at 400K while varying  $L_s$  from 4 to 8. The index construction time increases rapidly with increase in suffix size  $L_s$ . This is expected because as stated previously, higher values of  $L_s$  imply that for a string  $s$ , more number of hash-keys will be generated by strings in  $H_\tau(s)$ .

We next compare the average query execution achieved by the two algorithms i.e. the proposed implementation and the Flamingo Library. Figure 8 shows the average query time achieved by the Flamingo Library implementation and our own algorithm for different values of the edit distance threshold ( $\tau$ ). Figure 8 clearly shows that our algorithm is an order of magnitude faster than the Flamingo Library, one of the current state of the art methods. Thus our claim that deletion neighborhoods are more selective, when compared to  $q$ -gram based methods, and are scalable stands verified.

It must be mentioned here that the value of  $L_s$  is not the same for each  $\tau$ . For each  $\tau$ , we select the optimal value of  $L_s$  within the range [4:8] by performing exhaustive experimentation. The results are shown in Figure 9 through 13. The optimal values obtained are mentioned in Table 2. We admit that the probed range is rather small, but since the space requirement increases rapidly with increase in  $L_s$ , this puts a serious restriction on the values of  $L_s$  that can be used. Keeping in the mind the memory constraint (48 GB) as specified in the competition, we only considered those values of  $L_s$  which seemed plausible for the competition.

In Figures 9 through 13 we show the variation in average query time with change in the suffix size parameter  $L_s$ . Increasing  $L_s$  would decrease collisions between different strings in the dictionary i.e. in Figure 1, the average length of the list corresponding to each  $h_j$  would decrease. However, the total number of hash-keys generated by  $H_\tau(q)$  would also increase. Hence more lists would need to be merged. Hence, changing the  $L_s$  value is essentially a trade-off between merging fewer lists which are not very selective, or more number of lists which are selective. For all values of  $\tau = \{0, 1, 2, 3, 4\}$ , the behavior is nearly similar. Increasing the  $\tau$  value initially provides significant reduction in the average query time. However, it quickly assumes a minima and increasing  $L_s$  beyond the minima slowly increases the average query time.

In Figure 14, we present the comparison between the total time taken by the two disk writing methods i.e. standard disk I/O and buffered disk I/O. In standard I/O, we write

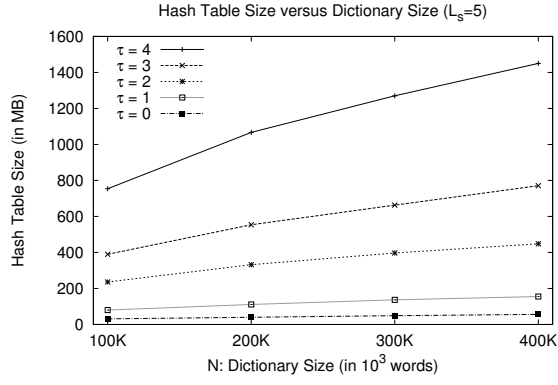


Figure 3: Hash table size versus dictionary size

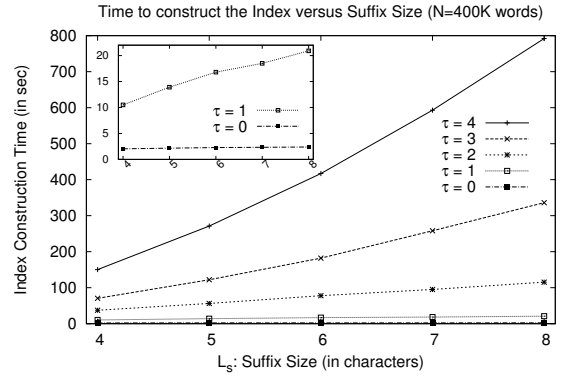


Figure 6: Index construction time versus suffix size

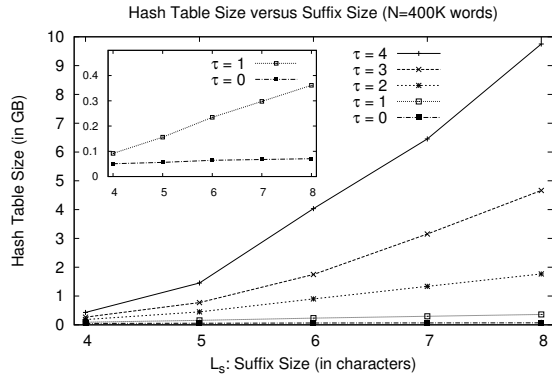


Figure 4: Hash table size versus suffix size

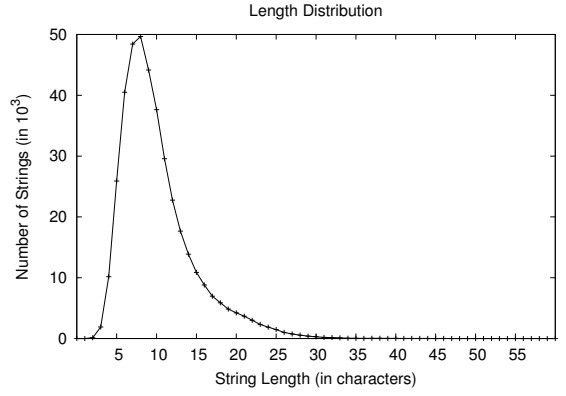


Figure 7: Length distribution of dictionary strings

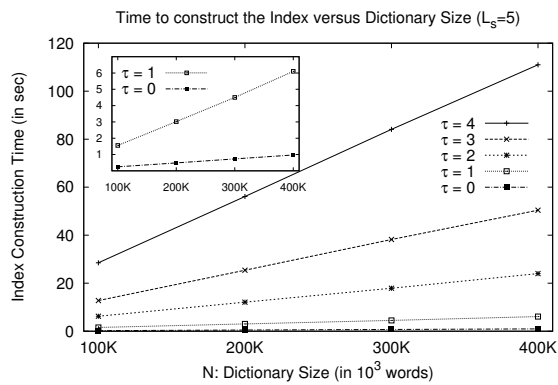


Figure 5: Index construction time versus dictionary size

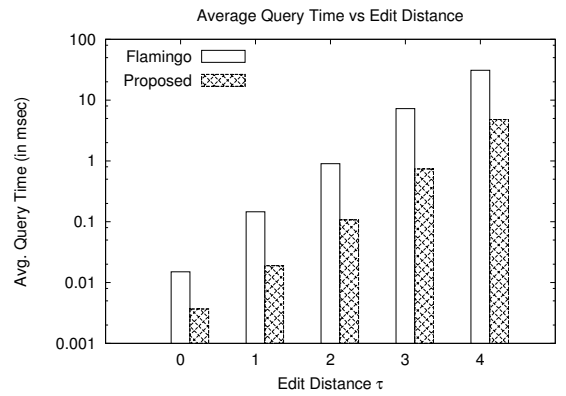
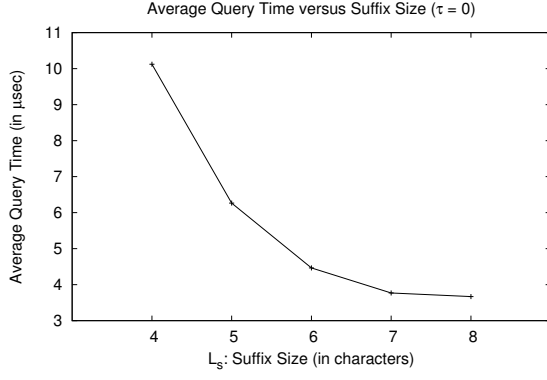
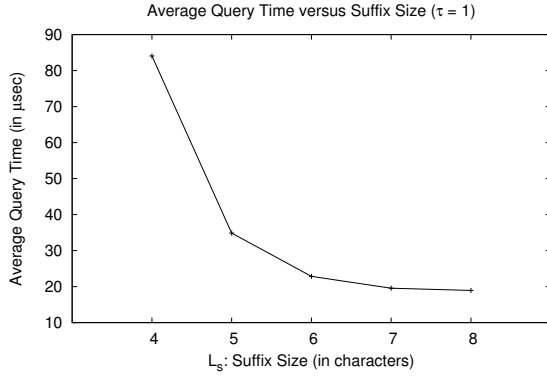


Figure 8: Average query time for different values of  $\tau$  at optimal  $L_s$  and  $N = 400K$



**Figure 9:** Average query time for different values of  $L_s$  and  $\tau = 0$ ,  $N = 400K$

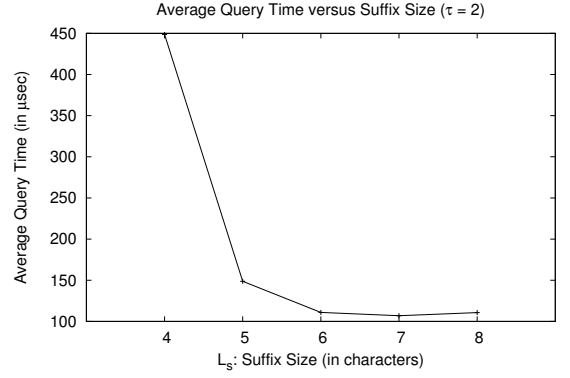


**Figure 10:** Average query time for different values of  $L_s$  and  $\tau = 1$ ,  $N = 400K$

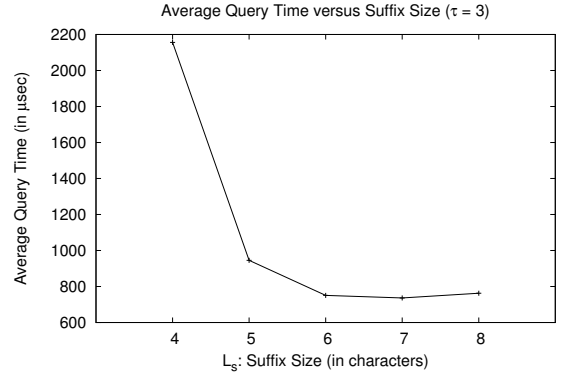
each answer of each query on a separate line, as soon as it is confirmed. In the latter, the same is first written to a buffer in the memory. After all the queries are processed, the buffer is then traversed and the result is written onto the disk in a single operation. Figure 14 shows the percentage reduction in the total time (query computation + disk i/o) when using the buffered approach as compared to the standard approach. As expected, using buffered I/O is significantly faster for higher values of  $\tau$ , particularly because higher  $\tau$  implies a bigger answer set. In Figure 15 we show the total time spent on writing the results to the disk as a fraction (in %) of the total time spent in the query. Notice that the time spent on file I/O is very small (less than 0.55%) of the total time spent on the query.

## 6. FUTURE WORK

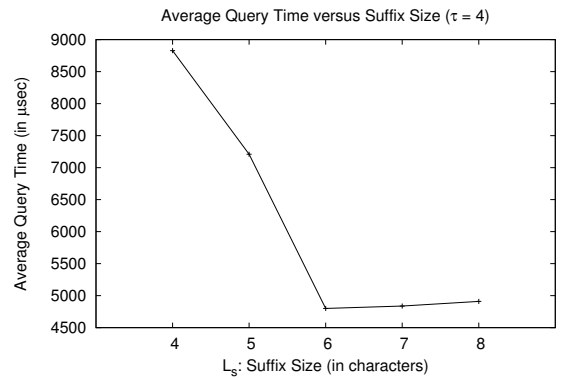
There are many possible directions of future work. While in the paper, we address the problem of string similarity search laying stress on reducing the running time with an abuse to memory, steps can be taken to make this solution more memory efficient. The index structure could be made more compact in a number of ways. At a finer level of granularity, the idea of bucketing could be modified incorporating the use of non-overlapping buckets instead of the overlapping ones. Whereas at a broader level the hashing scheme could



**Figure 11:** Average query time for different values of  $L_s$  and  $\tau = 2$ ,  $N = 400K$

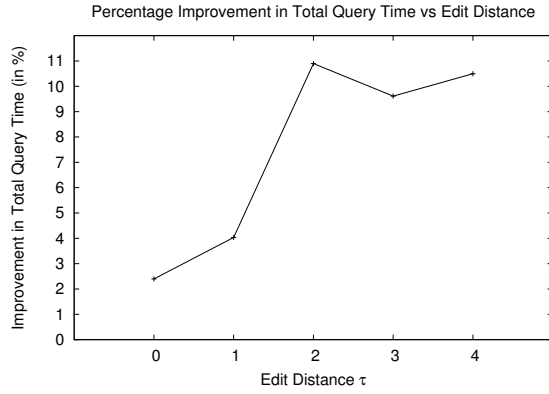


**Figure 12:** Average query time for different values of  $L_s$  and  $\tau = 3$ ,  $N = 400K$

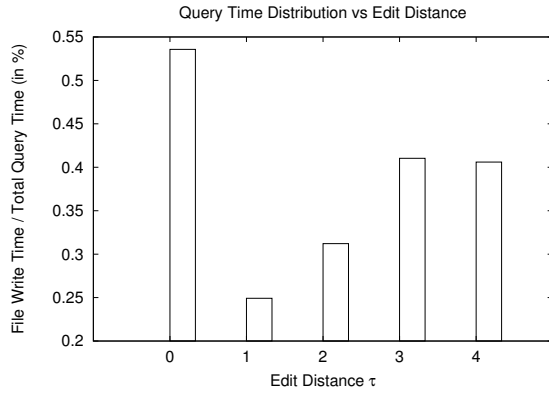


**Figure 13:** Average query time for different values of  $L_s$  and  $\tau = 4$ ,  $N = 400K$





**Figure 14: Standard versus buffered I/O: Total query time for  $L_s = 5$ ,  $N = 400K$  and different values of  $\tau$**



**Figure 15: Percentage time spent on file I/O: File Write Time / Total query time (in %) for  $L_s = 5$ ,  $N = 400K$  and different values of  $\tau$**

be altered, by using a single hash table constructed only for the  $\tau_{max}$  and maintaining pointers along with each  $\tau$  variant as to which value of  $\tau$  generated it, instead of separate hash tables for each value of  $\tau$ . The above two modifications will guarantee reduction in memory requirements, whereas the running times might change, which needs to be further investigated by rigorous experimentation. The edit distance computation can also be made more efficient as explained as follows. Given two strings, if we are able to get hold of any common substring, then this information can be leveraged to compute edit distance by parts which is more efficient as shown in [5]. It is not hard to see that deletion neighborhoods contain the above mentioned information inherently, which could be obtained in constant time while constructing the  $\tau$  variant family.

One other improvement can be made by selecting hash functions from 2-Universal Hash Family and hashing the complete strings. By varying size and number of hash tables we could eliminate majority of non-answers in the candidate set.

## 7. CONCLUSION

We present in this paper the details of a system developed at the *Special Interest Group in Data (SIGDATA)*, *Indian Institute of Technology, Kanpur* for the *String Similarity Search Competition, EDBT, 2013*. Our system emphasizes the power of deletion-neighborhoods over other finger-print schemes. We propose the use of novel hashing schemes to reduce the space requirement of the index structure. We also propose the use of bucketing schemes to reduce collisions in the aforementioned hashing scheme. The proposed method achieves significant reduction in the average query execution time when compared against a previous state of the art method.

## 8. REFERENCES

- [1] T. Bocek, E. Hunt, and B. Stiller. Fast similarity search in large dictionaries. 2007.
- [2] D. Deng, G. Li, and J. Feng. An efficient trie-based method for approximate entity extraction with edit-distance constraints. In *ICDE*, pages 762–773, 2012.
- [3] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.
- [4] G. Li, D. Deng, and J. Feng. Faerie: efficient filtering algorithms for approximate dictionary-based entity extraction. In *SIGMOD Conference*, pages 529–540, 2011.
- [5] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.
- [6] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6(1):132–137, 1985.
- [7] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD Conference*, pages 85–96, 2012.
- [8] W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit distance constraints. In *SIGMOD Conference*, pages 759–770, 2009.
- [9] 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.