Fast Algorithms for Top-k Approximate String Matching

Zhenglu Yang [#1] **Jianjun Yu** [*2] **Masaru Kitsuregawa** [#3]
[#] Institute of Industrial Science, The University of Tokyo, Japan
[1 3]{yangzl, kitsure}@tkl.iis.u-tokyo.ac.jp
[*]Computer Network Information Center,
Chinese Academy of Sciences, Beijing, China
[2]yujj@cnic.ac.cn

Abstract

Top-k approximate querying on string collections is an important data analysis tool for many applications, and it has been exhaustively studied. However, the scale of the problem has increased dramatically because of the prevalence of the Web. In this paper, we aim to explore the efficient top-k similar string matching problem. Several efficient strategies are introduced, such as length aware and adaptive q-gram selection. We present a general q-gram based framework and propose two efficient algorithms based on the strategies introduced. Our techniques are experimentally evaluated on three real data sets and show a superior performance.

Introduction

Similar string searching is an important problem because it involves many applications, such as query suggestion in search engines, spell checking, similar DNA searching in large databases, and so forth. From a given collection of strings, such queries ask for strings that are similar to a given string, or those from another collection of strings (Li, Wang, and Yang 2007).

While the applications based on similar string querying are not new, and there are numerous works supporting these queries efficiently, such as (Arasu, Ganti, and Kaushik 2006; Li, Wang, and Yang 2007; Gravano et al. 2001; Ukkonen 1992), the scale of the problem has dramatically increased because of the prevalence of the Web (Bayardo, Ma, and Srikant 2007).

To measure the similarity between two strings, different metrics have been proposed (Levenshtein 1966; Rijsbergen 1979; Cohen 1998). In this paper, we focus on similarity search with edit distance thresholds. We will discuss the extension of our strategies to other metrics at the end of the paper. Many algorithms have used q-gram based strategies (Ukkonen 1992; Navarro and Baeza-Yates 1998; Navarro 2001) to measure the edit distance between two strings. A q-gram is a consecutive substring of a string with size q that can be used as a signature of the string. To hasten the search process, many approaches pre-construct some index structures (e.g., suffix tree (Ukkonen 1993)) and then search these compact indices in an efficient way. However, most of the works are rooted in a threshold-based framework (i.e., error threshold predefined), and there are not many works on exploring the top-k issue of the problem.

In this paper, we propose two efficient algorithms to tackle the issue of top-k similar string searching. It is challenging because of the main issue:

- **Time consuming frequency counting**: The size of the whole inverted lists corresponding to the query string may be large. Although the state-of-the-art techniques (i.e., divide-merge-skip (Sarawagi and Kirpal 2004; Li, Lu, and Lu 2008)) are applied, counting the frequency of q-grams is still time consuming.

In this paper, we propose efficient strategies to address the issue. Specifically, we introduce a string length-aware technique and propose to switch the q-gram dictionary. For the second strategy, we construct a set of q-gram dictionaries in the preprocessing. In each iteration, we select an appropriate value of q (hence, a q-gram dictionary is chosen) based on the top-k similarity score of the last iteration. The new value of q is guaranteed to increase, which indicates that the size of the inverted lists is decreasing (i.e., large q leads to shorter inverted lists). Hence, the cost of the frequency counting is reduced.

Our contributions in this paper are as follows:

- We introduce several efficient strategies for top-k approximate string search. The count filtering and length-aware mechanism are adapted to tackle the top-k similar search issue, while the adaptive q-gram selection is employed to improve further the performance of frequency counting.
- Based on these strategies, we propose two efficient algorithms in a general q-gram based framework. Several state-of-the-art techniques are applied (i.e., divide-merge-skip (Sarawagi and Kirpal 2004; Li, Lu, and Lu 2008)).
- We conduct comprehensive experiments on three real data sets, and evaluate the querying efficiency of the proposed approaches in terms of several aspects, such as the dataset size, value of k, and q-gram dictionary selection. Our experimental results show that the proposed algorithms exhibit a superior performance.

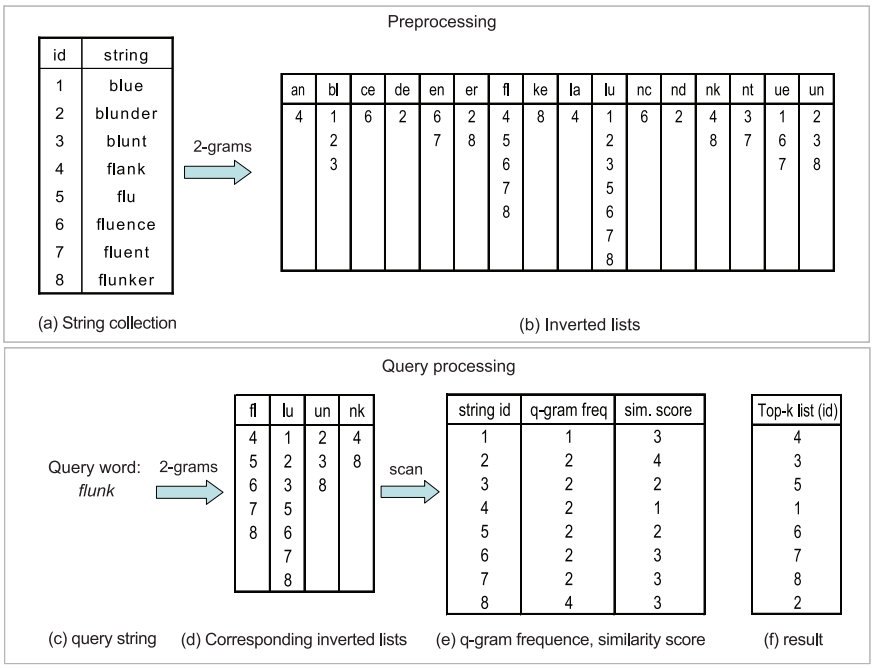


Figure 1: A general framework for approximate top-k string search with q-gram

Problem Statement and Analysis

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

Q-Grams: Given a string s and a positive integer q, a positional q-gram of s is a pair (i, g), where g is the q-gram of s starting at the i-th character, that is $g = s[i, i + q - 1]$. The set of positional q-grams of s, denoted by $G(s, q)$, is obtained by sliding a window of length q over the characters of string s. There are $|s| - q + 1$ positional q-grams in $G(s, q)$. For instance, suppose q = 3, and s = **university**, then $G(s, q)$ = {(1,**uni**), (2,**niv**), (3,**ive**), (4,**ver**), (5,**ers**), (6,**rsi**), (7,**sit**), (8,**ity**)}. We use a sliding window of size q on the new string to generate positional q-grams. For simplicity, in our notations we omit positional information, which is assumed implicitly to be attached to each gram.

Top-k Approximate String Queries: The edit distance (also known as Levenshtein distance) between two strings s_1 and s_2 is the minimum number of edit operations of single characters needed to transform s_1 to s_2. Edit operations include insertion, deletion, and substitution. We denote the edit distance between s_1 and s_2 as $ed(s_1, s_2)$. For example, ed("blank", "blunt") = 2. In this paper, we consider the top-k approximate string query on a given collection of strings S. Formally, for a query string Q, finding a set of k strings R in S most similar to Q, that is, $\forall r \in R$ and $\forall s \in (S - R)$ will yield $ed(Q, r) \leq ed(Q, s)$.

To extract the top-k similar strings efficiently, we apply the q-gram strategy (Ukkonen 1992) with deliberate optimization, as will be discussed shortly. The q-gram similarity of two strings is the number of q-grams shared by the strings, which is based on the following lemma.

Lemma 1 (q-gram lemma (Jokinen and Ukkonen 1991)). *Let P and S be strings with the edit distance τ. Then, the q-gram similarity of P and S is at least*

$$t = max(|P|, |S|) - q + 1 - q \cdot \tau \quad (1)$$

A General Framework for Top-k Similar String Query Processing

We first introduce the general q-gram based top-k querying system in this section. In the next sections, we will propose several efficient strategies and introduce our optimal algorithms for the top-k similar string search . The main issue is that the cost of counting the frequency of the q-grams is high, and thus efficient technique is preferred.

Algorithm 1 shows the pseudo code of the general framework, which is based on the traditional threshold-based framework (i.e., presetting a similarity threshold) (Jokinen and Ukkonen 1991). An example illustrating the details of the framework is shown in Fig. 1, which presents the preprocessing and the query processing.

Example 1. Suppose a user wants to obtain the top-1 similar word to $flunk$. While querying, the query word $flunk$ is parsed into four 2-grams as fl, lu, un, and nk; their corresponding inverted lists are $\{4, 5, 6, 7, 8\}$, $\{1, 2, 3, 5, 6, 7, 8\}$, $\{2, 3, 8\}$, and $\{4, 8\}$ respectively. The lists are merged to count the frequency of the strings in the collection resulting in 1:1, 2:2, 3:2, 4:2, 5:2, 6:2, 7:2, and 8:4, where each pair of values is of type sid:freq. Sid and freq represent

the string ID and the frequency value of the n-grams included by the string, respectively. For instance, the string $blue$ (whose ID is 1) includes 1 n-gram of the queried word, giving us $1:1$. Finally, the edit distances between the candidate strings and the query word are calculated as $blue$:3, $blunder$:4, $blunt$:2, $flank$:1, flu:2, $fluence$:3, $fluent$:2, and $flunker$:2, where each pair of values is of type string:distance. The top-1 similar string to the query word $flunk$ is thus $flank$.

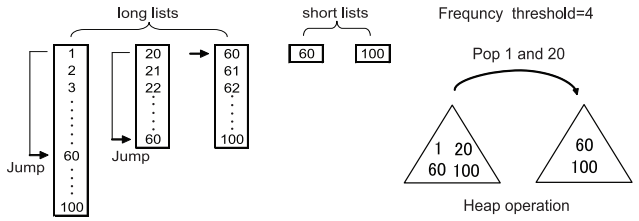
As the threshold-based similar string search is thoroughly studied, and the algorithm is self-explanatory with the help of the example, we did not include the details anymore.

Algorithm 1: Similar String Search by Q-gram Matching

Input: String collections SC, query string s, top-k, q
Output: Top-k similar strings

```
1  construct q-gram dictionary as trie t, based on SC;
2  parse the query string s into a candidate q-gram list CL;
3  set q-gram distance t=1; //to include all cand. strs;
4  foreach x ∈ CL do
5      find the string list SL of x in the trie t;
6      foreach y ∈ SL do
7          frequency[y]++;
8          if frequency[y] ≥ t then
9              put y into a candidate result list CRL;
10 foreach z ∈ CRL do
11     compute the similarity ed(z, s);
12 Output the top-k strings with smallest similarities;
```

Optimization of Frequency Counting. There are many techniques in the literature proposed to improve the performance of q-gram based query processing. The most time consuming step in the query processing is to count the frequency of q-grams, which has to scan a large amount of inverted lists. Recently, (Sarawagi and Kirpal 2004; Li, Lu, and Lu 2008) have proposed efficient strategies (i.e., divide-merge-skip) to count the frequency of q-grams. Fig. 2 illustrates the basic idea for efficiently scanning the inverted lists. It first divides the inverted lists into short lists and long lists. The whole short lists are then scanned, and the elements are collected. Finally, it binary searches these elements in the long lists to find strings whose frequencies are larger than the threshold. For example, suppose we aim to find strings whose frequencies are larger than the threshold t (i.e., 4) in the inverted lists in Fig. 2(a). The five inverted lists are deliberately divided into three long lists and two short ones, that is, the number of long lists is t-1 (Sarawagi and Kirpal 2004). This way, only those strings that exist in the short lists have the chance to pass the threshold test (i.e., their frequencies are equal to or larger than t). We scan the two short lists to obtain the candidate strings whose IDs are 20 and 60. These candidate strings are then used to for binary searching the long lists, as shown in Fig. 2(a). Many strings can be skipped in the long lists, and thus the efficiency is improved. Heap operation is applied to hasten the process. Refer to (Sarawagi and Kirpal 2004; Li, Lu, and Lu 2008) for details.



(a) Divide inverted lists into short and long lists, scan short lists and binary search long lists

(b) Heap operation

Figure 2: Divide-merge-skip strategy

Top-k similar search algorithms

In this section, we propose two efficient algorithms for top-k approximate string search. We first introduce two filtering strategies (i.e., count filtering and length filtering) for the first algorithm and then we present the adaptive q selection strategy for the second algorithm.

Count Filtering

The intuition of count filtering is that strings within a small edit distance of each other share a large number of q-grams in common. The q-gram count filtering for the top-k similar string search is derived from that of the traditional threshold-based framework (Ukkonen 1992). The difference is that the frequency threshold is dynamically updated for the former issue, while it keeps static for the later issue. Based on Eq. 1, we have the following.

Lemma 2 (q-gram lemma for top-k approximate string search). *Let P be the query string and S be the top-k similar string in the candidate string list so far. P and S have the edit distance τ'_{top-k}. Then, the q-gram similarity t'_{top-k} of P and S is at least*

$$t'_{top-k} = max(|P|, |S|) - q + 1 - q \cdot \tau'_{top-k} \qquad (2)$$

The count filtering for top-k query is a general optimization technique. As will be mentioned shortly, in our proposed algorithms, count filtering is implemented in different cases by combining it with the length filtering and the adaptive q-gram strategy, respectively.

Length Filtering

The intuition of length filtering is that if there are two strings within a small edit distance τ of each other, then the difference of their lengths will not exceed τ.

When scanning the inverted lists, we choose to test the candidate strings in ascending order of the difference between their lengths and that of the query string. The process can be early terminated based on the following lemma.

Lemma 3 (Early terminate). *Let P be the query string and S be the top-k candidate similar string so far. We denote the set R of the remaining untested strings. If $\forall r \in R, ed(P, S) \leq ||P| - |r|| + 1$, then the search process can be early terminated.*

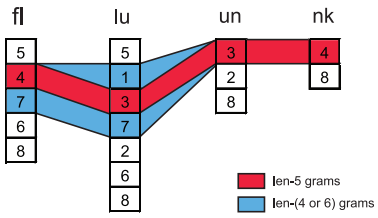


Figure 3: Illustration of the length filtering.

The work in (Hadjieleftheriou et al. 2008; Gravano et al. 2001) also proposed a similar idea on using the length bounding property. Nevertheless, their works require a user-specified threshold, τ, to indicate the lower bound of the distance between the similar strings. This parameter is difficult for users to set because many answers may be returned if the value is too large and there may be no answers if the value is too small. The difference is due to the different purposes of the two kinds of works: (Hadjieleftheriou et al. 2008; Gravano et al. 2001) aims to tackle the traditional threshold-based similar string join (or search), while this paper addresses the top-k issue. Similar to the count filtering, the length bound needs to be dynamically updated.

Branch and Bound Algorithm

In this section, we propose an algorithm that follows the branch and bound manner (BB) based on count filtering and length filtering. For simplicity and without loss of generality, we assume that a specific q-gram set is constructed in the preprocessing. The query processing is executed as follows:

- Initialization: Set the frequency threshold t'_{top-k}=1, and the length difference ld=0. P is parsed into a set of q-grams. The following steps are executed iteratively until k similar strings are output.
- Branch: Extract the corresponding inverted lists in which any string S has $||S| - |P|| = ld$.[1] Scan these inverted lists and discover the candidate similar strings whose frequencies are no less than t'_{top-k}.[2]
- Bound: Rank the candidate strings and obtain the temporal edit distance threshold τ'_{top-k}. Terminate the process if the top-k string Q in the candidate list so far has $ed(P,Q) \leq (ld+1)$. Compute the temporal frequency threshold t'_{top-k} based on τ'_{top-k} (Lemma 2) and set $ld = ld + 1$.

Example 2. We present an example to illustrate the process. The same setting is given as in Example 1. As the length of the query string $flunk$ is 5, instead of scanning all the strings in the inverted lists of the corresponding grams (i.e., as done in Example 1), we first only scan strings whose lengths are equal to 5. As illustrated in Fig. 3, the results are two candidate strings: $blunt$ (whose ID is 3) and $flank$ (whose ID is 4). We then compute their edit distance to the query word. We obtain the top-1 similar word $flank$ and the edit distance $ed(flunk, flank)$=1. As $ed(flunk, flank) \leq ld + 1$, we can output $flank$ as the top-1 similar string immediately.

In case we want to extract other top-k similar strings (where k >1), we can recursively execute the process. As illustrated in Fig. 3, the next iteration needs to scan the strings whose lengths are equal to 4 or 6. The pseudo code is shown in Algorithm 2.

Algorithm 2: Branch and bound algorithm (BB)

Input: String collections S, a query string P, k
Output: Top-k similar strings

```
1  build index IDX to record strings based on their
   lengths from S;
2  construct q-gram dictionary T;
3  parse P into a set of q-grams, t'_{top-k}=1, ld=0;
4  while true do
5      Inv = the inverted lists in T with any string S has
       ||S| - |P|| = ld;
6      apply skip-merge-skip strategy to get the candidate
       strings from Inv;
7      rank the candidate strings, Q=the top-k string;
8      τ'_{top-k}=ed(P, Q);
9      if ed(P, Q) ≤ (ld + 1) then
10         Output the top-k strings; break;
11     compute t'_{top-k} based on τ'_{top-k};
12     ld=ld+1;
13 End
```

Adaptive q-gram selection

We assume from above that the value of q is static, which indicates that we only use one set of q-gram inverted lists in the entire process. However, as well known in the literature, a larger value of q results in a smaller size of inverted lists, which may reduce the cost of the frequency counting. We propose that an appropriate q-gram dictionary (may be varying) be chosen in each iteration, which is calculated by the top-k similarity score in the last iteration. As illustrated in the experiments, this strategy can reduce the size of the inverted lists to be scanned, therefore improving the query performance.

Lemma 4 (Adaptive q selection). *Let P be the query string and S be the top-k similar string in the candidate string list so far. P and S have the edit distance τ'_{top-k}. Then, the adaptive value of integer q, q'_{top-k}, is selected in the following formula.*

$$q'_{top-k} = \begin{cases} \frac{max(|P|,|S|)}{\tau'_{top-k}+1} & , \quad if \frac{max(|P|,|S|)}{\tau'_{top-k}+1} \geq 2 \\ 2 & , \quad else \end{cases} \tag{3}$$

Note that the formula is derived from Eq. 2 with the frequency threshold t=1. The reason is that we should ensure that no similar strings are missing (i.e., they should exist in the extracted inverted lists with $t \leq$1) and also guarantee a large value of q is selected (i.e., smaller t is preferred). As

[1] We pre-build an index to record strings based on their lengths.
[2] Divide-merge-skip strategy is applied as aforementioned.

the search processes, τ'_{top-k} decreases and $max(|P|, |S|)$ increases (i.e., by combining the length filtering strategy). Therefore, q'_{top-k} is guaranteed to be increasing, which may further reduce the size of the inverted lists.

After selecting the new value of q, the new threshold frequency t'_{top-k} is recomputed based on Eq. 2. Note that t'_{top-k} may be larger than 1.

Adaptive q-gram Algorithm

The adaptive q-gram algorithm (AQ) is constructed based on the three strategies introduced above (i.e., count filtering, length filtering, and adaptive q-gram selection). The pseudo code is shown in Algorithm 3. The difference from the BB algorithm exists in lines [2-3, 6-7, 15-16]. As will be illustrated in the experiments, the AQ algorithm with adaptive q selection strategy can improve the whole performance of the query processing. In real implementation, we cannot build all possible q-gram dictionaries due to limited resources. Moreover, from the experiments, we discover that it is not always performance optimal to choose a too large value of q.

Algorithm 3: Adaptive q-gram Algorithm (AQ)

Input: String collections S, a query string P, k
Output: Top-k similar strings

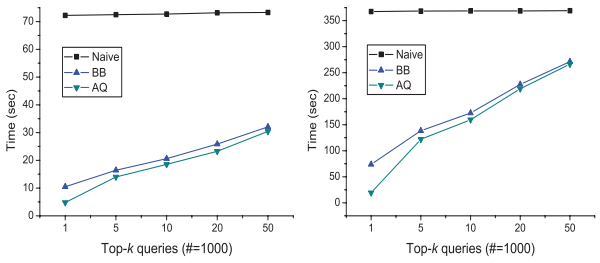
```
1  build index IDX to record strings w.r.t length from S;
2  construct a set of q-gram dictionaries where
   q ∈ [q_min, q_max];
3  q'_{top-k}=q'_min;
4  t'_{top-k}=1, ld=0;
5  while not k strings are output do
6      choose q'_{top-k}-gram dictionary T';
7      parse P into a set of q'_{top-k}-grams;
8      Inv = the inverted lists in T' with any string S has
       ||S| − |P|| = ld;
9      apply skip-merge-skip strategy to get the candidate
       strings from Inv;
10     rank the candidate strings, Q=the top-k string;
11     τ'_{top-k}=ed(P, Q);
12     if ed(P, Q) ≤ (ld + 1) then
13         Output the top-k strings; break;
14     ld=ld+1;
15     compute q'_{top-k} based on Eq. 3;
16     compute t'_{top-k} based on q'_{top-k} and τ'_{top-k};
17 End
```

Performance Analysis

To evaluate the efficiency of our strategies, we conducted extensive experiments. We performed the experiments using a Intel(R) Core(TM) 2 Dual CPU PC (3GHz) with a 2G memory, running Redhat linux. The $Naive$ algorithm was implemented based on the general top-k querying framework. All the algorithms were written in C++. The default value of q is set to 2 for $Naive$ and BB, while for AQ algorithm the default dictionaries are [2,3]-grams. We conducted experiments on three real life data sets.

1. $Dict$. It was downloaded from the Web site of GNU Aspell project[3], which is an English dictionary. It included 103K words, with an average length of 8.
2. $Person$. It was downloaded from the Web site of the Texas Real Estate Commission[4]. The file included a list of records of person names, companies, and addresses. We used about 151K person names, with an average length of 33.
3. $DBLP$. It was from the DBLP Bibliography. It included 308K titles, with an average string length of 51.

Efficiency of Query Processing



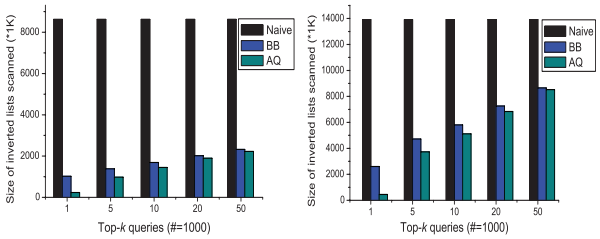
(a) **D**ict dataset (b) **P**erson dataset

Figure 4: Query performance of different algorithms

In this section, we evaluate the query answering performance of our proposed algorithms when varying the value of k. To test the effect of top-k query, we randomly selected 1000 different queries from the data sets.

The result is shown in Fig. 4, from where we can see that our algorithms always perform better than $Naive$, especially when k is small. This is not surprising because of the dynamic length filtering and count filtering strategies employed. Between our two algorithms, AQ is superior to BB because the former can reduce the cost by adaptively selecting a relative large value of q.

Size of inverted lists scanned



(a) **D**ict dataset (b) **P**erson dataset

Figure 5: Inverted lists scanned among different algorithms

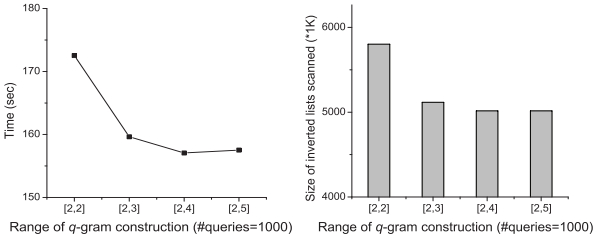
In this section, we evaluate the size of inverted lists scanned in different algorithms when varying the value of k. As illustrated in Fig. 5, we can see that BB and AQ

[3] www.aspell.net/

[4] www.trec.state.tx.us/LicenseeDataDownloads/trecfile.txt

scan smaller sets of inverted lists compared with the other algorithm. This is the intrinsic reason why our algorithms performed better on query processing (as shown in Fig. 4).

Effect of q_{max}

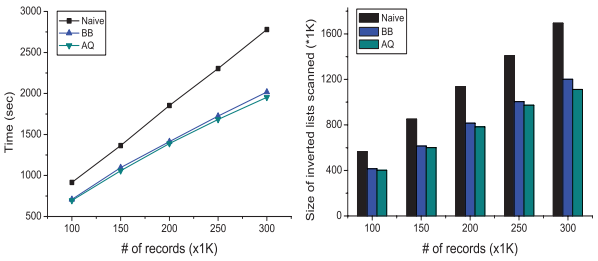


(a) **P**erson dataset (b) **P**erson dataset

Figure 6: Effect of q_{max}

We evaluate the effect of q_{max} in the AQ algorithm. Fig. 6(a) illustrates the query performance of AQ when querying the top-10 similar strings on the $Person$ dataset. We can see that the adaptive q-gram selection strategy improved the overall performance, when a relative large range of q-grams was selected (i.e., [2,4]). The reason why the performance did not improve on larger value of q_{max}, is due to the failure of selecting these q-grams during query processing. Fig. 6(b) illustrates the phenomenon.

Effect of data set



(a) **D**BLP dataset (b) **D**BLP dataset

Figure 7: Query performance versus data size

We evaluate the query performance on different sizes of the $DBLP$ data set[5], by querying top-10 similar strings. As illustrated in Fig. 7(a), we can see that our proposed algorithms performed better when the data size becomes larger. The reason is the same, that many candidate strings in the inverted lists have not been scanned in BB and AQ.

Conclusions

In this paper we have studied the efficient top-k approximate string searching problem. Several efficient strategies are proposed, i.e., length filtering and adaptive q-gram selection. We present two algorithms in a general framework by combining these strategies. Extensive experiments are conducted on three real data sets, the results show that our approaches can efficiently answer the top-k string queries.

[5]To generate different data sets, we follow the same technique as be used in (Li, Wang, and Yang 2007)

We mention a recent work (Vernica and Li 2009) which also proposed to tackle the top-k similar search issue. The authors defined the similarity between two strings as a linear combination of the Jaccard similarity (or edit distance) and the weight of the strings, which is different from our definition which only concerns edit distance. Both of the works apply the divide-merge-skip and the dynamic count filtering techques yet the difference is that we introduce the adaptive length-aware and q-gram selection strategies. Further experimental comparison with their work will be conducted in the future version of this paper, especially on comparing the 2PH algorithm with an optimal tight similarity threshold.

In addition, we will evaluate some other metrics, such as dice coefficient, cosine metric, and so forth. We believe our strategies are general enough, since the length range for a specific threshold exists for all these metrics (Li, Lu, and Lu 2008), and the adaption of the q-gram selection technique.

References

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

Bayardo, R. J.; Ma, Y.; and Srikant, R. 2007. Scaling up all pairs similarity search. In *WWW*, 131–140.

Cohen, W. W. 1998. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *SIGMOD*, 201–212.

Gravano, L.; Ipeirotis, P. G.; Jagadish, H. V.; Koudas, N.; Muthukrishnan, S.; and Srivastava, D. 2001. Approximate string joins in a database (almost) for free. In *VLDB*, 491–500.

Hadjieleftheriou, M.; Chandel, A.; Koudas, N.; and Srivastava, D. 2008. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, 267–276.

Jokinen, P., and Ukkonen, E. 1991. Two algorithms for approximate string matching in static texts. In *In Proc. 2nd Ann. Symp. on Mathematical Foundations of Computer Science*, 240–248.

Levenshtein, V. I. 1966. Binary codes capable of correcting deletions, insertions, and reversals. Technical Report 8, Soviet Physics Doklady.

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

Li, C.; Wang, B.; and Yang, X. 2007. Vgram: improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, 303–314.

Navarro, G., and Baeza-Yates, R. A. 1998. A practical q -gram index for text retrieval allowing errors. *CLEI* 1(2).

Navarro, G. 2001. A guided tour to approximate string matching. *ACM Computing Survey* 33(1):31–88.

Rijsbergen, C. J. 1979. Information retrieval. *Butterworth-Heinemann*.

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

Ukkonen, E. 1992. Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science* 92(1):191–211.

Ukkonen, E. 1993. Approximate string-matching over suffix trees. In *CPM*, 228–242.

Vernica, R., and Li, C. 2009. Efficient top-k algorithms for fuzzy search in string collections. In *SIGMOD Workshop on Keyword Search on Structured Data (KEYS)*, 9–14.