Efficient Merging and Filtering Algorithms for Approximate String Searches

Chen Li, Jiaheng Lu, Yiming Lu

Department of Computer Science, University of California, Irvine, CA 92697, USA

chenli@ics.uci.edu, {jiahengl,yimingl}@uci.edu

***Abstract*— We study the following problem: how to efficiently find in a collection of strings those similar to a given query string? Various similarity functions can be used, such as edit distance, Jaccard similarity, and cosine similarity. This problem is of great interests to a variety of applications that need a high real-time performance, such as data cleaning, query relaxation, and spellchecking. Several algorithms have been proposed based on the idea of merging inverted lists of grams generated from the strings. In this paper we make two contributions. First, we develop several algorithms that can greatly improve the performance of existing algorithms. Second, we study how to integrate existing filtering techniques with these algorithms, and show that they should be used together judiciously, since the way to do the integration can greatly affect the performance. We have conducted experiments on several real data sets to evaluate the proposed techniques.**

I. Introduction

Text data is ubiquitous. Management of string data in databases and information systems has taken on particular importance recently. In this paper, we study the following problem: given a collection of strings, how to efficiently find those in the collection that are similar to a query string? Such a query is called an "approximate string search." This problem is of great interests to a variety of applications, as illustrated by the following examples.

Spell Checking: Given an input document, a spellchecker needs to find possibly mistyped words by searching in its dictionary those words similar to the words. Thus, for each word that is not in the dictionary, we need to find potentially matched candidates to recommend.

Data Cleaning: Information from different data sources often have various inconsistencies. The same real-world entity could be represented in slightly different formats. There could also be errors in the original data introduced in the data-collection process. For these reasons, data cleaning needs to find from a collection of entities those similar to a given entity. A typical query is "find addresses similar to `PO Box 23, Main St.`", and an entity of "`P.O. Box 23, Main St`" should be found and returned.

These applications require a high real-time performance for each query to be answered, especially for those applications adopting a Web-based service model. For instance, consider a spellchecker such as those used by Gmail, Hotmail, or Yahoo Mail. It needs to be invoked many times every second since there can be millions of users of the service. Each spellchecking request needs to be processed as fast as possible. Although from an individual user's perspective, there is not much difference between a 20ms processing time and a 2ms processing time, from the server's perspective, the former means 50 queries per second (QPS), while the latter means 500 queries per second. Clearly the latter processing time gives the server more power to serve more user requests per second. Thus it is very important to develop algorithms for answering such queries as efficiently as possible.

Many algorithms have been proposed, such as [1], [2], [3], [4], [5], [6], [7]. These techniques assume a given similarity function to quantify the closeness between two strings. Different string-similarity functions have been studied, such as edit distance [8], cosine similarity [2], and Jaccard coefficient [9]. Many of these algorithms use the concept of *gram*, which is a substring of a string to be used as a signature of the string. These algorithms rely on inverted lists of grams to find candidate strings, and utilize the fact that similar strings should share enough common grams. Many algorithms [9], [2] mainly focused on "join queries," i.e., finding similar pairs from two collections of strings. Approximate string search could be treated as a special case of join queries. It is well understood that the behavior of an algorithm for answering selection queries could be very different from that for answering join queries. We believe approximate string search is important enough to deserve a separate investigation.

Our contributions: In this paper we make two main contributions. First, we propose three efficient algorithms for answering approximate string search queries, called ScanCount, MergeSkip, and DivideSkip. Normally, the main operation in answering such queries is to merge the inverted lists of the grams produced from the query string. The ScanCount algorithm adopts a simple idea of scanning the inverted lists and counting candidate strings. Despite the fact that it is very naive, when combined with various filtering techniques, this algorithm can still achieve a high performance. The MergeSkip algorithm exploits the value differences among the inverted lists and the threshold on the number of common grams of similar strings to skip many irrelevant candidates on the lists. The DivideSkip algorithm combines the MergeSkip algorithm and the idea in the MergeOpt algorithm proposed in [9] that divides the lists into two groups. One group is for those long lists, and the other group is for the remaining lists. We run the MergeSkip algorithm to merge the short lists with a different threshold, and use the long lists to verify the candidates. Our

experiments on three real data sets showed that the proposed algorithms could significantly improve the performance of existing algorithms.

Our second contribution is a study on how to integrate various filtering techniques with the proposed merging algorithms. Various filters have been proposed to eliminate strings that cannot be similar enough to a given string. Surprisingly, our experiments and analysis show that a naive solution of adopting all available filtering techniques might not achieve the best performance to merge inverted lists. Intuitively, filters can segment inverted lists to relatively shorter lists, while merging algorithms need to merge these lists. In addition, the more filters we apply, the more groups of inverted lists we need to merge, and more overhead we need to spend for processing these groups before merging their lists. Thus filters and merging algorithms need to be integrated judiciously by considering this tradeoff. Based on this analysis, we classify filters into two categories: *single-signature filters* and *multi-signature filters*. We propose a strategy to selectively choose proper filters to build an index structure and integrate them with merging algorithms. Experiments show that our strategy reduces the running time by as much as one to two orders of magnitude over approaches without filtering techniques or strategies that naively use all the filtering techniques.

In this paper we consider several string similarity functions, including edit distance, Jaccard, Cosine, and Dice [2]. We qualify the effectiveness and generalization capability of these techniques by showing our new merging and filtering strategies are efficient for those similarity functions.

Paper Outline: Section II gives the preliminaries. Section III presents our new algorithms. Section IV discusses how to judiciously integrate filtering techniques with merging algorithms. Section V shows how the results on edit distance can be extended to other similarity functions. Section VI discusses related work, and Section VII concludes this paper.

II. Preliminaries

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: We introduce two characters α and β not in Σ. Given a string s and a positive integer q, we extend s to a new string s' by prefixing $q-1$ copies of α and suffixing $q-1$ copies of β. 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 of s', i.e., $g = s'[i, i+q-1]$. The set of *positional q-grams* of s, denoted by $G(s, q)$ (or simply $G(s)$ when the q value is clear in the context) 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 $\alpha = \#$, $\beta = \$$, $q = 3$, and s = "smith", then $G(s, q) = \{(1, \#\#\texttt{s})$, $(2, \#\texttt{sm})$, $(3, \texttt{smi})$, $(4, \texttt{mit})$, $(5, \texttt{ith})$, $(6, \texttt{th}\$)$, $(7, \texttt{h}\$\$)\}$. Our discussion in this paper is also valid when strings are not extended using the special characters.

Approximate String Search: Given a collection of strings S, a query string Q, and a threshold δ, we want to find all $s \in S$ such that the similarity between s and Q is no less than δ. Various similarity functions can be used, such as edit distance, Jaccard similarity, cosine similarity, and dice similarity. In this paper, we first focus on edit distance, then generalize our techniques to other similarity functions. The *edit distance* (a.k.a. Levenshtein distance) between two strings s_1 and s_2 is the minimum number of edit operations of single characters that are needed to transform s_1 to s_2. Edit operations include insertion, deletion, and substitution. We denote the edit distance between two strings s_1 and s_2 as $ed(s_1, s_2)$. For example, $ed($"Steven Spielburg", "Steve Spielberg"$) = 2$. When using this function, our problem becomes finding all $s \in S$ such that $ed(s, Q) \leq k$ for a given threshold k.

III. Merging Algorithms

Several existing algorithms assume an index of inverted lists for the grams of the strings in the collection S to answer approximate string queries on S. In the index, for each gram g of the strings in S, we have a list l_g of the ids of the strings that include this gram, possibly with the corresponding positional information of the gram in the strings. It is observed in [9] that the search problem based on several string-similarity functions can be solved by solving the following generalized problem:

> *T-occurrence Problem*: Let Q be a query, and $G(Q, q)$ be its corresponding set of q-grams for a constant q. Find the set of string ids that appear at least T times on the inverted lists of the grams in $G(Q, q)$, where T is a constant.

For instance, it is known that if the edit distance between two strings s_1 and s_2 is no greater than k, then they should share at least the following number of q-grams: $T = max\{|s_1|, |s_2|\} + q - 1 - k \cdot q$. If this threshold is zero or negative, then we need to scan the entire data set in order to compute the answers. Various filters can help us reduce the number of strings that need to be scanned (Section IV).

The result of generalized problem is a set of candidate strings. We then need to eliminate the false positives in the candidates by applying the similarity function on the candidates and the query string. Existing algorithms for solving this problem focus on reducing the running time to merge the record-id (RID) lists of the grams of the query string. An established optimization is to sort the record ids on each inverted list in an ascending order. We briefly describe two existing algorithms as follows [9].

Heap algorithm: When merging the lists, we maintain the frontiers of the lists as a heap. At each step, we pop the top from the heap, and increment the count of the record id corresponding to the popped frontier record. We remove this record id from this list, and reinsert the next record id on the list (if any) to the heap. We report a record id whenever its count is at least the threshold T. Let $N = |G(Q, q)|$ denote the number of lists corresponding to the grams from the query

string, and M denote the total size of these N lists. This algorithm requires $O(MlogN)$ time and $O(N)$ storage space (not including the size of the inverted lists) for storing the heap of the frontiers of the lists.

MergeOpt algorithm: It treats the $T-1$ longest inverted lists of $G(Q,q)$ separately. For the remaining $N-(T-1)$ relatively short inverted lists, we use the Heap algorithm to merge them with a lower threshold, i.e., 1. For each candidate string, we do a binary search on each of the $T-1$ long lists to verify if the string appears on at least T times among all the lists. This algorithm is based on the observation that a record in the answer must appear on at least one of the short lists. Experiments have shown that the algorithm is significantly more efficient than the Heap algorithm.

We now present three new merging algorithms.

A. Algorithm: ScanCount

This algorithm improves the Heap algorithm by eliminating the heap data structure and the corresponding operations on the heap. Instead, we just maintain an array of counts for all the string ids in S. We scan the N inverted lists one by one. For each string id on each list, we increment the count corresponding to the string by 1. We report the string ids that appear at least T times on the lists. The algorithm is formally descried in Figure 1.

```
Input: set of RID lists and a threshold T;
Output: record ids that appear at least T times on the lists.
1.   Initialize the array C of |S| counters to 0's;
2.   Initialize a result set R to be empty;
3.   FOR (each record id r on each given list) {
4.       Increment the value of C[r] by 1;
5.       IF (C[r] == T)
6.           Add r to R;
7.   }
8.   RETURN R;
```

Fig. 1. ScanCount Algorithm.

The time complexity of this algorithm is $O(M)$ (compared to $O(MlogN)$ for the Heap algorithm). The space complexity is $O(|S|)$, where $|S|$ is the size of the string collection, since we need to keep a count for each string id. This higher space complexity (compared to $O(N)$ for the Heap algorithm) is not a major concern, since this extra space tends to much smaller than that of the inverted lists. This algorithm shows that the T-occurrence problem is indeed different from the problem of merging multiple sorted lists into one long sorted lists, since we care more about finding those ids with enough occurrences, rather than generating a sorted list.

One computational overhead in the algorithm is the step to initialize the counter array to 0's for each query (line 1). This step can be eliminated by storing an additional query id for each counter in the array. When the system first gets started, we initialize the counters to 0's, and their associated query ids to be 0. When a new query arrives, we assign a unique id to the query (incrementally from 0). Whenever we access the counter for a string id from an inverted list, we first check if the query id associated with the counter is the same as the current query id. If so, we take the same actions as before. Otherwise, we assign the new query id to this string id, and set its counter to 1. In this way, we do not need to initialize the array counters for each query. The drawback of this new approach is that in each iteration, we need to do an additional comparison of the two query ids. Which approach is more efficient depends on the total number of strings in the collection, the expected number of strings whose counters need to be updated, and whether we want to support multiple queries concurrently.

Despite its simplicity, this algorithm could still achieve a good performance when combined with various filtering techniques, if they can shorten the inverted lists to be merged, as shown in our experimental results.

B. Algorithm: MergeSkip

This algorithm is formally described in Figure 2. Its main idea is to skip on the lists those record ids that cannot be in the answer to the query, by utilizing the threshold T. Similar to Heap algorithm, we also maintain a heap for the frontiers of these lists. A key difference is that, during each iteration, we pop those records from the heap that have the same value as the top record t on the heap. Let the number of popped records be n. If there are at least T such records, we add t to the result set (line 8 in the algorithm), and add their next records on the lists to the heap. Otherwise, we are sure record t cannot be in the answer. In addition to popping these n records, we pop $T-1-n$ additional records from the heap (line 12). Therefore, in this case, we have popped $T-1$ records from the heap. Let t' be the current top record on the heap. For each of the $T-1$ popped lists, we locate its smallest record r such that $r \geq t'$ (line 15). This locating step can be done efficiently using a binary search. We then push r to the heap (line 16). Notice that it is possible to reinsert the same record on the popped lists back to the heap if it is equal to the new top record t'. Also for those lists that do not have such a record $r \geq t'$, we do not insert any record from these lists to the heap.

As an example, consider the four RID lists shown in Figure 3 and a threshold $T = 3$. At the beginning, we push their frontier ids 1, 10, 50, and 100 to the heap. The current top of the heap is id 1. There is only one record on the heap with the value, and we pop this record from the heap (i.e., $n = 1$). Then we pop $T-1-n = 3-1-1 = 1$ smallest record from the heap, which is the record id 10 (line 12 in the algorithm). Now the top record on the heap is $t' = 50$, as shown on the right-hand side in the figure. For each of the two popped lists, we locate the next record (using a binary search) that is no greater than 50. In this way, we can skip many records that cannot be in the answer. The next records on these two lists both have the same value 50. In the next iteration, we have three records with the current top-record value 50, and we add this record to the result set.

C. Algorithm: DivideSkip

Its main idea is to combine MergeSkip and MergeOpt, both of which try to skip irrelevant records on the lists,

Input: a set of RID lists and a threshold T;
Output: record ids that appear at least T times on the lists.
1. Insert the frontier records of the lists to a heap H;
2. Initialize a result set R to be empty;
3. **WHILE** (H is not empty) {
4. Let t be the top record on the heap;
5. Pop from H those records equal to t;
6. Let n be the number of popped records;
7. **IF** ($n \geq T$) {
8. Add t to R;
9. Push next record (if any) on each popped list to H;
10. }
11. **ELSE** {
12. Pop $T - 1 - n$ smallest records from H;
13. Let t' be the current top record on H;
14. **FOR** (each of the $T - 1$ popped lists) {
15. Locate its smallest record $r \geq t'$ (if any);
16. Push this record to H;
17. }
18. }
19. }
20. **RETURN** R;

Fig. 2. MergeSkip Algorithm.

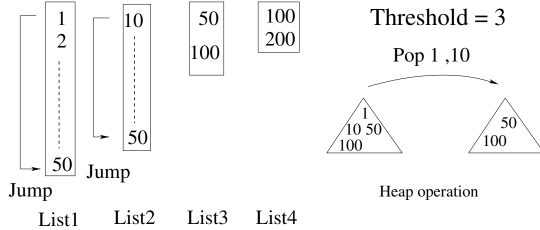


Fig. 3. Running MergeSkip algorithm.

but using different intuitions. MergeSkip exploits the *value differences* among the records on the lists, while MergeOpt exploits the *size differences* among the lists. Our new algorithm DivideSkip uses both differences to further improve the search performance.

Figure 4 formally describes the algorithm. Given a set of RID lists, we first sort these lists based on their lengths. We divide the lists into two groups. We group the L longest lists to a set $\mathcal{L}_{long}$, and the remaining short lists as another set $\mathcal{L}_{short}$. (The choice of the parameter L will be discussed shortly.) We use the MergeSkip algorithm on $\mathcal{L}_{short}$ to find records r that appear at least $T - L$ times on the short lists. For each such record r and each list l_{long} in $\mathcal{L}_{long}$, we check if r appears on l_{long}. This step can be done efficiently using $O(logp)$ time (where p is the length of l_{long}) if the list is implemented as an ordered list, or $O(1)$ time if the list is implemented as an unordered hash set. If the total number of occurrences of r among all these lists is at least T, then we add it to the result set R.

There are two main differences between MergeOpt and DivideSkip. (1) The number of long lists in DivideSkip is a tunable parameter L, which can greatly affect the performance of the algorithm. In MergeOpt, L is fixed to a constant $T-1$. (2) Unlike MergeOpt, which uses a heap-based algorithm

Input: set of RID lists and a threshold T;
Output: record ids that appear at least T times on the lists.
1. Initialize a result set R to be empty;
2. Let $\mathcal{L}_{long}$ be the set of L longest lists among the lists;
3. Let $\mathcal{L}_{short}$ be the remaining short lists;
4. Use MergeSkip on $\mathcal{L}_{short}$ to find ids that appear at least $T - L$ times;
5. **FOR** (each record r found) {
6. **FOR** (each list in $\mathcal{L}_{long}$)
7. Check if r appears on this list;
8. **IF** (r appears $\geq T$ times among all lists)
9. Add r to R;
10. }
11. **RETURN** R;

Fig. 4. DivideSkip Algorithm.

to process $\mathcal{L}_{short}$, the DivideSkip algorithm uses the more efficient MergeSkip algorithm to process the short lists.

1) Choosing Parameter L in DivideSkip: The parameter L affects the overall performance of the algorithm in two ways. If we increase L, fewer lists are treated as short lists, which need to be merged with a lower threshold $T - L$. The time of accessing the short lists will decrease. On the other hand, for each candidate after accessing the short lists, we need to do more lookups on the long lists. A main issue is how to choose a good L value for this algorithm. The best L value is difficult to decide since it depends on the query and its inverted lists. We propose a formula to calculate a good value for the parameter L for a given query, which has been empirically shown to be a close-to-optimal value.

Proposition 1: Given a set of inverted lists and a threshold T, a good L value in DivideSkip can be estimated as:

$$L_{good} = \frac{T}{\mu logM + 1}, \quad (1)$$

where M is the length of the longest inverted list of the grams of the query, and μ is a coefficient dependent on the data set, but independent from the query.

The following is the intuition behind this formula. Let L denote the number of lists in $\mathcal{L}_{long}$, and N denote the total number of records in $\mathcal{L}_{short}$. The total time to access the short lists can be estimated as:

$$C_1 = \phi \cdot N, \quad (2)$$

where ϕ is a constant. Let x denote the number of records whose number of occurrences in $\mathcal{L}_{short}$ is at least $\geq T - L$. We can estimate x as $\frac{\eta \cdot N}{T-L}$, where η is a parameter dependent on the data set S. For each candidate record from the short lists, its lookup time in the long lists can be estimated as $L \cdot logM$. Hence, the total lookup time on the long lists is:

$$C_2 = \frac{\eta \cdot N}{T - L} \cdot L \cdot \log M. \quad (3)$$

The total running time is $C_1 + C_2$. There is a tradeoff between C_1 and C_2. Assuming that the best performance is achieved when $C_1 = C_2$, we can get Equation 1 by replacing $\frac{\eta}{\phi}$ by μ.

The parameter μ in the formula can be computed offline as follows. We generate a workload of queries. For each query

q_i, we try different L values and identify its optimal value for this query. Using Equation 1 we compute a value μ_i for this query. We set μ as the average of these μ_i values from the queries.

Weighted Functions: The three new algorithms can be easily extended to the case where different grams have different weights. In ScanCount, we can record the cumulative weights of each string id in the array C (line 4). In MergeSkip and DivideSkip, we can replace the occurrence of a string id on the inverted lists with the sum of their weights on the lists, while the main idea and steps are the same as before.

D. Experiments

We evaluated the performance of the five merging algorithms: Heap, MergeOpt, ScanCount, MergeSkip, and DivideSkip, on three real data sets.

- **DBLP dataset**: It includes paper titles downloaded from the DBLP Bibliography site[1]. The raw data was in an XML format, and we extracted 274,788 paper titles with a total size 17.8MB. The average size of gram inverted lists for a query was about 67, and the total number of distinct grams was $59,940$.
- **IMDB dataset**: It consists of the actor names downloaded from the IMDB website[2]. There were 1,199,299 names with a total size of 22MB. The average number of gram lists for a query was about 19, and the number of unique grams was 34,737.
- **WEB Corpus dataset**: It is a collection of a sequence of English words that appear on the Web. It came from the LDC Corpus set (number LDC2006T13) at the University of Pennsylvania. The raw data was around 30GB. We randomly chose 2 million records with a size of 48.3MB. The number of words on the sequences varied from 3 to 5. The average number of inverted lists for a query was about 26, and the number of unique grams was 81,620.

We used edit distance as the similarity function. The gram length q was 3 for the data sets. All the algorithms were implemented using GNU C++ and run on a Dell PC with 2GB main memory, and a 2.13GHz Dual Core CPU running a Ubuntu operating system. Index structures were assumed to be in memory.

Query time: We ran 100 queries with an edit-distance threshold of 2. We increased the number of records for each data set. Figure 5 shows the average query time for each algorithm. For all the data sets, the three new algorithms were faster than the two existing algorithms. DivideSkip always achieved the best performance. It improved the performance of the Heap and MergeOpt algorithms 5 to 100 times. For example, for a DBLP data set with $200,000$ strings, the Heap algorithm took $114.50ms$ for a query, the MergeOpt algorithm took $13.3ms$, while DivideSkip required just $1.34ms$. This significant improvement can be explained using Figure 6, which shows the number of string ids visited during the merging phase of the algorithms. Heap and ScanCount need to read and process all the ids on the inverted lists of the grams in each query. Our new algorithms can skip many irrelevant ids on the lists. The number of ids visited in DivideSkip is the smallest, resulting in a significant reduction in the running time.

MergeSkip was more efficient than MergeOpt for all the data sets. Although both algorithms try to exploit the threshold to skip elements on the lists, MergeSkip often skipped more irrelevant elements than MergeOpt. For example, for the Web Corpus data set with 2 million strings, the number of visited string ids was reduced from 1600K (MergeOpt) to 1090K (MergeSkip). As a consequence, MergeSkip reduced the running time from 79ms to 42ms.

Choosing L for DivideSkip: We empirically evaluated the trade-off between the merging time to access the short lists and the lookup time for checking the candidates from the short lists on the long lists in the DivideSkip algorithm with various L values on the DBLP data set. Figure 7 shows the results, which verified our analysis: increasing the L value can reduce the merging time, but increase the lookup time. In Figure 8, we report the total running time by varying the L value. For comparison purposes, we also used an exhaustive search approach for finding an optimal L value, which was very close to the one computed by the formula. The results verified that the formula in Equation 1 indeed provides us a good optimal L value. For example, the running time was $26.40ms$ when $L = T - 1$, and it was reduced to $1.95ms$ when $L = \frac{T}{\mu logM+1}$. The μ value was 0.0085. The figure does not show the optimal L value found by the exhaustive search, which is very close to the value computed by the formula.

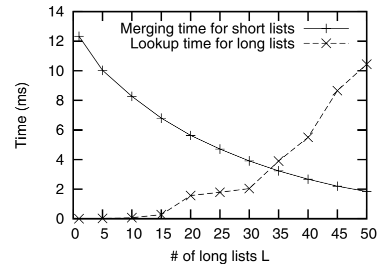
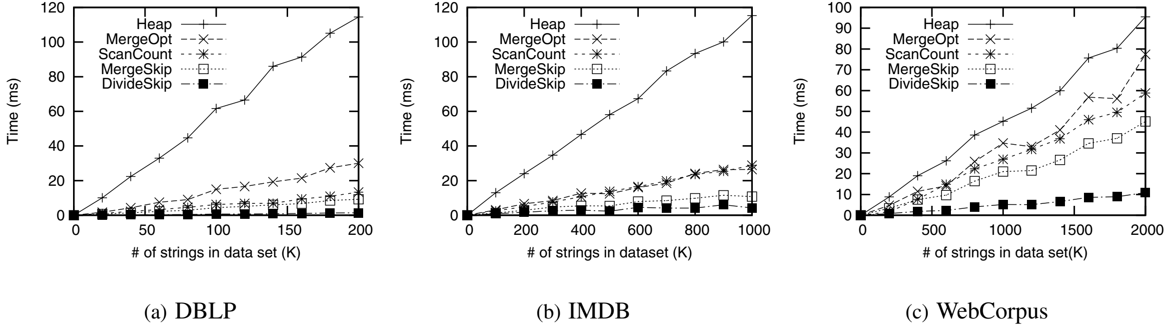


Fig. 7. Tradeoff between merging time and lookup time in DivideSkip (DBLP).

IV. Integrating Filtering Techniques with Merging Algorithms

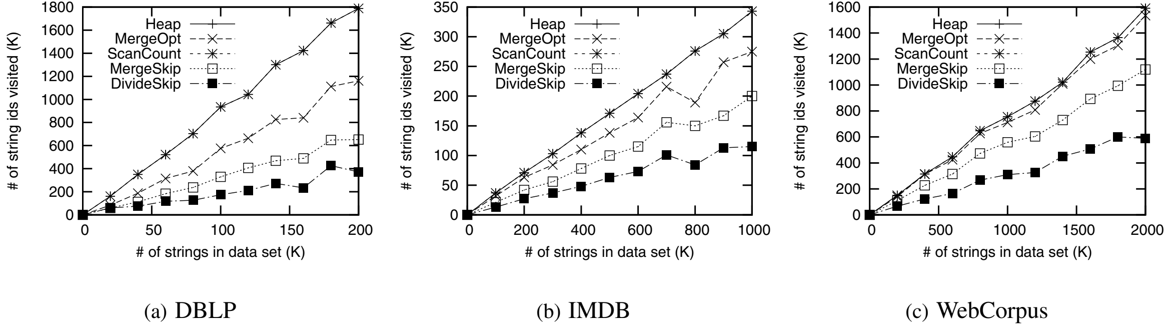
Various filters have been proposed in the literature to eliminate strings that cannot be similar enough to a query string. In this section, we investigate several filtering techniques, and study how to integrate them with merging algorithms to enhance the overall performance. A surprising observation is that adopting all available filters might not achieve the best performance, thus we need to do the integration judiciously.

[1] www.informatik.uni-trier.de/∼ley/db
[2] www.imdb.com



(a) DBLP (b) IMDB (c) WebCorpus

Fig. 5. Average query time versus data set size.



(a) DBLP (b) IMDB (c) WebCorpus

Fig. 6. Number of string ids visited by the algorithms.

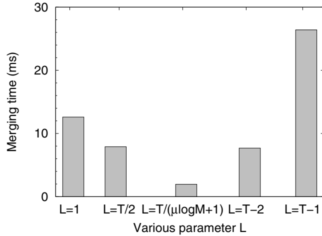


Fig. 8. Running time versus the L value.

For simplicity, in this section we mainly focus on the edit distance function.

A. Classification of Filters

A filter generates a set of *signatures* for a string, such that similar strings share similar signatures, and these signatures can be used easily to build an index structure. These filters can be classified into two categories. *Single-signature filters* are those that generate a single signature (typically an integer or a hash code) for a string. *Multi-signature filters* are those that generate multiple signatures for a string. To illustrate these two categories, we use three well-known filtering techniques: length filter, position filter [4], and prefix filter [10], which can be used for the edit distance function.

Length Filtering: If two strings s_1 and s_2 are within edit distance k, the difference between their lengths cannot exceed k. Thus, given a query string s_1, we only need to consider strings s_2 in the data collection such that the difference between $|s_1|$ and $|s_2|$ is no greater than k. This filter is a single-signature filter, since it generates a single signature for a string, which is the length of the string.

Position Filtering: If two strings s_1 and s_2 are within edit distance k, then a q-gram in s_1 cannot correspond to a q-gram in the other string that differs by more than k positions. Thus, given a positional gram (i_1, g_1) in the query string, we only need to consider the other corresponding gram (i_2, g_2) in the data set, such that $|i_1 - i_2| \leq k$. This filter is a multi-signature filter, since it produces a set of positional grams as signatures for a string.[3]

Prefix Filtering [10]: Given two q-gram sets $G(s_1)$ and $G(s_2)$ for strings s_1 and s_2, we can fix an ordering $\mathcal{O}$ of the universe from which all set elements are drawn. Let $p(n, s)$ denote the n-th prefix element in $G(s)$ as per the ordering $\mathcal{O}$. For simplicity, $p(1, s)$ is abbreviated as p_s. An important property is that, if $|G(s_1) \cap G(s_2)| \geq T$, then $p_{s_2} \leq p(n, s_1)$, where $n = |s_1| - T + 1$. For instance, consider the gram set $G(s_1) = \{1, 2, 3, 4, 5\}$ for a query string, where each gram is represented as a unique integer. The first prefix of any set $G(s_2)$ that shares at least 4 elements with $G(s_1)$ must be

[3]Notice that the formulated problem described in Section III can be viewed as a generalization of the "count filter" described in [4] based on grams as signatures (called thereafter "gram filter"). The position filter can be used together with the count filter.

$\leq p(2, s_1) = 2$. Thus, given a query string Q and an edit distance threshold k, we only need to consider strings s in the data set such that $p_s \leq p(n, Q)$, where $n = |G(s)| - T + 1$, and the threshold $T = |s| + q - 1 - q \cdot k$. This filter is a single-signature filter, since it produces a single signature for each string s, which is p_s.

B. Applying Filters before Merging Lists

Existing filters can be combined to improve the search performance of merging algorithms. One way to combine them is to build a tree structure, in which each level corresponds to a filter. Such an indexing structure is called *FilterTree*. An example is shown in Figure 9. The first level of the tree is using a length filter. That is, we partition the strings based on their lengths. The second level is using a gram filter; we generate the grams for each string, and for each of its grams, we add the string id to the subtree of the gram. The third level is using a position filter; we further decide the child of each gram to which the string id should be added to based on the position of the gram in the string. Each leaf node in the filter tree is an inverted list of string ids. In the tree in the figure, the shown leaf node includes the inverted list of the ids of strings that have a length of 2, and have a gram `za` at the second position.

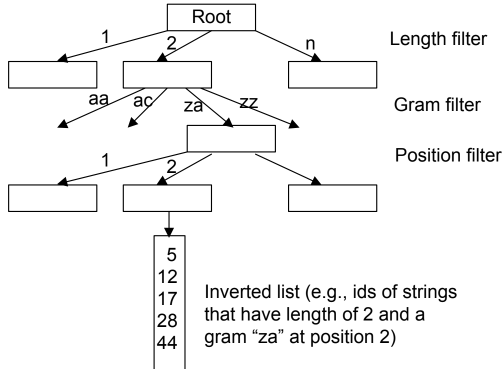


Fig. 9. A filter tree.

It is critical to decide which filters should be used on which levels. In order to achieve a high performance, we should first use those single-signature filters (close to the root of the tree), such as the length filter and the prefix filter. The reason is that, when constructing the tree, each string in the data set will be inserted to a single path, instead of appearing in multiple paths. During a search, for these filters we only need to traverse those paths on which the candidate strings can appear. After adding those single-signature filters, we add those multi-signature ones, such as the gram filter and the position filter. Using these filters, a string id could appear in multiple leaf nodes.

To answer an approximate string query using the index structure, we traverse the tree from the root. For the single-signature filters such as the length filter and the prefix filter, we only traverse those branches or paths with signatures that are "close" to those of the query. For instance, if the length of the query is 10, and the edit distance threshold is 3, for the level of the length filter, we only need to traverse the branches with a length between 7 and 13. After these levels, *for each candidate path*, we use the multi-signature filters such as the gram filter and the positional filter to identify the inverted lists, and use an algorithm to merge these lists. Notice that we run the algorithm for the inverted lists corresponding to each candidate path after the single-signature filters, and take the union of the results of the multiple calls of the merging algorithm.

Example 1: Consider the filter tree in Figure 9. Suppose we have a query that asks for strings whose edit distance to the string `smith` is within 1. Since the first level is using the length filter, and the length of the string is 5, we traverse the tree to visit the first-level children with a length between 4 and 6. We generate the 2-grams from the query string. For each of the three children, we use these grams to find the corresponding children. (This step can be done by doing some lookup within each of the three children.) For each of the identified gram node, we use the position of the gram in the query to identify the relevant leaf nodes using the position filter. For all these inverted lists from the children of the gram nodes, we run one of the merging algorithms. We call this merging algorithm for each of the three first-level children (with the length range between 4 and 6).

C. Experimental Results

We empirically evaluated various ways to integrate these filters with merging algorithms on the three data sets. Figure 10 shows the average running time for a query with an edit distance threshold 2, including the time to access the lists of grams from the query (columns marked as "Merge") and the total running time (columns marked as "Total"). The total time includes the time to merge the lists, the time to postprocess the candidate strings after applying the filters, and other time such that that of finding the inverted lists for grams. The smallest running time for each data set is marked as bold face. For instance, for the DBLP data set, the best performance was achieved when we used just the length filter with the DivideSkip algorithm. The total running time was 0.76ms, of which 0.47ms was spent to merge the lists. Figure 11 shows the number of lists and the total number of string ids on these lists per merging-algorithm call for various filter combinations. These numbers are independent from the merging algorithm.

We have the following observations from the results. First, for all the cases, DivideSkip always achieved the best performance among the merging algorithms, which is consistent with the observations in Section III. Second, the length filter reduced the running time significantly. Enabling prefix filtering in conjunction with length filtering further reduces the number of candidate strings.

The third observation is surprising: Adding more filters does not always reduce the running time. For example, for the DBLP data set, the best performance with all the filters was $4.16ms$, which was worse than just using the length

Time (ms)		No filters		Len		Len+Pre		Len+Pos		Len+Pre+Pos	
		Merge	Total	Merge	Total	Merge	Total	Merge	Total	Merge	Total
DBLP	Heap	114.53	115.42	11.67	11.98	7.69	8.83	2.77	3.64	2.62	5.69
	MergeOpt	13.32	14.22	1.09	1.40	0.95	2.12	5.96	6.78	5.82	8.89
	ScanCount	30.01	30.91	2.41	2.68	1.98	3.19	1.26	2.14	1.10	4.25
	MergeSkip	9.22	10.12	0.79	1.09	1.04	2.19	1.79	2.65	1.70	4.77
	DivideSkip	1.34	2.23	0.47	**0.76**	0.44	1.57	1.12	1.96	1.10	4.16
IMDB	Heap	115.32	113.7	58.78	58.91	26.07	26.54	24.19	24.58	24.29	25.32
	MergeOpt	28.83	29.21	11.20	11.32	6.25	6.65	23.46	23.76	20.76	21.70
	ScanCount	26.40	26.85	42.11	42.24	20.17	20.57	20.73	21.05	19.45	20.40
	MergeSkip	10.89	11.26	4.42	4.55	3.43	3.82	11.6	11.92	11.12	12.08
	DivideSkip	4.20	4.61	2.18	2.32	1.47	**1.84**	7.28	7.58	6.52	7.41
Web	Heap	95.49	96.58	25.35	25.47	30.92	31.50	21.40	22.07	19.25	20.84
	MergeOpt	58.83	59.52	14.24	14.35	12.16	12.67	28.42	28.92	26.64	28.08
	ScanCount	77.44	78.21	24.03	24.15	25.37	25.88	17.79	18.29	17.95	19.45
	MergeSkip	45.16	45.80	9.49	9.64	9.55	10.05	19.09	19.77	16.97	18.55
	DivideSkip	10.98	11.66	4.98	5.11	3.92	**4.42**	9.20	9.71	8.29	9.72

Fig. 10. The average running time for a query using various filters and merging algorithms ("Len" = length filter, "Pre" = Prefix filter, "Pos" = Position filter, edit distance threshold = 2, and 3-grams).

	# of lists					# of string ids on the lists (in thousands)				
	None	Len	Len+Pre	Len+Pos	Len+Pre+Pos	None	Len	Len+Pre	Len+Pos	Len+Pre+Pos
DBLP	65	64	32	191	951	2,448	24	12	9	5
IMDB	20	17	8	52	26	4,078	149	75	141	71
Web	27	27	14	89	47	1,591	87	45	56	29

Fig. 11. Number of lists and total number of string ids on the inverted lists per merging-algorithm call for various filter combinations.

filter ($0.76ms$). In other words, combining the position filter and the prefix filter even increased the running time. The same observation is also true for the other two data sets. As another example, for the DBLP data set, adding the prefix filter increased the running the time compared to the case where we only use the length filter. For both the IMDB data set and the Web Corpus data set, the best performance was achieved when we used the length and prefix filters.

Now we analyze why adding one more filter may not improve the performance. This additional filter can partition an inverted list into multiple, relatively shorter lists. The benefits of skipping irrelevant string ids on the shorter lists using algorithms such as DivideSkip or MergeSkip could be reduced. In addition, for each of the lists, we need to have an additional overhead such as the time of finding the inverted list for a gram (which is usually implemented as a lookup on a hash map). As a consequence, the overall running time for a query can be longer.

In order to further study the effect of the position filter, we group several positions together to one branch on the tree. For instance, inverted lists of grams with positions 7 to 10 can be grouped into one inverted list of a branch. Figures 12 and 13 show the results for the DBLP data set. Figure 12 shows how the running time changed as we increased the number of positions in one group. We used an edit distance threshold of 2 and the DivideSkip algorithm. Figure 13 shows the total number of string ids on the lists for each call to the merging algorithm. We find that as the number of positions in each group increased, the total number of string ids also increased, but the running time decreased. The results show that the position filter may not improve the performance for this data set.

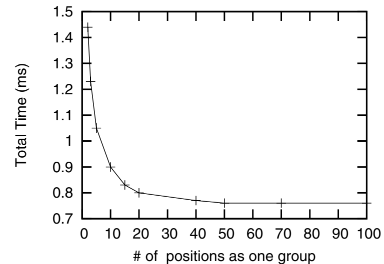


Fig. 12. Total running time with different numbers of positions in one group.

Summary: To efficiently integrate filters with merging algorithms, single-signature filters normally should be applied first on the filter tree. The effect of multi-signature filters on the overall performance needs to be carefully investigated before being used, since they may reduce the performance. It is due to the tradeoff between their filtering power and the additional overhead on the merging algorithm.

V. Extension to Other Similarity Functions

Our discussion so far mainly focused on the edit distance metric. In this section we generalize the results to the following commonly used similarity measures: Jaccard coefficient, Cosine similarity, and Dice similarity. Formally, given two strings s and t, let S and T denote the q-gram set of s and t (for a

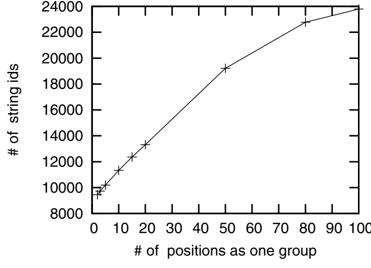


Fig. 13. Total number of string ids on inverted lists per merging-algorithm call.

given constant q), the Jaccard, Cosine, and Dice similarities are defined as belows.

- Jaccard(s, t) = $\frac{|S \cap T|}{|S \cup T|}$;
- Cosine(s, t) = $\frac{|S \cap T|}{\sqrt{|S| \cdot |T|}}$;
- Dice(s, t) = $\frac{2|S \cap T|}{|S|+|T|}$.

Table I and Table II show the formulas to calculate the corresponding overlapping threshold T in the corresponding "T-occurrence Problem." and the range on the string length using the length filter for each metric. In the tables, $|R|$ denotes the size of the gram set for a query, $|S_{min}|$ denotes the minimum size of the gram sets of the strings in the data set, and f is the given similarity threshold for the query. The prefix-filtering range for the metrics is the same as that in Section IV-A, and the threshold T needs to be updated correspondingly. The position filter is not applicable for these functions since they do not consider the position of a gram in a string.

TABLE I

LOWER BOUND ON THE NUMBER OF OCCURRENCES

Function	Definition	Merging threshold T
Jaccard	$\frac{\|R \cap S\|}{\|R \cup S\|} \geq f$	max($f \cdot \|R\|, \frac{\|R\|+\|S_{min}\|}{1+1/f}$)
Cosine	$\frac{\|R \cap S\|}{\sqrt{\|R\| \cdot \|S\|}} \geq f$	$f \cdot \sqrt{\|R\|}\sqrt{\|S_{min}\|}$
Dice	$\frac{2\|R \cap S\|}{\|R\|+\|S\|} \geq f$	$f \cdot (\|R\| + \|S_{min}\|)/2$

TABLE II

LENGTH RANGE.

	Length range
Jaccard	$[f \cdot \|R\| - q + 1\ ,\ \frac{\|R\|}{f} - q + 1]$
Cosine	$[f^2 \cdot \|R\| - q + 1,\ \frac{\|R\|}{f^2} - q + 1]$
Dice	$[\frac{f \cdot \|R\|}{2-f} - q + 1, \frac{(2-f)\|R\|}{f} - q + 1]$

We show how to derive the merging threshold for the Jaccard function only. The analysis for the other two functions is similar. For a query string, let R be its set of grams. Given a threshold f, we want to find those strings s in the dataset such that $Jaccard(R, S) = \frac{|R \cap S|}{|R \cup S|} \geq f$, where S is the gram set of the string s. Note that $|R \cup S| \geq |R|$. We have

$$|R \cap S| \geq f \cdot |R \cup S| \geq f \cdot |R|. \quad (4)$$

On the other hand, $\frac{|R \cap S|}{|R \cup S|} = \frac{|R \cap S|}{|R|+|S|-|R \cap S|}$. Hence,

$$|R \cap S| \geq \frac{|R| + |S|}{1 + 1/f} \geq \frac{|R| + |S_{min}|}{1 + 1/f}. \quad (5)$$

Combining Equations 4 and 5, we have:

$$|R \cap S| \geq max\{f \cdot |R|, \frac{|R| + |S_{min}|}{1 + 1/f}\}. \quad (6)$$

A. Experiments

Figure 14 shows the performance of the DivideSkip algorithm for the three similarity metrics on the DBLP data set. (The experimental results on IMDB and WebCorpus are similar.) Figure 14(a) shows the running time without filtering, Figure 14(b) shows the results with length filtering, and Figure 14(c) shows the results of using both the length filter and the prefix filter. We have the following observations. (1) The length filter is very effective improving the performance for all the three metrics. It could improve the performance of the case without using the filter five times. (2) The prefix filter in conjunction with the length filter further reduced the running time, and the average reduction was around 20%.

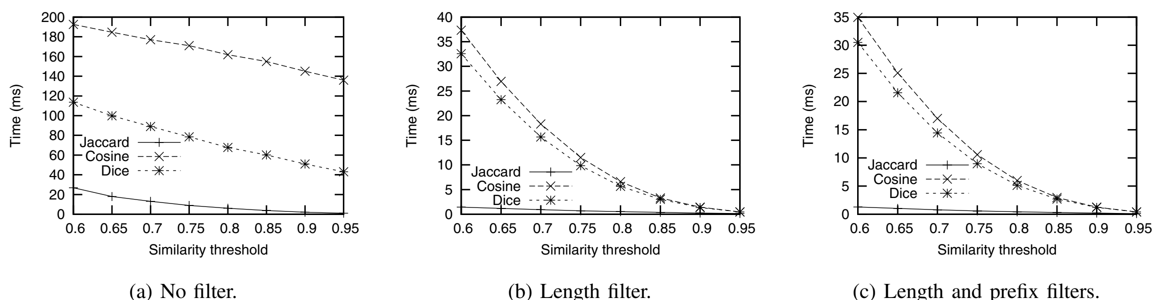
VI. RELATED WORK

In the literature "approximate string matching" also refers to the problem of finding a pattern string approximately in a text. There have been many studies on this problem. See [11] for an excellent survey. The problem studied in this paper is different; we want to search in a *collection of strings* those similar to a single query string ("selection queries"). In this paper we use "approximate string search" to refer to our problem.

Several algorithms (e.g., [3], [4]) have been proposed for answering approximate string queries efficiently. Their main strategy is to use various filtering techniques to improve the performance. These filters can be adopted with slight modifications to be written as SQL queries inside a relational DBMS. In this paper, we classify these filters into two categories, and analyze their effects on efficient approximate string search.

There is a large amount of work in the information retrieval (IR) community on designing efficient methods for indexing and searching strings. Their primary focus is to efficiently answer keyword queries using inverted indices. Our work is also based on inverted lists of grams. Our contribution here is in proposing several new merging algorithms for inverted lists to support approximate queries. Note that our "T-occurrence problem" is different from the problem of intersecting lists in IR. The IR community proposed many techniques to compress an in-memory inverted index, which would be useful in our problem too.

Other related studies include [1], [2], [10], [9], [12], [13] on similarity set joins. These algorithms find, given two collections of sets, those pairs of sets that share enough common elements. Similarity selections and similarity joins are in essence different. The former could be treated as a special case of the latter, but algorithms developed for the latter might not be efficient for the former. Approximate string



(a) No filter. (b) Length filter. (c) Length and prefix filters.

Fig. 14. Running time of DivideSkip using different similarity functions (DBLP data set).

search queries are important enough to deserve a separate investigation, which is the focus of this paper.

Recently, Kim et al. [14] proposed a technique called "n-Gram/2L" to improve space and time efficiency for inverted index structures. Li et al. [5] proposed a new technique called VGRAM to judiciously choose high-quality grams of variable lengths from a collection of strings. Our research in this paper is orthogonal to these studies and complementary to their work on grams. Our merging algorithms are independent on the indexing strategy, and can be easily used by those variant techniques based on grams.

VII. Conclusion

In this paper we studied how to efficiently find in a collection of strings those similar to a given string. We made two contributions. First, we developed new algorithms that can greatly improve the performance of existing algorithms. Second, we studied how to integrate existing filtering techniques with these algorithms, and showed that they should be used together judiciously, since the way to do the integration can greatly affect the performance. We reported the results of our extensive experiments on several real data sets to evaluate the proposed techniques.

References

[1] A. Arasu, V. Ganti, and R. Kaushik, "Efficient Exact Set-Similarity Joins," in *VLDB*, 2006, pp. 918–929.

[2] R. Bayardo, Y. Ma, and R. Srikant, "Scaling up all-pairs similarity search," in *WWW Conference*, 2007.

[3] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani, "Robust and Efficient Fuzzy Match for Online Data Cleaning," in *SIGMOD*, 2003, pp. 313–324.

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

[5] C. Li, B. Wang, and X. Yang, "VGRAM: Improving performance of approximate queries on string collections using variable-length grams," in *Very Large Data Bases*, 2007.

[6] E. Sutinen and J. Tarhio, "On Using q-Grams Locations in Approximate String Matching," in *ESA*, 1995, pp. 327–340.

[7] E. Ukkonen, "Approximae String Matching with q-Grams and Maximal Matching," *Theor. Comut. Sci.*, vol. 1, pp. 191–211, 1992.

[8] V. Levenshtein, "Binary Codes Capable of Correcting Spurious Insertions and Deletions of Ones," *Profl. Inf. Transmission*, vol. 1, pp. 8–17, 1965.

[9] S. Sarawagi and A. Kirpal, "Efficient set joins on similarity predicate," in *ACM SIGMOD*, 2004.

[10] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *ICDE*, 2006, pp. 5–16.

[11] G. Navarro, "A guided tour to approximate string matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, 2001.

[12] K. Ramasamy, J. M. Patel, R. Kaushik, and J. F. Naughton, "Set containment joins: The good, the bad and the ugly," in *VLDB*, 2000.

[13] N. Koudas, S. Sarawagi, and D. Srivastava, "Record linkage: similarity measures and algorithms," in *SIGMOD Tutorial*, 2005, pp. 802–803.

[14] M.-S. Kim, K.-Y. Whang, J.-G. Lee, and M.-J. Lee, "n-Gram/2L: A space and time efficient two-level n-gram inverted index structure." in *VLDB*, 2005, pp. 325–336.