Efficient Top-k Algorithms for Fuzzy Search in String Collections

Rares Vernica
Department of Computer Science
University of California, Irvine, CA 92697, USA
rares@ics.uci.edu

Chen Li
Department of Computer Science
University of California, Irvine, CA 92697, USA
chenli@ics.uci.edu

ABSTRACT

An approximate search query on a collection of strings finds those strings in the collection that are similar to a given query string, where similarity is defined using a given similarity function such as Jaccard, cosine, and edit distance. Answering approximate queries efficiently is important in many applications such as search engines, data cleaning, query relaxation, and spell checking, where inconsistencies and errors exist in user queries as well as data. In this paper, we study the problem of efficiently computing the best answers to an approximate string query, where the quality of a string is based on both its importance and its similarity to the query string. We first develop a progressive algorithm that answers a ranking query by using the results of several approximate range queries, leveraging existing search techniques. We then develop efficient algorithms for answering ranking queries using indexing structures of gram-based inverted lists. We answer a ranking query by traversing the inverted lists, pruning and skipping irrelevant string ids, iteratively increasing the pruning and skipping power, and doing early termination. We have conducted extensive experiments on real datasets to evaluate the proposed algorithms and report our findings.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; H.2.8 [**Database Applications**]: Miscellaneous

General Terms

Algorithms, Experimentation, Performance

1. INTRODUCTION

Consider an information system that has a relational table with records about actors. Its schema is (ID, FirstName, LastName, Popularity, ...), where the "Popularity" attribute has values between 0 and 1 indicating how popular this actor is (a higher value means a higher popularity). Figure 1 shows some of the records in the relation, with the first four columns corresponding to the four attributes. Suppose a user wants to find actors by providing a string `Shwartzenetrugger` as a last name. There is no record in the table that matches this string exactly, possibly due to the limited knowledge the user has about the exact spelling of the actor name(s) he or she is looking for.

ID	FirstName	LastName	Popularity	ED
10	`Al`	`Swartzberg`	0.20	8
11	`Hanna`	`Wartenegg`	0.18	8
12	`Rik`	`Swartzwelder`	0.10	8
13	`Joey`	`Swartzentruber`	0.10	**4**
14	`Rene`	`Swartenbroekx`	0.11	9
15	`Arnold`	`Schwarzenegger`	**0.95**	5
16	`Luc`	`Swartenbroeckx`	0.15	9
	⋮	⋮	⋮	⋮

Figure 1: Actor names, popularities, and edit distances to a query string "`Shwartzenetrugger`".

Instead of returning an empty answer to the user, the system relaxes the query condition and finds the best matches that could potentially be what the user is looking for. To find good answers, we need to use a similarity between the last name of an actor and the query string. Various functions can be used to measure the similarity between strings, such as Jaccard similarity, cosine similarity, and edit distance (a.k.a. Levenshtein distance). As an example, the last column in the figure shows the edit distance (marked as "ED") between each last name and the query string. Given these distances, one possible answer is the record with id 13, with the last name `Swartzentruber`, whose edit distance to the query is 4, the smallest among all the actors. On the other hand, this actor has a low popularity, which is 0.10. The actor number 15, with the last name `Schwarzenegger`, might also need to be returned, since it has a very high popularity (0.95) and its distance to the query is not too large (5). If the system can return only one entry to the user, it needs to make the decision based on how much importance it should give to the popularity of a data string and its similarity to the query string.

As illustrated by this example, many information systems need to support approximate string queries: given a collection of textual strings, such as last names in our example, telephone numbers, or addresses, find the strings in the collection that are similar to a given query string. Answering such queries is needed when the user may have limited knowledge about the data, or the data stored in the repository contains inconsistencies or errors. The following are a

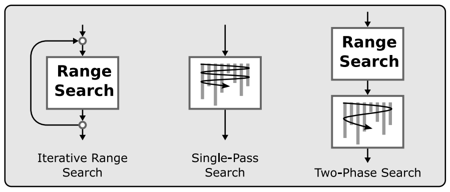


Figure 2: Overview of algorithms.

few applications. In record linkage [8], we often need to find from a table those records that are similar to a given query string that could represent the same real-world entity, even though they have slightly different representations, such as `Jack Lemmon` versus `Jacky Lemon`. In 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 in the document. Thus, for each word that is not in the dictionary, we need to find potentially matched candidates to recommend.

These applications often need to return the k best results for a query for a given integer k. As an example, a spellchecker can make a limited number of suggestions due to the user interface. In deciding the best answers, we need to consider both the similarity of a string to the query string and the "weight" of this string, such as the popularity of an actor in our running example, or the inverse-document frequency (IDF) of a word in a document corpus.

Contributions: In this paper we focus on how to support efficient string relaxation. Specifically, we relax a string by finding the most relevant strings based on their similarity with the original string and their importance. In the literature there have been studies on string similarity search assuming a similarity threshold is given. In addition to utilizing these existing techniques, we also need novel indexes and algorithms for finding top-k similar strings, which have not been studied in the past.

We first formulate the problem of approximate ranking queries on string collections, where the score of a record string is related to both its weight and similarity to the query string (Section 2). In Section 3 we present an iterative algorithm that can leverage existing string-relaxation algorithms developed for the case where a similarity threshold is given. In Section 4 we develop a single-pass-traversal algorithm. We use an index of inverted lists of grams and traverse the lists using a heap. Finally, we show how the range-search algorithm and the single-pass-traversal algorithm can be combined in what we call the two-phase algorithm. Figure 2 gives an overview of the proposed algorithms in this paper. In Section 5 we report our experimental results on real datasets to evaluate the proposed techniques.

1.1 Related Work

There have been many studies on ranking queries on relational tables after the pioneering studies by Fagin [3, 4]. See [7] for a recent survey. Our string-relaxation approach is different from the traditional setting of top-k queries in the following aspects. In the traditional setting, a record has multiple attributes, and each attribute has a list of record ids sorted based on their similarity to a query on that attribute. An aggregation function is used to combine these similarities (possibly with weights) to compute an overall score for each record, and we want to find the k best records. In our setting, we do not have multiple attributes. Instead, a string has multiple *grams*, and we have multiple lists corresponding to the grams in the query string. Each list only includes ids of those strings with the corresponding gram. *The similarity of a string to the query string is closely related to the number of occurrences of the string on these lists.* In addition, each string has a *single* weight, and we use its similarity to the query string and the weight to compute its overall score.

Several recent papers have focused on approximate string *selection* (or range search) [6, 9]. They assume a similarity threshold is given. Our ranking algorithms do not assume this threshold. In [6] strings are decomposed in grams and an IDF score is computed for each gram. The weight of a string is defined by aggregating the individual gram weights. In our setting, weights are assigned at the string level and grams do not have weights.

Many algorithms were developed for the problem of approximate string joins based on various similarity functions [1, 2, 5, 12], especially in the context of record linkage [8]. Some of them are proposed in the context of relational DBMS systems. The VGRAM technique [10] was shown to improve those algorithms based on edit distance.

Our approximate-string-search problem is different from the problem of finding within a long text string those substrings that are similar to a given query pattern. See [11] for an excellent survey on research related to this problem.

2. PRELIMINARIES

String Collection: Let S be a set of strings, such as a column of a table. Each string s in S has a weight $w(s)$ associated with it, which indicates the relative importance of this string. The weight could be the importance of a string in a particular application domain. For example, it can be computed using the inverse-document frequency (IDF) of a string.

Top-k Similar Strings: Given a string collection S, a string r, a similarity function θ, an aggregation scoring function F (that assigns a score to a string), and an integer k, the top-k similar strings to r are the k best strings in S in terms of overall score to r.

For a string s, 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. A *positional q-gram* of s is a pair (i,g), where g is the substring of length q starting at the i-th character of s, i.e., $g = s[i, i+q-1]$. The set of *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 s. For instance, suppose $q = 2$, and $s =$ "`john`", then $G(s,q) = \{$`jo`, `oh`, `hn`$\}$. The number of q-grams of the string s is $|s| - q + 1$.

Given two strings, the similarity function θ computes a similarity value $\theta(s_1, s_2)$ between two strings s_1 and s_2. Various similarity functions can be used. Commonly used similarity functions include edit distance, Cosine similarity, and Jaccard similarity. For instance, the *Jaccard* similarity of two strings s_1 and s_2 based on q-grams is $jaccard(s_1, s_2) = \frac{|G(s_1,q)\cap G(s_2,q)|}{|G(s_1,q)\cup G(s_2,q)|}$.

The scoring function F computes $F(r,s)$ as an overall *score* of the string s to the string r in terms of its weight and similarity to r.

For example, we can use Jaccard as the similarity function, and a linear combination of the similarity and the weight of the string as the final score of the string:

$$F\big(\theta(r,s), w(s)\big) = \alpha \cdot \theta(r,s) + \beta \cdot w(s). \qquad (1)$$

3. ITERATIVE RANGE-SEARCH-BASED ALGORITHM

We study how to answer a ranking query by answering (possibly multiple) range selection queries. Each selection query has a threshold on the similarity between the given string and a string in the collection. In this way, we can leverage existing approximate-string-selection techniques [12, 6, 9] without modifying their implementations. We develop an algorithm called "Iterative Range Search" ("IRS" for short). Algorithm 1 shows the pseudo-code of the algorithm. We start with an initial similarity threshold, τ, which could be a fixed value (e.g., 0.9 for Jaccard similarity) or a value computed based on the query (line 5). The algorithm has two steps.

Algorithm 1 : IRS Algorithm for a top-k query

```
 1: Let $k$ be the number of results requested;
 2: Let $w_{max}$ be the maximum weight of a string in the
    dataset;
 3: Let $f \geq 1$ be a multiplication factor;
 4: Let $R \leftarrow \phi$ be the range-search-result set;
 5: Let $\tau$ be the initial similarity threshold;
    {Step 1: Computing initial candidates}
 6: while $size(R) < f \cdot k$ do
 7:    $R \leftarrow ApproxRangeSearch(\tau)$;
 8:    if $size(R) < f \cdot k$
 9:    then Decrease $\tau$;
10: end while
    {Step 2: Finalizing results}
11: Compute scores for elements in $R$ and keep the first $k$;
12: Let $\tau_1$ be the minimum similarity for which
    $Score(\tau_1, w_{max}) > Score(R[k])$;
13: if $\tau_1 < \tau$ then
14:    $R \leftarrow ApproxRangeSearch(\tau_1)$;
15:    Compute scores for elements in $R$ and keep the first
       $k$;
16: end if
17: Return $R[1..k]$;
```

Step 1: Computing initial candidates (lines $6-10$). The goal of this step is to compute at least $f \cdot k$ results, where $f \geq 1$ is a multiplication factor. We call a function "ApproxRangeSearch" to run an approximate-string-range-search algorithm of our choice, and find the strings that pass the similarity threshold τ. Next, we decrease the similarity threshold, depending on the number of results we got. This step ends when we get at least $f \cdot k$ results.

Step 2: Finalizing results (lines $11-16$). We compute the score for each element computed in step 1, and keep the first k elements ordered by their scores. Next, we want to be certain that these k elements are indeed the best results. Consider one element e that was not seen before, and it has the maximum possible weight in the dataset. We compute how similar e needs to be to the query in order to have a better score than the current k^{th} element. We use this similarity as the new similarity threshold τ_1 (line 12). If $\tau_1 < \tau$, we call the approximate-range-search function one more time, using τ_1 as the threshold (line 14). The more results we got from step 1, the better the k^{th} element will be, and the tighter the similarity threshold τ_1 will be.

ID	String	Weight	Jaccard	Score
1	abcd	0.10	1.00	**1.10**
2	abcde	0.20	0.75	0.95
3	abc	0.30	0.66	**0.96**
4	abce	0.20	0.50	0.70
5	ab	0.70	0.33	1.03

Figure 3: Example of results for given string "abcd", Jaccard similarity, $k = 2$.

Figure 3 shows the intuition behind the need for a second step. The figure shows an example of the results for string "abcd". Suppose the first step returns the results with ids between 1 and 4. The current top-2 results are 1 and 3. The second step is needed in order to capture elements which have a low similarity but a high weight, like element with id 5. The final top-2 results are 1 and 5.

Advantages and Limitations: The IRS algorithm has the advantage that it can utilize any of the existing algorithms for approximate-string range search. It is easy to implement as it uses the range-search algorithm as a black-box function. One main limitation of the algorithm is that it needs to run multiple search queries, which may take a lot of time. In addition, it is not easy to choose a good initial similarity threshold τ (line 5) and decrease τ properly for the next query (line 9).

4. SINGLE-PASS SEARCH ALGORITHM

There are many different index structures that can be used to relax strings. In this paper, we use a *q-gram inverted-list index*, built on the string collection. For each gram in S, we have an inverted list of ids of strings containing this gram. For instance, Figure 4(a) shows four strings with their weights and Figure 4(b) shows the corresponding inverted lists of 2-grams. The ids on each list are sorted in ascending order. Without loss of generality, we assume that the ascending order of the string ids is identical to the descending order of their weights. If not, we can map each string id to a new id (a one-to-one mapping), so that the new string ids have this property.

ID	String	Weight
1	ab	0.80
2	ccd	0.70
3	cd	0.60
4	abcd	0.50
5	bcc	0.40

(a) Dataset

ab	cc	cd	bc
1	2	2	4
4	5	3	5
		4	

(b) Inverted lists

Figure 4: Example of gram inverted-list index.

We study how to answer a query by accessing the inverted lists only once. (We assume an answer should share at least one common gram with the given string.) A naive way to traverse the lists would be to loop over the lists, reading one element at a time from each list. During the traversal, we maintain the information about the visited elements (candidates) and a top-k buffer of the best k elements seen so far. The algorithm maintains bounds on the score for each candidate and stop when the top-k buffer cannot be improved.

A better way of traversing the lists is to use a heap. The algorithm is called "Single-Pass Search" ("SPS" for short). It traverses the lists in a sorted order using a heap of the current top elements of the lists. This traversal order has two advantages: we do not have the overhead of maintaining the candidate set, and we have more chances to skip elements. Algorithm 2 shows the pseudo-code of the algorithm.

Algorithm 2 : SPS Algorithm for a top-k query

```
 1: Let $n$ be the number of grams in the query;
 2: Let $l[0..n-1]$ be the lists of ids for the query grams;
 3: Let $g \leftarrow 1$ be the frequency threshold;
 4: Insert the top element on each list to a heap, $H$;
 5: $Topk \leftarrow \phi$;
 6: while $H$ is not empty do
 7:    Let $T$ be the top element on $H$;
 8:    Pop from $H$ those elements equal to $T$;
 9:    Let $p$ be the number of popped elements;
10:    if $p \geq g$ then
11:       if $Score(T) > Score(k^{th}$ in $Topk)$ then
12:          Insert $T$ into $Topk$ and pop the last one;
13:          Recompute threshold $g$;
14:          if $g > n$
15:          then break;
16:       end if
17:       Push next element (if any) of each popped list to
          $H$;
18:    else
19:       Pop additional $g-p-1$ elements from $H$;
20:       Let $T'$ be the current top element on $H$;
21:       for each of the $g-1$ popped lists do
22:          Locate its smallest element $E \geq T'$ (if any);
23:          Push $E$ to $H$;
24:       end for
25:    end if
26: end while
27: Return the elements in $Topk$;
```

We first initialize the frequency threshold g (line 3). The algorithm maintains a cursor for each list pointing to the current element. The cursor is initially set to the first element on the list. We maintain a heap, H, of the elements pointed by the cursors on the lists (line 4). We pop an element from the heap, process it, and push another element of this list to the heap. We traverse the lists by pushing and popping elements to and from the heap (lines $6-26$). If a string id appears on multiple lists, the heap has multiple copies of that id. Whenever we pop one element from the heap, we pop all its copies (line 8). In this way we know the total frequency, p, of the string id on the lists (line 9). Then, the element is processed, and retained in the top-k buffer, or discarded. There is no need for maintaining a candidate set (lines $10-25$).

The first k elements visited during the traversal become the top-k candidates, and are added to the top-k buffer (line 12). Next, we compute what new frequency threshold g a new element needs to have in order to have a better score than the k^{th} element in the buffer (line 13). We describe how to compute the frequency threshold g later in this section. If a new element has a frequency p less than the frequency threshold g, we pop additional $g-p-1$ elements from the heap, and move the cursor on each popped list to the first element greater than or equal to the current top on the heap (lines $19-24$). (Similar intuition has been used in the MergeSkip algorithm proposed in [9].) The algorithm stops when reaching the last element on each list, or when the frequency threshold g is greater than the number of inverted lists n.

We use the example in Figure 5 to show the intuition behind the skipping step. For simplicity, we use the number of common grams as similarity between two strings and a scoring function based only on the similarity. The figure shows a snapshot of the element-id lists after element 10 has been read from lists 1 and 2, and retained as the top-1 candidate. Currently the frequency threshold g is 3, thus an element needs to appear at least 3 times in order to be better than the current top-1. Next, the algorithm reads element 20 from the heap. As its frequency p is 1, the algorithm reads one more element from the heap. After that, the current head of the heap is 40, and the algorithm jumps on lists 1 and 2 to the element 40. In this way the algorithm skips many elements on lists 1 and 2. All the skipped elements could appear at most 2 times on the lists, while the frequency threshold is 3. Thus they cannot be the top-1 answer.

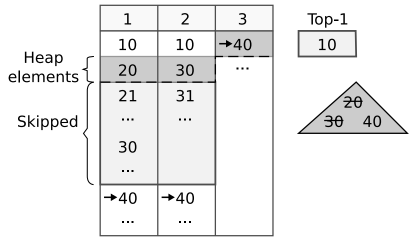


Figure 5: Skipping elements during the traversal of the lists in the SPS algorithm.

Computing Frequency Threshold: Given the score of the k^{th} candidate in the buffer, γ_k, we need to compute a frequency threshold, g, that an element needs to have in order to have a better score. First, the largest weight of the last seen weights on the lists is the maximum weight that an unseen element could have. We denote it by η_{max}. Next, using the definition of the scoring function, γ_k, and η_{max}, we derive the minimum similarity an element needs to have in order to be better than the current k^{th} element. We denote that by τ_{min}. For example, if the scoring function is Equation 1, then τ_{min} can be computed as:

$$\tau_{min} = \frac{\gamma_k - \beta \cdot \eta_{max}}{\alpha}.$$

Given τ_{min}, we can compute g using a formula derived from the definition of the similarity function. Formulas for deriving the frequency threshold for a given similarity have been defined in [9] for the common similarity functions. For example, if the similarity function is Jaccard, g can be computed as follows:

$$g = max(\tau_{min} \cdot g_r, \frac{g_r + g_{min}}{1 + 1/\tau_{min}}),$$

where g_r is the number of grams in the query, and g_{min} is the minimum number of grams of a string in the collection.

Improvement by Separating Lists The algorithm can be improved by partitioning the lists into a set of long lists and a set of short lists. It treats the long lists separately, since these lists could take a lot of time to traverse. The algorithm only searches in them for elements found on those short lists. A similar idea of treating the long lists differently was proposed in algorithms in [12, 9]. We use a heap to traverse the short lists. Whenever an element in the short lists has a frequency at least $g - n_{long}$, where n_{long} is the number of long lists, we can search for this element in each of the long lists by doing a binary search. We can use a formula derived in [9] for deciding n_{long}.

Two-Phase Algorithm: We can combine the IRS algorithm and the SPS algorithm. This new algorithm is called "two-phase" algorithm (2PH). In the first phase, we execute a single range search with a tight similarity threshold, τ. In the second phase, we run the SPS algorithm, but the initial bound on the number of common grams is computed based on the records retrieved in phase 1. The algorithm is based on the following two observations. (1) Retrieving the records very similar to the query could be done efficiently using existing range-search algorithms. (2) The SPS algorithm is efficient since it can skip many elements. Still, a low initial frequency threshold makes the algorithm process a lot of elements at the beginning. The initial top-k candidates computed in phase 1 could give as a higher initial frequency threshold. Moreover, the traversal might stop earlier since the records very similar to the query have already been considered.

5. EXPERIMENTAL EVALUATION

In this section we present our experimental evaluation of the proposed algorithms for string relaxation. We used two real datasets.

- *IMDB Actor Names*: It consisted of actor names, and the numbers of movies they played in. The data was downloaded from the IMDB website[1]. There were 1.2 million names, with the average length of 15. We log-normalized the number of movies an actor played in and used it as the weight of the actor.
- *WEB Corpus Word Grams*: It came from the LDC Corpus set at the University of Pennsylvania[2]. This dataset, contributed by Google Inc., contained sequences of English words and their observed frequency counts on the Web. The raw data was around 30GB. We randomly chose 2.4 million sequences, with the average length of 20. We log-normalized the frequency of the sequence and used it as its weight.

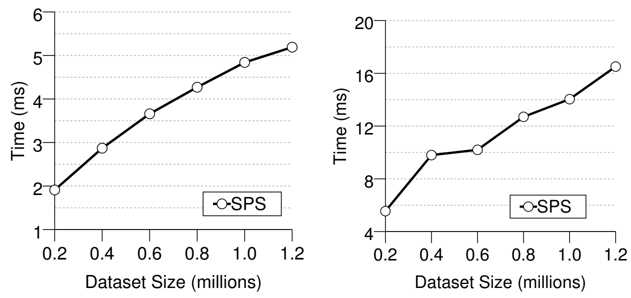
We used $q = 3$ for the gram length. For each dataset, we constructed 100 queries by randomly selecting strings from the dataset. All the algorithms were implemented using C++ (GNU compiler) and run on a Intel 2.40GHz PC with 2GB main memory, running a Ubuntu Linux operating system.

5.1 Efficiency of String Relaxation Algorithms

We evaluated the performance of the string relaxation algorithms for top-10 queries using the IMDB and Web Corpus datasets. We considered the scoring function in Equation (1), with $\alpha = 1$ and $\beta = 1$. We ran each query 5 times and used its average running time in order to compute accurate performance numbers.

[1]http://www.imdb.com/interfaces
[2]http://www.ldc.upenn.edu/Catalog, number LDC2006T13

Figure 6(a) shows the average query time of the SPS top-k string relaxation algorithm on the IMDB dataset using the Jaccard similarity. The IRS algorithm performed very poorly and we did not plot its running time. Even for the optimal initial threshold, the IRS algorithm needed around 5 seconds to compute the top-10 results for 1.2 million entries. This result was the best performance of the algorithm when we used the optimal initial similarity threshold and ran a single range search, and most time was spent on the post-processing phase. The performance of the algorithm was even worse if we did not use the optimal initial threshold. The SPS and 2PH algorithms have similar performance. For the 2PH algorithm we heuristically computed the initial threshold as being equal to the number of grams in the query. They needed around 5 ms to answer a top-10 query on 1.2 million entries. The time increased slowly as the dataset size increased.



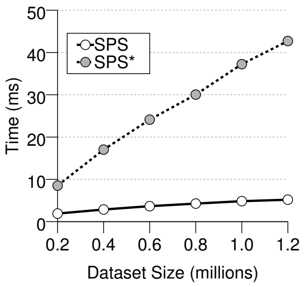
(a) Jaccard similarity (b) Normalized edit similarity

Figure 6: Average running time for top-10 string relaxation (IMDB).

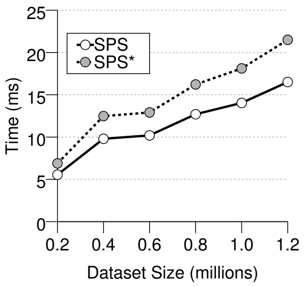
Figure 6(b) shows the results for the same setting but using the normalized edit similarity. The relative performance order of the algorithms is preserved, but all of them needed more time to compute the top-10 results than in the case we used Jaccard. This behavior is due to the fact that for Jaccard, from the number of common grams between two strings we could compute their exact similarity. For the normalized edit similarity, we computed bounds on the similarity, since computing the exact similarity is expensive. We observed similar results on the WEB Corpus dataset. Due to space limitations, we do not present the results.

5.2 Benefits of Skipping Elements

In this experiment, we analyzed how the skipping operation affects the running time of the single-pass search algorithms. We also ran the SPS algorithm for various subsets of the IMDB dataset with different sizes, with the skipping feature disabled. Figure 7(a) shows the average running time for the SPS algorithm with and without the skipping feature, using the Jaccard similarity. In the figure, the asterisk (*) near an algorithm name means that the algorithm had the skipping feature disabled. We can see that the skipping operation improves the performance. For example, the SPS algorithm needed around 40 ms if no skipping was performed, while it only needed around 5 ms with skipping (on 1.2 million entries). Figure 7(b) shows the results when we used the normalized edit similarity. The skipping operation helps to improve the performance of the algorithms for this function as well.



(a) Jaccard similarity

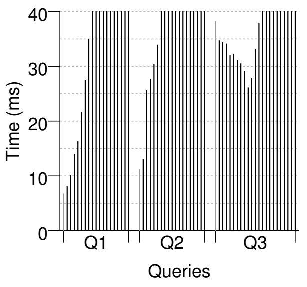


(b) Normalized edit similarity

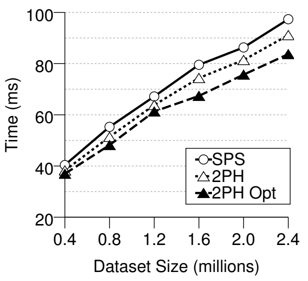
Figure 7: Average running time for top-10 string relaxation (IMDB).

5.3 Effect of the Initial Threshold for the 2PH Algorithm

We evaluated how the initial threshold affects the performance of the 2PH algorithm. We randomly generated 3 top-10 queries and ran the 2PH algorithm for different initial thresholds. We used the WEB Corpus dataset and normalized edit similarity. We converted the initial similarity threshold needed by the 2PH algorithm to a bound on the number of common grams, g, using the formulas in [9]. Figure 8(a) shows the average running times for the three queries, using different bounds for g. For each query, the first bar represents the execution time for only the second phase of the 2PH algorithm, i.e., we mainly ran the SPS algorithm. The subsequent bars represent the running times with g between the number of grams in the query and 1. We can see how the initial threshold affected the performance of the algorithm, and why the algorithm can have a better time than the SPS algorithm. In particular, for the third query, when we decreased the similarity threshold, the overall time first decreased, then increased.



(a) Running times for the 2PH algorithm on 3 queries, with different initial thresholds (2.4 million entries).



(b) Average running time for various dataset sizes.

Figure 8: Influence of the initial threshold for the 2PH algorithm on the WEB Corpus dataset (normalized edit similarity).

For each query in our workload, we computed the optimal g bound for the 2PH algorithm as follows. We ran each query with a g value between the number of grams in the query and 1, and selected the bound with the minimum running time. Figure 8(b) shows the average running time of the 2PH algorithm with a heuristically computed initial threshold (equal to the number of grams in the query) and the optimal initial threshold (shown as "2PH Opt" in the figure). This experimental result shows that the 2PH algorithm can indeed decrease the running time when a good initial threshold is used.

6. CONCLUSIONS

In this paper, we formulated the problem of approximate ranking queries in string collections, in which a string is ranked based on its weight and similarity to the query string. Answering such queries is important to many applications such as record linkage where there is a mismatch between a user query and the representations of the entities the user is looking for. We developed efficient algorithms for answering ranking queries by considering several commonly used similarity functions. Our extensive experiments on real datasets showed that these algorithms can answer ranking queries efficiently on large datasets.

7. REFERENCES

[1] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.
[2] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.
[3] R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, pages 216–226, 1996.
[4] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.
[5] 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*, pages 491–500, 2001.
[6] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, pages 267–276, 2008.
[7] I. Ilyas, G. Beskales, and M. A. Soliman. A Survey of Top-k Query Processing Techniques in Relational Database Systems. ACM Computing Surveys, 2008.
[8] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD Conference*, pages 802–803, 2006.
[9] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.
[10] C. Li, B. Wang, and X. Yang. VGRAM: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, pages 303–314, 2007.
[11] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.
[12] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, pages 743–754, 2004.