Efficient Top-k Algorithms for Approximate Substring Matching
Project Final Report

Apurva Katti
Department of Computer Science
Florida State University
Tallahassee, Florida
agk17c@my.fsu.edu

Muna Tammar
Department of Computer Science
Florida State University
Tallahassee, Florida
mt16k@my.fsu.edu

Abstract—***Text data is the most prevalent form of data observed in real-life. We often need to find strings of the large string database that is the closest match to the given input string. The Existing methods, however, requests the user to specify a similarity threshold. The algorithms mentioned in this paper provides an efficient way to reduce the number of expensive distance computations by proposing the efficient algorithms for finding the top-k approximate substring matches with a given query string of a set of data strings. Experimental computation is performed using one real-life datasets.***

Keywords—q-gram;lowerbound;maxheap; Top-k approximate substring matching;edit distance;

I. INTRODUCTION

The increasing trend of applications to deal with vast amounts of data, retrieving similar strings or the substrings has become a challenging problem today. A variety of applications are required to query the database of texts to search for similar strings or substrings. For example, approximate substring matching can be used for spell checking, 2D shape recognition, named entity recognition and bio-informatics for finding DNA subsequences. The Existing methods, however, requests the user to specify a similarity threshold but with this, the users have to try different distance threshold values, which can lead to unpredictable results. Also, the value of proper threshold will be unknown in advance which in turn may lead to empty results (if the threshold chosen is too low) or too many results increasing the running time (if the threshold chosen is too high).

To compute the approximate substring matching, various (dis) similarity measures such as edit distance, Hamming distance, Jaccard coefficient, cosine similarity and Jaro-Winkler distance have been considered. In accordance with the research paper, we have chosen edit distance as the distance measure throughout the project. Using the popular definition of approximate substring edit distance which is the minimum distance among the edit distances between σ (query string) and every substring of s, is calculated. Given a set of strings D and a query string σ, Naïve algorithm of top-k approximate substring matching examines every string s in D and computes the substring edit distance d_{sub} (s, σ) to all the strings by using a max heap. We started our project with the implementation of this algorithm. This is the brute force method. Since this is computationally expensive and takes more time for execution, we next proceeded to implement the proposed algorithm in the paper called the TopK-LB. This algorithm utilizes the generated q-grams for both query string and strings from the database and finds the common q-grams. Using the common matching, a lower bound for edit distances was calculated which significantly reduces the number of edit distance computations.

In the third algorithm called the TopK-SPLIT, which allows not to calculate the lower bounds for some strings by dividing the database D into partitions based on q-grams and then utilizing the

SIGMOD'13, June 22-27, 2013, New York, New York, USA.

proposed lower bound method of TopK-LB to find the edit distance between query and all the strings in each partition.

To speed up TopK-SPLIT, TopK-INDEX utilized the posting lists of some q-grams in the query string, which can be obtained from the existing inverted positional q-gram index.

In this project, we have successfully been able to implement the top-k algorithms mentioned above for the real dataset called the DBLP dataset. Given a query string and the value of k by the user, this project provides the requested k top approximate results to the query string.

II. SURVEY ON RELATED WORK

As a part of stage two of project formulation, a detailed literature survey was carried out on the two important topics related to top-k approximate substring matching.

A. *Approximate string matching*

The VGRAM algorithm mentioned in [1], developed a novel technique, called VGRAM, to improve the performance of algorithms for approximate string matching. Many algorithms were developed using fixed-length grams, which were substrings of a string used as signatures to identify similar strings. But the main idea of this paper was to judiciously choose high-quality grams of variable length from a collection of strings to support queries on the collection.

Using Iterative range-search-based algorithms, Single pass and two pass algorithm [2], instead of returning an empty answer to the ranking queries, the system relaxed the query condition and found the best match using indexing structures of gram-based inverted lists. Here approximate string query is based both on query importance and its similarity to the query string. Experiments were conducted using the IMDB Actor names dataset and WEB Corpus Word Grams.

The count filtering and length aware mechanisms [3] were used to tackle the top-k similar search issue, while an adaptive g-gram selection was employed to improve the performance of frequency counting.

B. *Approximate substring matching*

Fast Similarity Search algorithm (FastSS) [4], uses a proposed neighborhood deletion-based method to solve this problem. The TopK-FSS algorithm that the authors of Top-K Approximate Substring Matching papers proposed is an extension of the FastSS algorithm. They have adopted this to compute the top-k approximate substring matches (TopK-FSS). The way they did this was by using FastSS algorithm which was used to read the first k-th string in the set or array, sit k as the initial threshold and then compared each edit distance. If the edit distance is smaller than the threshold k, they update k to that value.

Neighborhood Generation with partitioning and prefix-based pruning(NGPP) [5], Given a dictionary of entities and a document, they first found all substrings in the document that are similar to some entities using the edit distance measure. This work is considered an improvement to the FastSS algorithm discussed above. The main difference is that they applied a semi-join style reduction technique to avoid considering many unnecessary query and entity pairs unlike FastSS, in addition to other optimizations. They improved the neighborhood size by using partitioning and prefix pruning techniques that resulted in a neighborhood size $O(l_p{}^{t2})$ where $L_b \leq m$ is a tunable parameter.

III. TECHNICAL DISCUSSION

Our entire project is based in java on NetBeans IDE and for parsing the dataset DBLP we use SAX. Parser and store the data in MySQL database. The detailed description of the dataset as well as the analysis of the implemented algorithms is given in the next section.

A. *Processing the dataset*

As mentioned in the research paper, we made a detailed analysis and implementation of the proposed algorithms using the real-life dataset called the DBLP dataset. This is a Computer Science Bibliography from the University of Trier which contains more than 1.2 million bibliographic records. For computer science researchers the DBLP web site is a popular tool to trace the work of colleagues and to retrieve bibliographic details when composing the lists of references for new papers.

The DBLP data may be downloaded at http://dblp.uni-trier.de/xml/. The bibliographic records are contained in a huge XML file. The size of the file is about 2.24 GB and contains about 13,966,030 strings. It also contains a data type definition file dblp.dtd. You need this auxiliary file to read the XML file with a standard parser. These two files are saved at the same folder for parsing using MySQL database and java IDE NetBeans.
dblp.xml has a simple layout:

```
<article key="journals/cacm/Szalay08"
    mdate="2008-11-03">
  <author>Alexander S. Szalay</author>
  <title>Jim Gray, astronomer.</title>
  <pages>58-65</pages>
  <year>2008</year>
  <volume>51</volume>
  <journal>Commun. ACM</journal>
  <number>11</number>
  <ee>http://doi.acm.org/10.1145/
    1400214.1400231</ee>
  <url>db/journals/cacm/
    cacm51.html#Szalay08</url>
</article>
```

To be specific, four key information namely author, conference, paper, and citation under <inproceedings>tag will be extracted and stored in MySQL database as four tables. The code is provided in the files element.java, conference.java and paper.java, dblp.sql. The code for connection to MySQL database is given in DBConnection.java file. Using SAX parser, the large XML file is parsed and stored in the database and the source code for this is provided in parser.java.
For the sake of approximate substring matching, we use only the DBLP titles appearing in the dataset as mentioned in the research paper. The only element which has to exist in every DBLP publication record is the title element. It may contain sub elements for subscripts, sup elements for superscripts, i elements for italics, and tt for typewriter text style. These elements may be nested.

This can be seen in each of the algorithms implemented in the project. The total number of titles in the dataset is 2,128,206 strings.

B. Top-K Algorithms

Let us begin the discussion of the algorithms implemented from the brute force technique.

TopK-NAÏVE: First we establish a connection to the database using the code from DBConnection.java file by providing the correct URL, password and username. It examines every string s in D and computes the substring edit distance d_{sub} (s, σ). To find the top-k approximate substring matches in D, we maintain a max-heap H_{Topk} storing the k strings s' with the smallest d_{sub} (s', σ) s which are used as the keys in the max-heap H_{Topk}. If the size of the heap is less than k, we just insert the string s to H_{Topk}. Otherwise, we check whether d_{sub} (s, σ) is smaller than d_{sub} (s_R, σ) of the string s_R at the root of H_{Topk}. (i.e., whether d_{sub} (s, σ) is smaller than the k-th smallest substring edit distance so far). It so, we delete the string at the root sR of H_{Topk} and insert the string s to H_{Topk}. If not, we move to the next string and repeat the above step until we encounter the last string in D. The analysis of this algorithm with respect to the values of k and with other algorithms is given in the later section.

TopK-LB: As an improvisation on TopK-Naive the next algorithm called the TopK-LB was proposed. This algorithm utilizes the q-grams of query string and the strings in D to calculate the substring edit distance. The q-grams of a string s are s [i, i+q-1] s with all 1<i<(|s|-q+1).

In accordance with the lemmas given in the paper, we calculate the lower bound for the edit distance by the method of dynamic programming. First, we find out the common q-grams between the query string and the string from the database generate a list R= {(g_1, p_1), (g_2, p_2)}. Similar to TopK-Naïve, we scan all the strings in D. However, for each string s in D, we generate the common q-gram list and sort it in the increasing order of positions where position pi of each (g_i, pi) in R is the position in s at which the q-gram qi appears. Then we compute the lower bound. If lo (d_{sub} (s, σ)) is not smaller than k-th smallest substring edit distance so far, we do not need to calculate the actual value of d_{sub} (s, σ) since s cannot be a string of the top-k approximate substring matches.

We also make evaluations by considering different values of k and show that the number of edit distance computations by using TopK-LB is less than the number of edit distance computations using TopK-NAÏVE.

TopK-SPLIT: This algorithm calculates the edit distance values by computing the lower bound similar to the TopK-LB. But the only difference here is that, instead of calculating the edit distance value between common grams, we consider the non-overlapping q-gram set G'. If G is the set of all q-grams in σ and let G' be a subset of G such that every q-gram in G' does not overlap with another q-gram in the set. We then divide the strings in D into $D^{+}_{G'}$ and $D^{-}_{G'}$, where $D^{+}_{G'}$ is the set of strings in D each of which has at least a q-gram in G' and $D^{-}_{G'}$ is $(D- D^{+}_{G'})$.

While scanning each string s, we can check easily whether the string s belongs to $D^{+}_{G'}$ or $D^{-}_{G'}$ by checking the q-grams in s. Similar to TopK-LB, when examining every string s in D one by one, TopK-SPLIT also generates the common positional q-grams R and skip the edit distance computation if the lower bound value is at least the k-th smallest substring edit distance so far. However, unlike TopK-LB, TopK-SPLIT can even skip computing the lower bound for each string as follows.

Let s_R denote the string whose substring edit distance to σ is the k-th smallest value among the strings examined so far. Among the unseen strings we can skip distance computations for the strings whose substring edit distances to σ are at least $d_{sub}(s_R, σ)$. Since we split D into $D^{+}_{G'}$ and $D^{-}_{G'}$, after $d_{sub}(s_R, σ)$ becomes at most lo $(d_{sub}(D^{-}_{G'}, σ))$, we can skip every unseen string s belonging to $D^{-}_{G'}$.

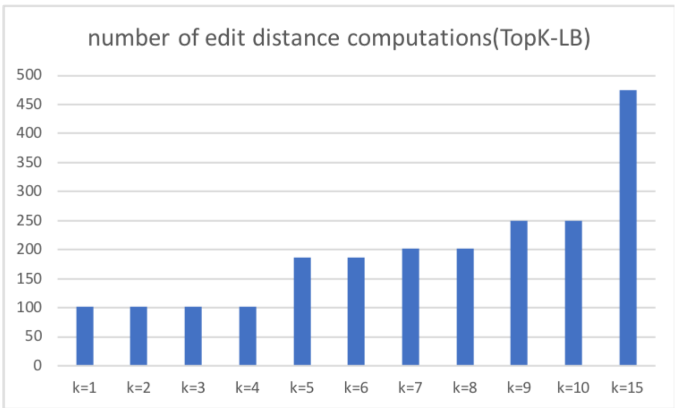
TopK-INDEX: TopK-INDEX algorithm, while scanning each string in D, also scans the posting lists of the q-grams in G together. We assume that the string id are sorted. By reading the posting lists of the q-grams in G only, we can obtain the common q-gram list R for the strings which share at least a q-gram with σ. Whenever we need to compute the actual substring edit distance, we actually access the string s stored in Since the size of each string in D is generally much larger than that of the query string, using the posting lists of the q-grams in G only speeds up the generation of R and computation of the lower bound significantly.

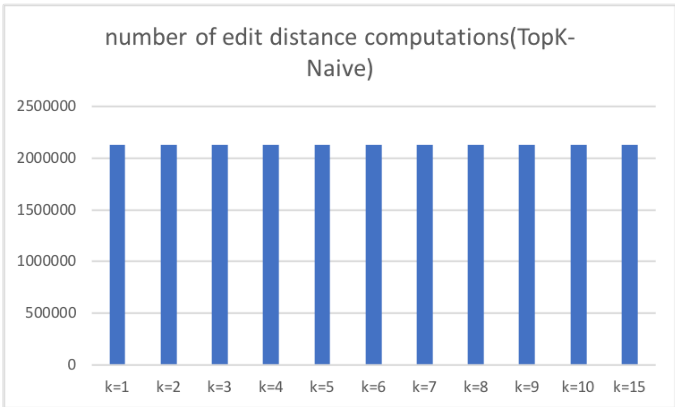
To obtain the list R in increasing order of string ids, we initially set the frontier of each posting list of a q-gram in G as the location of the first posting in the posting list. Let I be the given inverted index. The call of I. getFrontierId(G) returns the smallest string id sid_I among the ids in all frontiers, and I. getPosQgrams(sid_I) returns the common positional q-gram list R between the string of sid_I and the query string σ by reading all postings (gi, pi) from the posting lists of the q-grams in G such that gi=sid_I. The invocation of I. getPosQgrams also informs whether the string belongs to $D^{+}_{G'}$ or $D^{-}_{G'}$.

If we just follow the posting lists of the q-grams in G only to generate the list R, there may exist a string s in D which may not have any common q-gram with the query string σ and we may miss all of such strings. Thus, while we move a cursor on D together in the increasing order of string ids, if the smallest frontier id sid_I is larger than the id sid_D of D which the current cursor indicates, we read the string of sid_D from D, generate R from the actual string in D, and then perform the computations of lower bound and actual distance if necessary.

IV. EVALUATION

We have made the experimental evaluation of the number of edit distances the TopK-NAÏVE AND TopK-LB calculates by considering different values of k (1-10,15) and for different lengths of the query length. For example, if the query string is 'implementation', the length is 14 and the number of edit distances computed using TopK-NAÏVE and TopK-LB is shown in the figure below.





Here in the first graph, we can see that out of the 2128206 strings, the TopK-LB algorithm calculates the top-k values by computing the edit distances of less than 500 strings whereas for the graph of TopK-Naïve we can see that, it calculates the edit distance of every string (2128206) strings with that of the query string.

With respect to TopK-SPLIT and TopK-INDEX, we have implemented the algorithm by considering the example dataset given in the paper. The query string=" Jacksonv" and for this, we first calculate the best value of G' and then proceed to find out the lower bounds using both TopK-SPLIT and TopK-INDEX. The only difference between the two is that in TopK-INDEX we calculate the common q-grams using inverted q-gram indexes.

We have also calculated the number of lower bound computations for the query string. In case of TopK-SPLIT the number of edit distance computations is 3.

CONTRIBUTION

The workload was distributed evenly between the team members. I, Apurva Katti was assigned the work of completing the processing of the DBLP dataset along with its connection to MySQL database. I also completed the implementation of the first two algorithms TopK-NAÏVE and TopK-LB along with the experimental evaluations. Muna Tammar on the other hand was assigned the work of completing the implementation of the next two algorithms, TopK-INDEX and TopK-SPLIT.

CONCLUSION

The project undertaken to implement the efficient Top-k algorithms is completed. As a part of the future work we intend to study the minhash signature techniques mentioned in the paper and also integrate the dataset DBLP for both TopK-SPLIT as well as TopK-INDEX. We also plan to extend our implementation on the second dataset mentioned, Wikipedia.

REFERENCES

1. Y. Kim, k. Shim. Efficient Top-k Algorithms for Approximate Substring Matching. In *SIGMOD,* 2013.
2. C. Li, B. Wang, and X. Yang. VGRAM: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, 2007
3. R. Vernica and C. Li. Efficient top-k algorithms for fuzzy search in string collections. In *KEYS*, 2009.
4. Z. Yang, J. Yu, and M. Kitsuregawa. Fast algorithms for top-k approximate string matching. In *AAAI*, 2010.
5. T. Bocek, E. Hunt, and B. Stiller. Fast similarity search in large dictionaries. In *Technical Report*, 2007
6. W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit distance constraints. In *SIGMO*D, 2009.
7. M. Yu, G. Li, D. Deng, J. Feng. String similarity search and join: a survey. In *Springer-Verlag* 2016.
8. Xiao C., Wang W., Lin X., Shang H. Top-k Set Similarity Joins. *ICDE*, 2009, 916–927.