EECS 405 Final Report

for

Improving Efficiency of String Similarity Searches through Clustered and Indexed Pruning

Prepared by

James Fitzpatrick
Kyle Patterson

Version 1.0
May 5, 2014

Contents

1. Abstract

Numerous solutions exist to deal with the problems caused by data set inconsistencies, such as k-gram similarity and top-k edit distance. However, many of these solutions are limited by data size, as k-gram computation can be both time and resource intensive. With the decrease in digital storage space cost, the average size of data sets has seen rapid growth in the past decade, and frequently trumps the limitations of existing string similarity solutions. This paper proposes merging the realms of data mining and similarity queries as a means of reducing search times in large data sets while maintaining accuracy in similarity query results.

2. Introduction

String similarity search is a rapidly growing area of research within the computer science community due to its numerous overlapping applications in the modern world. From simple database joins to biological queries (such as *'Are these two genes from the same family?'*) to online social presence (*'Do these two accounts belong to the same user'*), being able to identify two objects that are highly similar to each other has proven necessary to understand the current world.

With increases in digital storage, the average query space has increased from kilobytes of space to gigabytes and, in some cases, terabytes. This query space can now contain tens of thousands to millions of entries. While exhaustive searching can be conducted in linear time, an exhaustive similarity search would require computing the edit distance between the query and every entry in at-best pseudo-quadratic time, which is no longer feasible with such large data sets.

The problem becomes much more prevalant when we consider more advanced similarity search techniques, such as k-edit pruning and k-gram searching. The former prunes the search space by removing the candidate from the search space once known edits reach some threshold k, but otherwise operating as an exhaustive search. In k-gram searching, grams of length k are constructed and stored for reference later; words with more than a predefined threshold of grams in common are considered similar and returned. While

gram searching is much more efficient than exhaustive search, it is more expensive in overhead as these grams must be constructed and stored, leading to high access cost on query execution.

One particular field of computer science, data mining, has produced several techniques to manage large data sets. Given some distance measure, it's possible to identify groups of data points, or clusters, that exhibit high similarity within the group and low similarity between different groups. Generally, Levenshtein distance the edit distance, or number of string operations to reach s_2 from a given s_1 is the metric of choice. However, clustering on solely edit distance would not yield quality groups; for example, 'hello' and 'yellow' have a Levenshtein distance of 2, as do 'cat' and 'that'. Clearly, 'hello' and 'that' do not exhibit high similarity and should not be in the same cluster.

An alternative to edit distance is the Longest Common Substring (LCS), or the longest consecutive substring that two strings have in common. By using LCS as a distance metric, we can guarantee high inter-class similarity in our clusters while reducing the number of 'false positives' in our grouping. For example, the LCS of 'hello' and 'yellow' may be clustered around 'ello', but 'hello' and 'that' have no characters in common and would not be placed in the same group.

3. Problem Definition

Let S be a complete data set consisting of strings from alphabet Σ. For each string s_i in S, we define the following:

- length(s_i) = number of characters in s_i.
  For example, length($'run'$) = 3.

- LCS(s_1,s_2) = longest common substring between s_1 and s_2.
  For example, LCS('run','fun') = 'un'.

By building a heirarchical index on the longest common substrings of the data set, we may effectively prune the search space prior to query execution by identifying a cluster within the full data set and restricting Trie construction to the located subspace. We will use two techniques to build

this heirarchical order: the BIRCH clustering algorithm and an Inverted List. Both will utilise a difference matrix of longest common substring, as calculated in Algorithm 1.

Algorithm 1 GenerateDifferenceMatrix(S)

```
1: M = new Matrix();
2: for all s_i in S do
3:    for all s_j in S do
4:       M[i,j] = LCS(s_i, s_j);
5:       if s_i equals s_j then
6:          break;
7:       end if
8:    end for
9: end for
10: return M;
```

3.1. Inverted List Structure

We define the Inverted List I as follows:

- For each element i in I, every element of i is either an InvertedList or a string
- Every InvertedList i in I has a key that equals the longest common substring of all elements in i
- When inserting a string pair s_1, s_2 into I, we recursively scan I for the key LCS(s_1, s_2). If the key is present and the strings are not yet in I, we insert s_1 and s_2 into the list with the appropriate key. If the key is not present, we construct a list containing both strings and insert it into I with key LCS(s_1, s_2).

Algorithm 2 shows how to generate an inverted list index for a given set of strings S.

Algorithm 2 Generate inverted list of LCS(S)

```
1: L = GenerateDifferenceMatrix(S);
2: M = InvertedList();
3: for i = max(length(s_1), ..., length(s_n)) to 0 do
4:    l = FindEntriesOfLength(L,i);
5:    for all (s_x, s_y) in l do
6:       if LCS(s_x, s_y) exists in M then
7:          M.Add(LCS(s_x, s_y), s_x, s_y);
8:       else
9:          M.Add(LCS(s_x, s_y), new InvertedList(LCS(s_x, s_y), s_x, s_y));
10:      end if
11:   end for
12: end for
13: return M;
```

We begin by computing the difference matrix of S, which is an $O(n^2)$ operation, and storing the matrix in L. We then loop through the matrix, seeking entries of decreasing size. These entries are stored as coordinates of string pairs, where the ID of each string would yield the matrix position of the LCS. For each pair, we check if the key (i.e, the longest common substring of the string pair) already exists within our inverted list; if it does, we append each string of the pair into the proper list. If the key is not yet present, we create a new List for the string pair and insert it into the inverted list using the computed key.

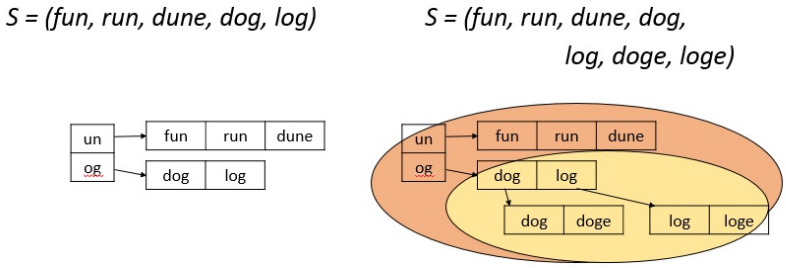

Figure 1: Two examples of an inverted list construction. In example 2, we construct all inverted lists with length(LCS) = 3 first, in yellow, then insert into an inverted list with length(LCS) = 2, in red.

4. Related Work

4.1. String Similarity

4.1.1. Top-K Search

Top-K Search, a search utilized in String Similarity, computes the top k results by comparing a query string to a dataset and finds the k elements that have the smallest edit distance from this string. However, this algorithm can become computationally expensive as the algorithm requires comparing the query to all strings in a dataset. To reduce the number of necessary computations, we implement a version of the pivotal Top-K Search as described by Deng et al. A trie-based approach can take advantage of the similarity strings. Each entry in the trie represents a character, save the root which is the null character. The path from the root to a leaf is a inorder representation of that string. This structure redcudes the number of strings we need to store in full by taking advantage of strings with similar prefixes. By using this trie, we can incrementally search the entire search space simultaneously utilizing edit distance measurements. Once k strings have been found, we have our strings to return.

4.1.2. Positional VGRAM Search

Gram based appraoches to string similarity have been discussed for quite some time. By assembling a inverted list of grams, we can divide a query into grams of size n to compare it to the strings of a dataset. The strings that share the most grams with this string should be considered to be "closer" to the query than others. Many string comparison algorithms consider grams of set lengths. However, there are many advantages and disadvantages to having any particular length of n-gram. Rather than to have a one-for-all solution to the strings, the authors propose a solution to generate grams of variable length between a minimum and maximum length based on a set gram dictionary, which is constructed based on the data collection of strings prior to the query. When computing the gram dictionary, we should only store longer grams that have a frequency above a certain threshold. The algorithm utilizes a trie to reduce the computations of grams that are substrings of longer grams, since the subgrams will necessarily occur when the larger grams are present. Since smaller grams are present in larger ones, the grams that survive pruning will more often than not be shorter grams.

Longer grams will be present in data sets only if they are very frequent. By this pruning, we can reduce our gram set to the grams that are most helpful in indentifying desired strings. Further, the position of these grams may be of importance to certain datasets. In some sets, a particular gram may occur at the beginning of the ending of a string, which will identify strings more accurately. By adding positional information, we can also reduce our search space.

4.2. BIRCH Clustering Algorithm

BIRCH is a well-known agglomerative hierarchical clustering algorithm, presented first in [6]. BIRCH, or Balanced Iterative Reducing and Clustering using Hierarchies, is the current leading algorithm in hierarchical clustering techniques and has several advantages. First, and most notably, BIRCH only requires one pass through the database; second, BIRCH utilises a CF-Tree, similar to a B+ Tree, that may take advantage of paging and external memory. However, it still runs in $O(n^2 \log n)$ time complexity, and requires the computation of a difference matrix during running time to detect the distances between clusters.

5. Methodology

5.1. Data preparation

Data was mirrored from the Database and Logic Programming bibliography (DBLP) and International Movie Database (IMDB) public sources to a MySQL server hosted on a micro instance of Amazon Elastic Cloud Computing (EC2). Data was cleaned to remove any incomplete entries, and trimmed to remove any entries that contained any non-standard ASCII characters (i.e, characters whose value is greater than 127).

In total, the IMDB database contained 55,743 movie entries while the DBLP database contained 2,502,896 publication entries.

5.2. Development Machines

Tries and tests were constructed and executed on a 64-bit Dell Latitude computer with 8 GB of RAM and an i7-2720QM quad-core processor operating at 2.20 GHz. Algorithms were developed using C# and executed in the Visual Studio 2013 v12 Runtime Environment.

5.3. Testing

Tries were evaluated using the following method:

1. Generate query structure for a randomized subset of data
2. Select random values from data subset and randomly edit values within k edits
3. Query trie structure for edited values

Tests were executed for data subsets in increments of 10,000 from 10,000 elements to the full data set, with 30 queries executed per trie. Query structures used include VGRAM and Top-K Tries.

6. Results

6.1. Index generation

To begin with, computation of the difference matrix is very costly in both time and memory. As an $O(n^2)$ operation, it took approximately 2.5 hours to compute the matrix on the IMDB data set, and the matrix returned was over 3.8 GB in size. Computation of the matrix for the DBLP data set was terminated after 20 hours, truncating at 0.5 (one-half) million entries of the 2.5 million total, and was over 8 GB in size.

One large disadvantage to the BIRCH clustering algorithm is that it requires the difference matrix to be loaded into main memory at computation; further, BIRCH operates in $O(n^2 \log n)$ time. Thus, we could not accurately construct a heirarchical clustering on either data set, given the computational resources and time available.

Construction of the Inverted List was also time consuming, taking as long as a 1.5 hours for the truncated DBLP data set. Searching was also costly on the Inverted List, which was stored primarily in external memory due to the large data size.

6.2. Trie generation

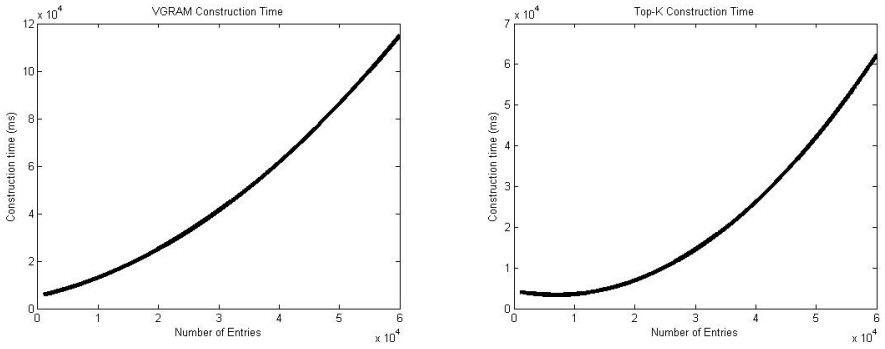

Figure 2: Construction times for VGRAM and Top-K per data set sizes.

As expected, Trie construction quickly grows in time cost as the size of S increases. VGRAM times were computed by merging together the IMDB and DBLP data sets; Top-K could not be constructed on the DBLP data set. By pruning the search space prior to Trie construction, we can restrict the time cost at this stage; however, query execution must be kept minimal to account for the search cost within the index.

6.3. Query executions

As seen in figure 3 on the following page, queries for VGRAM were magnitudes of order faster on IMDB than Top-K. This is largely due to the fundamental difference in technique; computation of edit distance within long strings in Top-K was much more expensive than computation of variable-length grams in VGRAM. Search time was only marginally reduced in a pruned Trie.

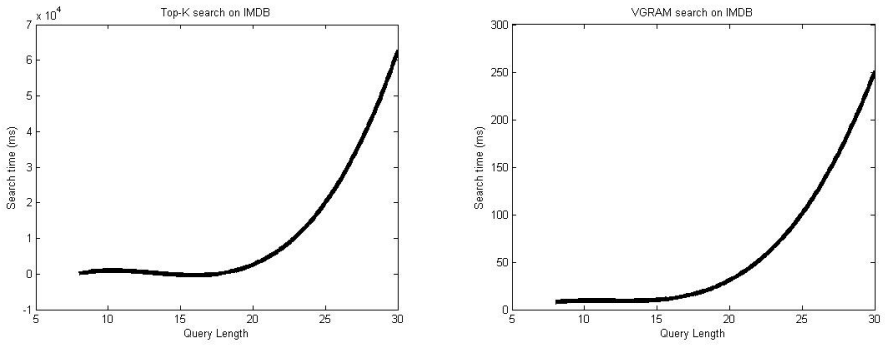

Figure 3: Query times per IMDB and DBLP data sets using the VGRAM Trie.

7. Conclusions

In this paper, we presented pruning techniques that rely on common data mining and indexing techniques such as the BIRCH heirarchical clustering algorithm and inverted lists. Top-K and VGRAM similarity search algorithms were implemented and tested with the pruning techniques; however, due to the computational cost associated with indexing by longest common substring, we conclude that it is only beneficial in certain situations, such as on the DBLP which had a high data count (over 2.5 million) with long string lengths, and not recommended in most cases.

8. References

1. Don Deng, Guoliang Li, Jianhua Feng, Wen-Syan Li, Top-k String Similarity Search with Edit-Distance Constraints, ICDE 2013.

2. C. Li, B. Wang, X, Yang. VGRAM: Improving Performance of Approximate Queries on String Collections Using Variable-Length Grams. VLDB, September 23-27, 2007.

3. M. Hadjieleftheriou, N. Koudas, and D. Srivastava. Incremental maintenance of length normalized indexes for approximate string matching. *SIGMOD Conference*, pages 429-440, 2009.

4. T. Kahveci and A. K. Singh. Efficient index structures for string databases. VLDB, pages 351-360, 2001.

5. Z. Yang, J. Yu and M. Kitsuregawa. Fast algorithms for top-k approximate string matching. AAAI, 2010.

6. T. Zhiang, R. Ramakrishnan and M. Livny. BIRCH: An Efficient Data Clustering Method for Very Large Databases. SIGMOD, 1996.