Calculating Edit Distance for Large Sets of String Pairs using MapReduce

Shagun Jhaver
Dept. of Computer Science
The University of Texas at Dallas
Richardson, Texas 75080
Email: sxj124330@utdallas.edu

Latifur Khan
Dept. of Computer Science
The University of Texas at Dallas
Richardson, Texas 75080
Email: lkhan@utdallas.edu

Bhavani Thuraisingham
Dept. of Computer Science
The University of Texas at Dallas
Richardson, Texas 75080
Email: bhavani.thuraisingham@utdallas.edu

***Abstract*—Given two strings X and Y over a finite alphabet, the edit distance between X and Y, $d(X,Y)$ is the number of elementary edit operations required to edit X into Y. A dynamic programming algorithm elegantly computes this distance. In this paper, we investigate the parallelization of calculating edit distance for a large set of strings using MapReduce, a popular parallel computing framework. We propose SIM_MR and PRE_MR algorithms, parallel versions of the dynamic programming solution, and present implementations of these algorithms. We study different cases by varying algorithm parameters, input size and number of parallel nodes, and analytically and experimentally confirm the superiority of our methods over the usual dynamic programming approach. This study demonstrates how MapReduce parallelization opens new avenues of designing for dynamic programming algorithms.**

Index Terms - Edit distance, Levenshtein distance, MapReduce, string manipulation, dynamic programming

I. Introduction

Given two strings s and t, the minimum number of edit operations required to transform s into t is called the edit distance. The edit operations commonly allowed for computing edit distance are: (i) insert a character into a string; (ii) delete a character from a string and (iii) replace a character of a string by another character. For these operations, edit distance is sometimes called Levenshtein distance [1]. For example, the edit distance between 'tea' and 'pet' is 2.

There are a number of algorithms that compute edit distances [2], [3], [4] and solve other related problems [5], [6], [7]. Edit distance has placed an important role in a variety of applications due to its computational efficiency and representational efficacy. It can be used in approximate string matching, optical character recognition, error correcting, pattern recognition [8], redisplay algorithms for video editors, signal processing, speech recognition, analysis of bird songs and comparing genetic sequences [9]. Sankoff and Kruskal provide a comprehensive compilation of papers on the problem of calculating edit distance [12].

The cost of computing edit distance between any two strings is roughly proportional to the product of the two string lengths. This makes the task of computing the edit distance for a large set of strings difficult. It is computationally heavy and requires managing large data sets, thereby calling for a parallel processing implementation. MapReduce, a general-purpose programming model for processing huge amounts of data with a parallel, distributed algorithm appears to be particularly well adapted to this task. This paper reports on the application of MapReduce, using its open source implementation Hadoop to develop a computational procedure for efficiently calculating edit distance.

The edit distance is usually computed by an elegant dynamic programming procedure [1]. Although, like the divide-and-conquer method, dynamic programming solves problems by combining the solutions to subproblems, it applies when the subproblems overlap - that is, when subproblems share subsubproblems [11]. Each subsubproblem is solved just once, and then the answer is saved, thereby avoiding the work of recomputing the answer every time it solves each subproblem. Unlike divide-and-conquer algorithms, dynamic programming procedures do not partition the problem into disjoint subproblems, therefore edit distance calculation does not lend itself naturally to parallel implementation. This paper develops an algorithm for calculating the edit distance for MapReduce framework and demonstrates the improvement in performance over the usual dynamic programming algorithm used in parallel.

We implement the dynamic programming approach for this problem in a top-down way with memoization [11]. In this approach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblem in an associative array. The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level; if not, the procedure computes the value in the usual manner [11]. Finding edit distance of a pair of strings (s, t) entails finding the edit distance of every pair (s', t'), where s' and t' are substrings of s and t respectively. All these distances are saved in an associative array h. Subsequently, if any new pair of strings share a pair of substrings for which the distance is already stored in h, the saved values are used, thereby saving the computation time. Pairs of strings that are likely to share common substrings are processed together, thus improving the performance over the standard dynamic programming parallel application for this problem.

The contributions of this work are as follows. First, to the best of our knowledge, this is the first work that addresses the calculation of unnormalized edit distance for a large number of string pairs in a parallel implementation. Our implementation in MapReduce improves upon the performance of usual dynamic programming implementation on a single machine.

Second, our proposed approach, which uses an algorithm tailored to the MapReduce framework architecture performs better than the simple parallel implementation. Finally, this serves as an example of using the MapReduce framework for dynamic programming solutions, and paves the way for parallel implementation for other dynamic programming problems.

In particular, the requirement for calculating edit distance for a large number of pairs of strings emerged in one of our previous research projects [36] on finding normative patterns over dynamic data streams. This project uses an unsupervised sequence learning approach to generate a dictionary which will contain any combination of possible normative patterns existing in the gathered data stream. A technique called compression method (CM) is used to keep only the longest and most frequent unique patterns according to their associated weight and length, while discarding other subsumed patterns. Here, edit distance is required to find the longest patterns.

The remainder of this paper is organized as follows. Section II discusses the problem statement and the dynamic programming solution to the problem on a single machine. Section III discusses our proposed approach, and the techniques used in detail. Section IV reports on the experimental setup and results. Section V then describes the related work, and Section VI concludes with directions to future work.

II. Background

The edit distance problem is to determine the smallest number of edit operations required for editing a source string of characters into a destination string. For any two strings $s = s_1 s_2 .... s_m$ and $t = t_1 t_2 ... t_n$ over an input alphabet of symbols $\sigma = \{a_1, a_2, ... a_r\}$, the valid operations to transform s into t are:

- Insert a character t_j appearing in string t
- Delete a character s_i appearing in string s
- Replace a character s_i appearing in string s by a character t_j in string t

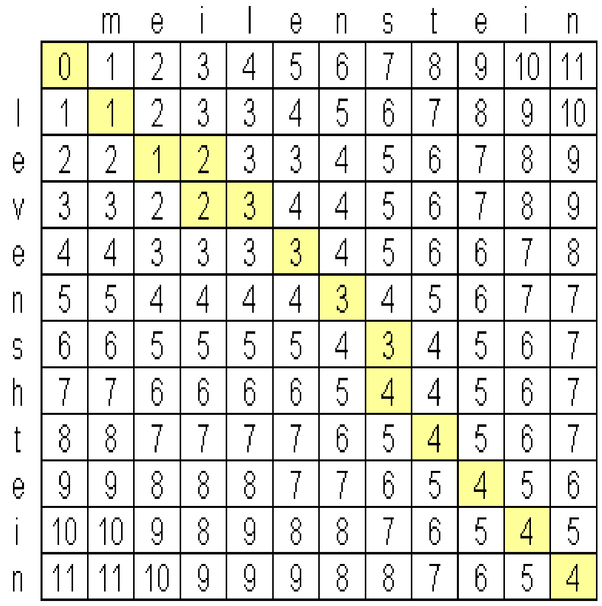
For strings $s = s_1 s_2 .... s_m$ and $t = t_1 t_2 ... t_n$, and an associative array h storing the edit distance between s and t, this problem can be solved sequentially in $O(mn)$ time. The memoized dynamic programming algorithm for this, MEM_ED, is described in Algorithm 1:

Algorithm 1 EDIT-DISTANCE($s[1, 2, ..m]$, $t[1, 2, ..., n]$, h): **(MEM_ED)**

```
1:  if pair($s$, $t$) in $h$ then
2:      return $h$[pair($s$, $t$)]
3:  end if
4:  if len($s$)==0 then
5:      return $t$.length
6:  end if
7:  if len($t$)==0 then
8:      return $s$.length
9:  end if
10: $s' \leftarrow s[1, 2, ...m-1]$
11: $t' \leftarrow t[1, 2, ...n-1]$
12: $k_a \leftarrow$ EDIT-DISTANCE$(s', t')$
13: $k_b \leftarrow$ EDIT-DISTANCE$(s', t)$ + 1
14: $k_c \leftarrow$ EDIT-DISTANCE$(s, t')$ + 1
15: if s[m]==t[n] then
16:     $k_d \leftarrow k_a$
17: else
18:     $k_d \leftarrow k_a$ + 1
19: end if
20: $c \leftarrow min(k_b, k_c, k_d)$
21: $h$[pair($s$, $t$)] $\leftarrow c$
22: return $c$
```

For an input pair of strings (s, t), step 1 in MEM_ED algorithm checks whether the pair is already stored in the input associative array h. If present, the algorithm returns the stored value for (s, t) in step 2. If one of the strings is empty, MEM_ED returns the length of the other string as the output. Steps 10-11 in this algorithm divide the problem inputs into subproblem inputs of smaller size. Steps 12 - 14 calculate the edit distance recursively for these subproblems. Step 20 derives the edit distance for the problem, and step 21 stores this result in an associative array, h for further use, thereby memoizing the recursive procedure.

Fig. 1 shows the associative array entries for calculating the edit distance between two strings - 'levenshtein' and 'meilenstein'. For example, for calculating the edit distance between the string pair ('*levens*', '*meilens*'), the edit distances k_a, k_b and k_c for the pairs ('*leven*', '*meilen*'), ('*levens*', '*meilen*') and ('*leven*', '*meilens*') are considered respectively. By a recursive procedure in steps 12-14 of MEM_ED, these values are calculated to be 3, 4 and 4 respectively. Since the input string pair ('*levens*', '*meilens*') have the same last character '*s*', the value k_d is calculated to be equal to k_a = 3 in steps 15-19 of MEM_ED. Step 20 computes c, the minimum of k_b, k_c and k_d to be 3. Step 21 associates string pair ('*levens*', '*meilens*') with value 3 in the associative array h for further use. Step 22 returns this edit distance value.



		m	e	i	l	e	n	s	t	e	i	n
	0	1	2	3	4	5	6	7	8	9	10	11
l	1	1	2	3	3	4	5	6	7	8	9	10
e	2	2	1	2	3	3	4	5	6	7	8	9
v	3	3	2	2	3	4	4	5	6	7	8	9
e	4	4	3	3	3	3	4	5	6	6	7	8
n	5	5	4	4	4	4	3	4	5	6	7	7
s	6	6	5	5	5	5	4	3	4	5	6	7
h	7	7	6	6	6	6	5	4	4	5	6	7
t	8	8	7	7	7	7	6	5	4	5	6	7
e	9	9	8	8	8	7	7	6	5	4	5	6
i	10	10	9	8	9	8	8	7	6	5	4	5
n	11	11	10	9	9	9	8	8	7	6	5	4

Fig. 1: Edit Distance between two strings

On a single machine, we compute the edit distance for every pair of distinct strings in an input text document by repeatedly using MEM_ED for each pair of distinct strings. The SIN_ED procedure in Algorithm 2 describes this approach.

Algorithm 2 Single Machine Implementation for calculating Edit Distance for all string pairs (**SIN_ED**)

```
1: $dist\_strings \leftarrow$ list of distinct strings in doc d
2: for all string pairs ($s$, $t$) $\epsilon$ $dist\_strings$ do
3:     $H \leftarrow$ new ASSOCIATIVE_ARRAY
4:     $c \leftarrow$ EDIT_DISTANCE($s$, $t$, $H$)
5:     EMIT(pair($s$, $t$), $c$)
6: end for
```

Step 1 in SIN_ED algorithm collects all the distinct strings in the input document. Step 3 initializes an associative array. Step 4 uses the EDIT_DISTANCE procedure of MEM_ED to calculate the edit distance for each distinct string pair. The implementation of SIN_ED takes $O(t^2n^2)$ time for t distinct strings and string length of order n. This is computationally very expensive; hence we need to implement this algorithm in parallel for faster computations.

III. Related Work

Extensive studies have been done on edit distance calculations and related problems over the past several years. Ristad and Yianilos [15] provide a stochastic model for learning string edit distance. This model allows for learning a string edit distance function from a corpus of examples. Bar-yossef, Jayram, Krauthgamer and Kumar develop algorithms that solve gap versions of the edit distance problem [16]: given two strings of length n with the promise that their edit distance is either at most k or greater than l, these algorithms decide which of the two holds.

A lot of studies have been dedicated to normalized edit distance to effect a more reasonable distance measure. Abdullah N. Arslan and Ömer Egecioglu discuss a model for computing the similarity of two strings X and Y of lengths m and n respectively where X is transformed into Y through a sequence of three types of edit operations: insertion, deletion, and substitution. The model assumes a given cost function which assigns a non-negative real weight to each edit operation. The amortized weight for a given edit sequence is the ratio of its weight to its length, and the minimum of this ratio over all edit sequences is the normalized edit distance. Arslan and Egecioglu [18] give an O(mn logn)-time algorithm for the problem of normalized edit distance computation when the cost function is uniform, i.e, the weight of each edit operation is constant within the same type, except substitutions which can have different weights depending on whether they are matching or non-matching.

Jie Wei proposes a new edit distance called Markov edit distance [17] within the dynamic programming framework, that takes full advantage of the local statistical dependencies in the string/pattern in order to arrive at enhanced matching performance. Higuera and Micó define a new contextual normalized distance, where each edit operation is divided by the length of the string on which the edit operation takes place. They prove that this contextual edit distance is a metric and that it can be computed through an extension of the usual dynamic programming algorithm for the edit distance [20].

Fuad and Marteau propose an extension to the edit distance to improve the effectiveness of similarity search [19]. They test this proposed distance on time series data bases in classification task experiments and prove, mathematically, that this new distance is a metric.

Robles-Kelly and Hancock compute graph edit distance by converting graphs to string sequences, and using string matching techniques on them [24]. They demonstrate the utility of the edit distance on a number of graph clustering problems. Bunke introduces a particular cost function for graph edit distance and shows that under this cost function, graph edit distance computation is equivalent to the maximum common subgraph problem [21].

Hanada, Nakamura and Kudo discuss the issue of high computational cost of calculating edit distance of a large set of strings [22]. They contend that a potential solution for this problem is to approximate the edit distance with low computational cost. They list the edit distance approximation methods, and use the results of experiments implementing these methods to compare them. Jain and Rao present a comparative study to evaluate experimental results for approximate string matching algorithms such as Knuth-Morris-Pratt, Boyer-Moore and Raita on the basis of edit distance [23].

A few studies have also been done that target a parallel implementation of calculating normalized edit distance. Instead, in this work, we address the calculation of unnormalized edit distance for large number of string pairs in a parallel implementation, and we use MapReduce for it.

IV. Proposed Approach

We discussed in the Background section that the single machine implementation for calculating the edit distance of all distinct pairs of strings, described in SIN_ED, is computationally expensive. We propose a parallel computing approach to do this more efficiently.

MapReduce is emerging as an important programming model for expressing distributed computations in data-intensive applications [30]. It was originally proposed by Google and is built on well-known principles in parallel and distributed processing dating back several decades. MapReduce has since enjoyed widespread adoption via Hadoop, a popular open-source implementation developed primarily by Yahoo and Apache. It enables easy development of scalable approaches to efficiently processing massive amounts of data on clusters of commodity machines. MapReduce systems are evolving and extending rapidly and today, Hadoop is a core part of the computing infrastructure for many web companies, such as Facebook, Amazon, Yahoo and Linkedin. Because of its high efficiency, high scalability, and high reliability, MapReduce framework is used in many fields [30], such as life science computing [32], text processing, web searching, graph processing [34], relational data processing, data mining, machine learning [35] and video analysis [33].

We use the MapReduce framework for the parallel implementation of calculating edit distance for a large set of strings. The idea is to use the associative array in SIN_ED to store the edit distances across the computations for many pairs of strings. Once the edit distance for a pair of strings (s, t) is calculated, the edit distance for all pairs (s', t'), where s' and t' are substrings of s and t respectively are stored in the associative array. Subsequent to this, for a new pair of strings (a, b), the calculations at steps 12, 13 and/or 14 in MEM_ED can be saved, if the input pairs of strings for these steps already have an entry in the associative array.

The SIM_MR algorithm (Algorithm 3) describes a simple Map Reduce approach to calculating edit distance in parallel using these ideas.

Algorithm 3 Simple MapReduce approach to calculating Edit Distance for all string pairs (**SIM_MR**)

```
1: class MAPPER
2:     method MAP(docid $a$, doc $d$)
3:         $dist\_strings \leftarrow$ list of distinct strings in doc d
4:         $count \leftarrow 0$
5:         for all string pairs ($s$, $t$) $\epsilon$ $dist\_strings$ do
6:             $count \leftarrow count + 1$
7:             $reducer\_index \leftarrow count$ % $num\_reducers$
8:             EMIT($reducer\_index$, pair($s$, $t$))
9:         end for
10:
11: class REDUCER
12:     method REDUCE($reducer\_index$, pairs [($s_1$, $t_1$), ($s_2$, $t_2$),...])
13:         $H \leftarrow$ new ASSOCIATIVE_ARRAY
14:         for all string pairs ($s$, $t$) $\epsilon$ pairs [($s_1$, $t_1$), ($s_2$, $t_2$),...] do
15:             $c \leftarrow$ EDIT_DISTANCE($s$, $t$, $H$)
16:             EMIT(pair($s$, $t$), $c$)
17:         end for
```

SIM_MR first constructs a list of distinct strings from the input document in Step 3 in the Mapper phase. The '$count'$' variable initialized in Step 4 tracks the count of the string pair being processed. The '$num\_reducers$' parameter determines the number of reducers to be used in the reduce phase of the procedure. The '$reducer\_index'$' variable determines the reducer that would process the current string pair. The value of '$reducer\_index'$' is calculated in Step 7. This value is independent of the strings in the string pair being processed. Step 8 emits with $'reducer\_index'$ as the key and the current string pair as the value.

In the reduce phase, an associative array, H is initialized in Step 13. For every input string pair, Step 15 calculates the edit distance of the current string pair using H with the EDIT_DISTANCE procedure of MEM_ED. The entries stored in H during the edit distance calculations of any string pair can be used across calculations for different string pairs. Step 16 emits the string pair with its corresponding edit distance value.

We note that the $reducer\_index$ value in SIM_MR depends just on the count of the string pair being processed. We propose a modified algorithm that uses the strings in the string pair to effect a more efficient way of determining the reducer where the current string pair gets processed.

The pairs of strings to be processed at a single node need to be chosen such that they are likely to have some pairs of substrings for which the edit distance has already been computed, and the computation time is saved via an associative array look-up. To accomplish this, we collect all pairs of strings with a common prefix pair at a single reducer node. This prefix pair is constructed by taking the first $prefix\_length$ characters of both strings to form a string pair. The procedure for the proposed approach, PRE_MR, is described in Algorithm 4.

Algorithm 4 Prefixed MapReduce approach to calculating Edit Distance for all string pairs (**PRE_MR**)

```
1: class MAPPER
2:     method MAP(docid $a$, doc $d$)
3:         $dist\_strings \leftarrow$ list of distinct strings in doc d
4:         for all string pairs ($s$, $t$) $\epsilon$ $dist\_strings$ do
5:             $s\_prefix \leftarrow s[1 : prefix\_length]$
6:             $t\_prefix \leftarrow t[1 : prefix\_length]$
7:             EMIT(pair($s\_prefix$, $t\_prefix$), pair($s$, $t$))
8:         end for
9:
10: class REDUCER
11:     method REDUCE(prefix_pair ($s'$, $t'$), pairs [($s_1$, $t_1$), ($s_2$, $t_2$),...])
12:         $H \leftarrow$ new ASSOCIATIVE_ARRAY
13:         for all string pairs ($s$, $t$) $\epsilon$ pairs [($s_1$, $t_1$), ($s_2$, $t_2$),...] do
14:             $c \leftarrow$ EDIT_DISTANCE($s$, $t$, $H$)
15:             EMIT(pair($s$, $t$), $c$)
16:         end for
```

For the current string pair (s, t), Steps 5 and 6 in PRE_MR calculate the s_prefix and t_prefix values by taking the first $prefix\_length$ characters from s and t respectively. For example, for $prefix\_length$ = 2, and string pair (s, t) = ('mango', 'gate'), the s_prefix and t_prefix values are computed to be 'ma' and 'ga' respectively. Step 7 emits with the string pair (s_prefix, t_prefix) as the key, and the string pair (s, t) as the value.

The reduce phase for PRE_MR is similar to the reduce phase in the SIM_MR procedure. An associative array H is initialized in step 12, and the edit distance of every string pair in pairs [(s_1, t_1), (s_2, t_2),...] is computed using the EDIT_DISTANCE procedure of MEM_ED. Step 15 emits the results.

Hadoop runs its map and reduce processes in such a way that these processes operate on independent chunks of data and have no inter process communication. We've customized our algorithms to satisfy this constraint. For PRE_MR, the mapper sends all string pairs sharing the same pair of prefixes to a single reducer. In the reduce stage, all these string pairs are processed together. For each string pair, the associative array H saves the calculated intermediate edit distances, and a look-up in this array often saves the computations for many other pairs of strings that are input to this reducer. These savings in computations make our algorithms, especially PRE_MR more efficient than the SIN_ED approach.

Fig. 2 shows an example of the implementation for PRE_MR algorithm with '$prefix\_length$' = 2. Mapper constructs a prefix pair ('ma', 'la') for input pair of strings ('mad', 'laughter'), and emit with ('ma', 'la') as the key and

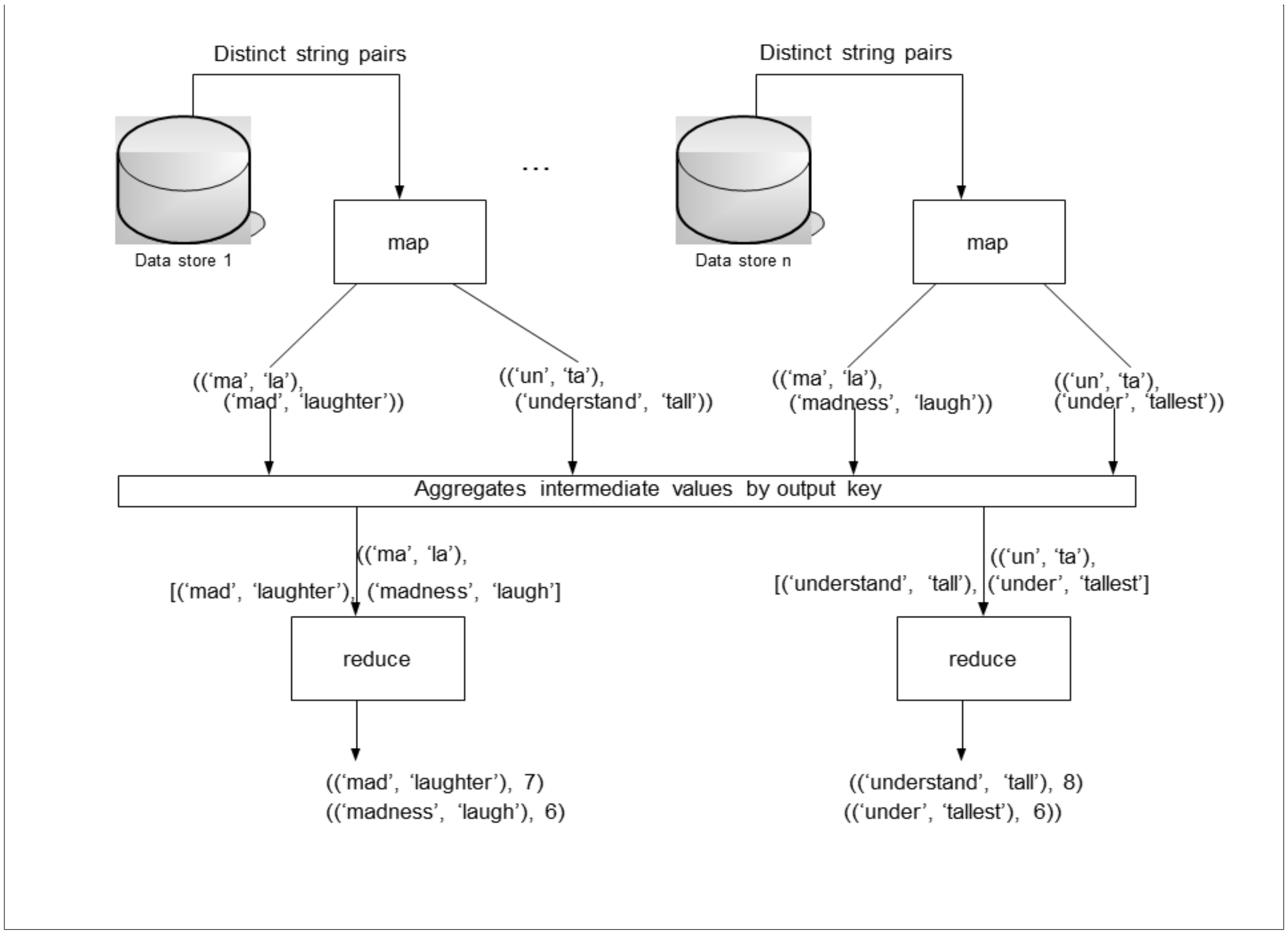

Distinct string pairs
Data store 1
map
...
Distinct string pairs
Data store n
map
((‘ma’, ‘la’), (‘mad’, ‘laughter’))
((‘un’, ‘ta’), (‘understand’, ‘tall’))
((‘ma’, ‘la’), (‘madness’, ‘laugh’))
((‘un’, ‘ta’), (‘under’, ‘tallest’))
Aggregates intermediate values by output key
((‘ma’, ‘la’), [(‘mad’, ‘laughter’), (‘madness’, ‘laugh’]
((‘un’, ‘ta’), [(‘understand’, ‘tall’), (‘under’, ‘tallest’]
reduce
reduce
((‘mad’, ‘laughter’), 7)
((‘madness’, ‘laugh’), 6)
((‘understand’, ‘tall’), 8)
((‘under’, ‘tallest’), 6))

Fig. 2: PRE_MR algorithm flow-chart

(‘mad’, ‘laughter’) as the value. In the reduce phase, all strings pairs sharing the prefix (‘ma’, ‘la’) are processed together. Therefore, the string pairs (‘mad’, ‘laughter’) and (‘madness’, ‘laugh’) are processed at the same node. Since these string pairs share common substrings, many computations are saved, and the procedure is faster.

V. Experimental Setup and Results

Our hadoop cluster (cshadoop0-cshadoop9) has ten virtual machines that run in the Computer Science vmware esx cloud. Each of these VM’s has 4 GB of RAM and a 256 GB virtual hard drive. These VM’s are spread across three ESX hosts to balance the load. We’ve used one name node and nine slave nodes. For our implementation, we used Hadoop version 1.0.4 and JAVA JDK version 1.6.0.37.

The data sets were created from the ebooks for which the copyright has expired. We used the text of ‘Pride and Prejudice’ by Jane Austen available at http://www.gutenberg.org/ebooks/1342, and developed files of size 10kB, 20kB,..., 100kB from it.

We implemented a preprocessing step for each of the experiments, where all the duplicate strings in the input files were eliminated, thus all the experiments described have been conducted on unique strings.

We processed each of these files using SIN_ED, SIM_MR and PRE_MR algorithms. The results are described in Table I and Fig. 3. It shows the comparison of the performance of neutral baseline of SIN_ED implementation (plain sequential implementation) with our proposed algorithms. For Fig. 3, we’ve taken the input file sizes (in kB) on the x-axis and the times taken by each of the procedures (in seconds) on the y-axis. These results are obtained using 4 reducer nodes.

TABLE I: SIN_ED vs. SIM_MR vs. PRE_MR implementation

File Size	SIN_ED	SIM_MR	PRE_MR
10 kB	12 sec	72 sec	68 sec
20 kB	33 sec	73 sec	70 sec
30 kB	62 sec	82 sec	71 sec
40 kB	90 sec	94 sec	76 sec
50 kB	122 sec	147 sec	79 sec
60 kB	155 sec	120 sec	80 sec
70 kB	189 sec	125 sec	85 sec
80 kB	218 sec	140 sec	88 sec
90 kB	276 sec	145 sec	93 sec
100 kB	293 sec	209 sec	101 sec

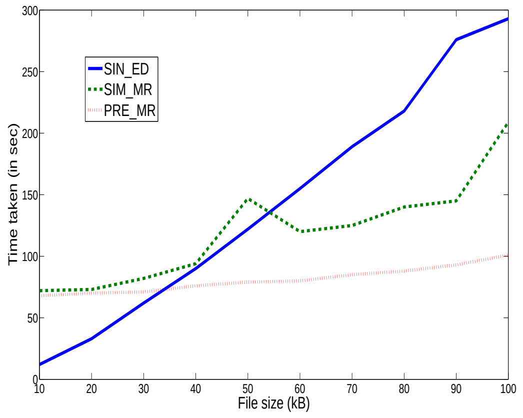


Fig. 3: SIN_ED vs. SIM_MR vs. PRE_MR implementation

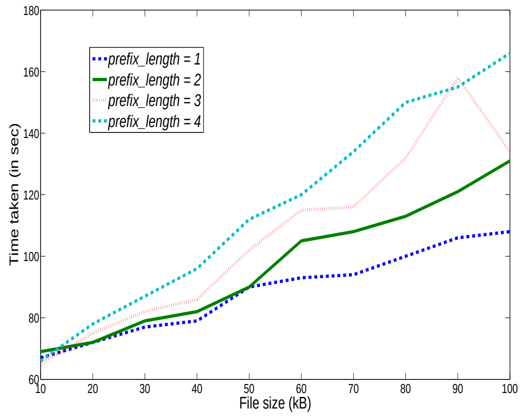


Fig. 4: PRE_MR performance for different $prefix\_length$ values

Table I results indicate that PRE_MR algorithm gives the best results. For example, for a file of size 80 kB, SIN_ED takes 218 sec, SIM_MR takes 140 sec and PRE_MR algorithm takes 88 sec. Therefore, we conduct the rest of the experiments only for PRE_MR.

We experimented with different values of the parameter '$prefix\_length'$ used in the MAP phase for the PRE_MR implementation. The time taken for different file sizes are documented in Table II, and Fig. 4. For Fig. 4, the x-axis is file size (in kB), and the y-axis is the runtime for experiments with different '$prefix\_length$' values. For this experiment, we chose to use 2 mappers and 1 reducer in each case. We see that, generally, smaller '$prefix\_length$' values tend to give better performance. For example, for a file of size 80 kB, '$prefix\_length$' = 1 case takes 100 sec, '$prefix\_length$' = 2 case takes 113 sec, '$prefix\_length$' = 3 case takes 132 sec and '$prefix\_length$' = 4 case takes 150 sec.

TABLE II: PRE_MR performance for different $prefix\_length$ values

File Size	*prefix_length*=1	*prefix_length*=2	*prefix_length*=3	*prefix_length*=4
10 kB	67 sec	69 sec	65 sec	66 sec
20 kB	72 sec	72 sec	75 sec	78 sec
30 kB	77 sec	79 sec	82 sec	87 sec
40 kB	79 sec	82 sec	86 sec	96 sec
50 kB	90 sec	90 sec	102 sec	112 sec
60 kB	93 sec	105 sec	115 sec	120 sec
70 kB	94 sec	108 sec	116 sec	134 sec
80 kB	100 sec	113 sec	132 sec	150 sec
90 kB	106 sec	121 sec	158 sec	155 sec
100 kB	108 sec	131 sec	134 sec	166 sec

We also experimented with different number of reducers in the PRE_MR implementation for three cases: '$prefix\_length$' = 1, '$prefix\_length$' = 2 and '$prefix\_length$' = 3. In each case, in the corresponding Fig., we take the file size as the x-axis and the runtime for the experiment as the y-axis.

Table III and Fig. 5 detail the times taken for this experiment when the '$prefix\_length$' parameter is set to 1. We see that the performance generally improves with increasing number of reduce nodes. For example, for a file of size 80 kB, 1 reducer node takes 100 sec, 2 reducers take 91 sec and 4 reducers take 88 sec.

TABLE III: PRE_MR performance for different number of reducers, $prefix\_length$=1

File Size	1 reducer	2 reducers	4 reducers
10 kB	67 sec	65 sec	68 sec
20 kB	72 sec	68 sec	70 sec
30 kB	77 sec	69 sec	71 sec
40 kB	79 sec	75 sec	76 sec
50 kB	90 sec	86 sec	79 sec
60 kB	93 sec	95 sec	80 sec
70 kB	94 sec	90 sec	85 sec
80 kB	100 sec	91 sec	88 sec
90 kB	106 sec	96 sec	93 sec
100 kB	108 sec	112 sec	101 sec

Table IV and Fig. 6 describe the times taken when the '$prefix\_length$' parameter in PRE_MR is set to 2. For example, for a file of size 90 kB, 1 reducer node takes 121 sec, 2 reducers take 117 sec, and 4 reducers take 102 sec.

TABLE IV: PRE_MR performance for different number of reducers, $prefix\_length$=2

File Size	1 reducer	2 reducers	4 reducers
10 kB	69 sec	63 sec	67 sec
20 kB	72 sec	66 sec	71 sec
30 kB	79 sec	72 sec	73 sec
40 kB	82 sec	76 sec	82 sec
50 kB	90 sec	81 sec	78 sec
60 kB	105 sec	91 sec	90 sec
70 kB	108 sec	98 sec	89 sec
80 kB	113 sec	97 sec	97 sec
90 kB	121 sec	117 sec	102 sec
100 kB	131 sec	111 sec	101 sec

Table V and Fig. 7 list the times taken for PRE_MR when the 'prefix_length' parameter is set to 3. Again, increasing the number of nodes in reduce phase tend to improve the performance. For example, for a file of size 80 kB, 1 reducer case takes 132 sec, 2 reducers take 124 sec and 4 reducers take 108 sec.

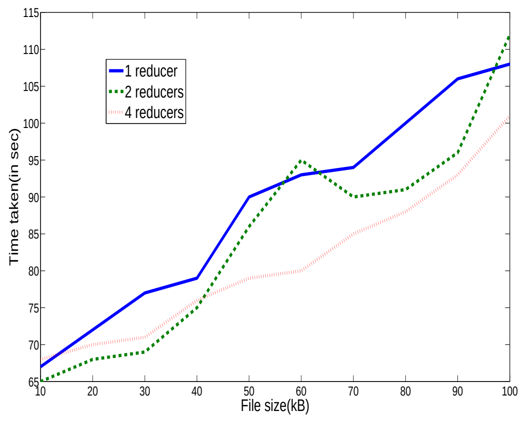


Fig. 5: PRE_MR performance for different number of reducers, $prefix\_length$=1

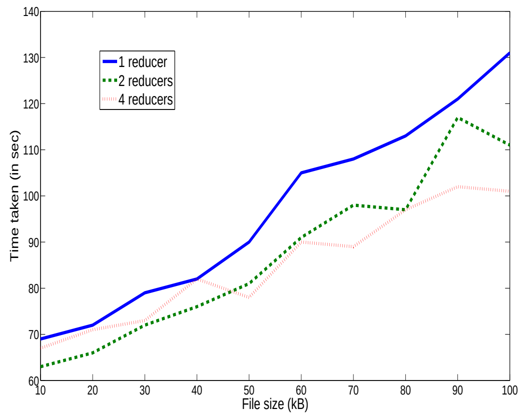


Fig. 6: PRE_MR performance for different number of reducers, $prefix\_length$=2

TABLE V: PRE_MR performance for different number of reducers, $prefix\_length$=3

File Size	1 reducer	2 reducers	4 reducers
10 kB	65 sec	67 sec	67 sec
20 kB	75 sec	73 sec	77 sec
30 kB	82 sec	77 sec	82 sec
40 kB	86 sec	84 sec	104 sec
50 kB	102 sec	94 sec	95 sec
60 kB	115 sec	93 sec	92 sec
70 kB	116 sec	99 sec	114 sec
80 kB	132 sec	124 sec	108 sec
90 kB	158 sec	127 sec	98 sec
100 kB	134 sec	115 sec	122 sec

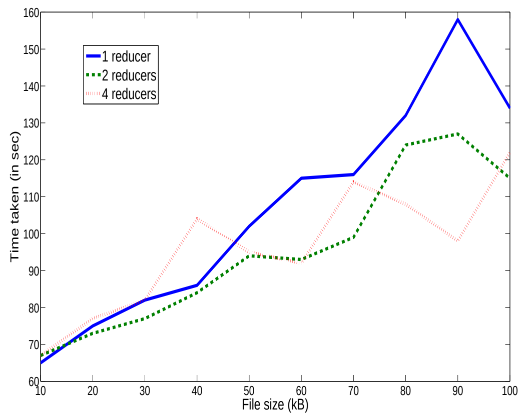


Fig. 7: PRE_MR performance for different number of reducers, $prefix\_length$=3

Table VI and Fig. 8 describe the times taken for different number of mappers for PRE_MR with $prefix\_length$ set to 1 and 4 reducers. In Fig. 8, the x-axis labels the size of the input file, and the runtime for the experiment are on the y-axis. As expected, with increase in the number of mapper nodes, the performance tends to improve. For example, for a file of size 80 kB, 2 mappers take 88 sec, 4 mappers take 85 sec and 8 mappers take 82 sec.

TABLE VI: PRE_MR performance for different number of mappers, $prefix\_length$=1, number of reducers=4

File Size	2 mappers	4 mappers	8 mappers
10 kB	68 sec	67 sec	66 sec
20 kB	70 sec	67 sec	67 sec
30 kB	71 sec	69 sec	67 sec
40 kB	76 sec	75 sec	76 sec
50 kB	79 sec	73 sec	74 sec
60 kB	80 sec	84 sec	78 sec
70 kB	85 sec	84 sec	82 sec
80 kB	88 sec	85 sec	82 sec
90 kB	93 sec	91 sec	90 sec
100 kB	101 sec	101 sec	90 sec

We observe in the results for PRE_MR performance that the running time does not always decrease when the number of mappers or reducers increases. We believe that this is because MapReduce resources are used to split the data and send it across to different nodes, and the intermediate results need to be shuffled across the network.

For the full text of Pride and Prejudice by Jane Austen, performing PRE_MR with 2 mappers and 4 reducers and after dividing the text into chunks of 100 kB took 684 seconds, when the '$prefix\_length$' parameter is set to 2. Using SIN_ED to do this after dividing the text into chunks of 10 kB took 967 seconds. However, we note that when reducing the file chunk size, the number of distinct string pairs, p reduce drastically, as p is proportional to the square of the number of distinct strings. So, we expect that the performance improvement using PRE_MR is much more than what this result indicates.

We verified the reproducibility of the experiments by carry-

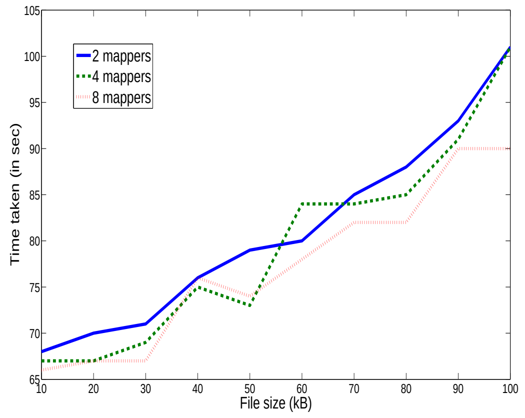


Fig. 8: PRE_MR performance for different number of mappers

ing out each of the experiments multiple times, and taking the average values. Besides, it was found that the results obtained had little standard deviation. In some additional experiments, for an increasing number of compute nodes, the improvement in performance was found to be quite substantial as the file size increased over 100 kB. For keeping the uniformity of the results across all experiments, we haven't presented the results for file sizes more than 100 kB or for larger number of map and reduce nodes, since we hadn't evaluated these cases on all experiments. The presented results are aimed to show trends in performance change with varying file sizes and different number of computation nodes.

VI. Conclusions and Future Work

Although there are several efficient algorithms for calculating edit distance and related problems, computing edit distance for a large set of strings is expensive. We propose an efficient parallel implementation for this, using MapReduce. With support from our experimental results of Section IV, we argue that our approach is much more efficient than the usual dynamic programming method. We can also tune the '$prefix\_length'$ parameter in PRE_MR, and the number of nodes used in the map phase and reduce phase to improve the performance of our algorithms for varying input file sizes.

As the number of mapper and reducer nodes are increased in MapReduce, there is greater parallelization and the number of processes increase. In Table I, PRE_MR is three times faster than the sequential procedure because it uses 4 reducers instead of 1. The speedup is not substantial when doubling the mappers and reducers because as mentioned previously, MapReduce resources are used to split the data and send it to these nodes, and the intermediate results are shuffled across the network. However, we expect this to get more than compensated for with larger files, where each prefix pair would be expected to have a larger number of corresponding string pairs, and thus each reduce process initiated would produce more results.

The optimal number of mappers, reducers and the '$prefix\_length'$ parameter value vary with the file size and file content. It is hoped that the results on varied experiments presented can help guide towards a good initial guess for these parameters.

The field of dynamic programming problems is far from exhausted when it concerns creating scalable, effective, parallel algorithms. We argue, however, that our algorithms are a step in the right direction. Future research includes further testing to explore their efficiency in different datasets. In addition, further analysis of dynamic programming algorithms can lead to more effective MapReduce solutions, especially for problems that require ad-hoc data analysis.

Acknowledgment

This material is based upon work supported by National Science Foundation under Award No. CNS 1229652, and the Air Force Office of Scientific Research under Award No. FA-9550-09-1-0468 and Award No. FA-9550-12-1-0077. We thank Dr. Robert Herklotz for his support.

References

[1] http://nlp.stanford.edu/IR-book/html/htmledition/edit-distance-1.html

[2] R. A. Wagner and M. J. Fischer, *The string-to-string correction problem*, J. Assoc. Comput. Machinery, vol. 21, no. 1, pp. 168-173, Jan. 1974.

[3] D. Sankoff and J. B. Kruskal, Time Warps, String Edits, and Macromolecules: *The Theory and Practice of Sequence Comparison*. Reading, MA: Addison-Wesley, 1983.

[4] W. J. Masek and M. S. Patterson, *A faster algorithm computing string edit distances,* J. Comput. Syst. Sci., vol. 20, pp. 18-31, Feb. 1980.

[5] P.A.V. Hall and G.R. Dowling, *Approximate string matching*, ACM Comput. Surveys, vol. 12, pp. 381-402, Dec. 1980.

[6] P. H. Sellers, *The theory and computation of evolutionary distances: Pattern recognition,* J. Algorithms, vol. 1, pp. 359-373, 1980.

[7] Y. P. Yang and T. Pavlidis, *Optimal correspondence of string subsequences,* IEEE Trans. Patt. Anal. Machine Intell., vol. 12, no. 11, pp.1080-1087. Nov. 1990.

[8] K. S . Fu, *Syntactic Pattern Recognition and Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1982.

[9] Marzal, A., and E. Vidal. *Computation of Normalized Edit Distance and Applications*. IEEE Transactions on Pattern Analysis and Machine Intelligence 15.9 (1993): 926-32. Print.

[10] Massimo Gaggero, Simone Leo, Simone Manca, Federico Santoni, Omar Schiaratura, Gianluigi Zanetti: *Parallelizing bioinformatics applications with MapReduce*. Cloud Computing and Its Applications 2008.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein: *Introduction to Algorithms*. Cambridge, MA: MIT, 1990. Print.

[12] David Sankoff and Joseph B. Kruskal. *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison.* Addison-Wesley Publishing Company, Inc., 1983.

[13] http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/

[14] http://mootools.net/forge/p/string_levenshtein

[15] Ristad, E.s., and P.n. Yianilos. *Learning String-edit Distance.* IEEE Transactions on Pattern Analysis and Machine Intelligence 20.5 (1998): 522-32. Print.

[16] Bar-Yossef, Z.; Jayram, T. S.; Krauthgamer, R.; Kumar, R., *Approximating edit distance efficiently,* Foundations of Computer Science, 2004. Proceedings. 45th Annual IEEE Symposium on , vol., no., pp.550,559, 17-19 Oct. 2004

[17] Jie Wei, *Markov edit distance*, Pattern Analysis and Machine Intelligence, IEEE Transactions on , vol.26, no.3, pp.311,321, March 2004

[18] Arslan, A.N.; Egecioglu, O., *An efficient uniform-cost normalized edit distance algorithm,* String Processing and Information Retrieval Symposium, 1999 and International Workshop on Groupware , vol., no., pp.8,15, 1999

[19] Fuad, M.M.M.; Marteau, P.-F., *The extended edit distance metric,* Content-Based Multimedia Indexing, 2008. CBMI 2008. International Workshop on , vol., no., pp.242,248, 18-20 June 2008

[20] de la Higuera, C.; Mico, L., *A contextual normalised edit distance,* Data Engineering Workshop, 2008. ICDEW 2008. IEEE 24th International Conference on , vol., no., pp.354,361, 7-12 April 2008

[21] H. Bunke. *On a Relation between Graph Edit Distance and Maximum Common Subgraph*. Pattern Recognition Letters, vol. 18, no. 8, pp. 689-694, 1997.

[22] Hiroyuki Hanada, Atsuyoshi Nakamura and Mineichi Kudo. *A practical comparison of edit distance approximation algorithms*. Granular Computing (GrC), page 231-236. IEEE, (2011)

[23] Shivani Jain and A.L.N. Rao. *A Comparative Performance Analysis of Approximate String Matching*. International Journal of Innovative Technology and Exploring Engineering (IJITEE) ISSN: 2278-3075, Volume-3, Issue-5, October 2013

[24] Robles-Kelly, A., and E.r. Hancock. *Graph Edit Distance from Spectral Seriation.* IEEE Transactions on Pattern Analysis and Machine Intelligence 27.3 (2005): 365-78. Print.

[25] Husain, M. F., P. Doshi, L. Khan, and B. Thuraisingham (2009). *Storage and retrieval of large rdf graph using hadoop and mapreduce*. *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, Berlin, Heidelberg, pp. 680–686. Springer-Verlag.

[26] Husain, M. F., L. Khan, M. Kantarcioglu, and B. Thuraisingham (2010). *Data intensive query processing for large rdf graphs using cloud computing tools*. *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, CLOUD '10, Washington, DC, USA, pp. 1–10. IEEE Computer Society.

[27] Husain, M. F., J. P. McGlothlin, M. M. Masud, L. R. Khan, and B. M. Thuraisingham (2011). *Heuristics-based query processing for large rdf graphs using cloud computing*. *IEEE Trans. Knowl. Data Eng. 23*(9), 1312–1327.

[28] Parveen, P., B. Thuraisingham, and L. Khan (2013b, October). *Map reduce guided scalable compressed dictionary construction for repetitive sequences.* *IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*

[29] Parveen, P., N. McDaniel, J. Evans, B. Thuraisingham, K. W. Hamlen, and L. Khan (2013, October). *Evolving insider threat detection stream mining perspective.* *International Journal on Artificial Intelligence Tools (World Scientific Publishing) 22*(5), 1360013–1–1360013–24.

[30] Cairong Yan, Xin Yang, Ze Yu, Min Li and Xiaolin Li. *IncMR: Incremental Data Processing based on MapReduce.* IEEE CLOUD, page 534-541. IEEE, (2012)

[31] Lin, Jimmy, and Chris Dyer. *Data-intensive Text Processing with MapReduce.* San Rafael: Morgan and Claypool, 2010. Print.

[32] J. Ekanakake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, and G. Fox. *Twister: a runtime for iterative mapreduce,* in 19th ACM International Symposium on High Performance Distributed Computing (HPDC), 2010

[33] R. Pereira, M. Azambuja, K. Breitman, and M. Endler, *An architecture for distributed high performance video processing in the cloud,* in 3rd International Conference on Cloud computing (Cloud), 2010

[34] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and C. G., *Pregel: a system for large-scale graph processing,* in International Conference on Management of data (SIGMOD), 2010

[35] Q. He, C. Du, Q. Wang, F. Zhuang, and Z. Shi, *A parallel incremental extreme svm classifier*, Neurocomputing, vol. 74, no. 16, pp. 25322540, 2011.

[36] Pallabi Parveen, Pratik Desai, Bhavani M. Thuraisingham, Latifur Khan: *MapReduce-guided scalable compressed dictionary construction for evolving repetitive sequence streams*. CollaborateCom 2013: 345-352