MapReduce-Guided Scalable Compressed Dictionary Construction for Evolving Repetitive Sequence Streams

(Invited Paper)

Pallabi Parveen, Pratik Desai, Bhavani Thuraisingham and Latifur Khan
Department of Computer Science
The University of Texas at Dallas
Richardson, Texas
Email: pxp013300, pxd123230, bhavani.thuraisingham, lkhan@utdallas.edu

***Abstract*—Users' repetitive daily or weekly activities may constitute user profiles. For example, a user's frequent command sequences may represent normative pattern of that user. To find normative patterns over dynamic data streams of unbounded length is challenging. For this, an unsupervised learning approach is proposed in our prior work by exploiting a compressed/quantized dictionary to model common behavior sequences. This work suffers scalability issues. Hence, in this paper, we propose and implement a MapReduce-based framework to construct a quantized dictionary. We show effectiveness of our distributed parallel solution on a benchmark dataset.**

***Index Terms*—MapReduce, Cloud, Sequence, Unsupervised Learning**

I. Introduction

Normal user profiles are considered to be repetitive daily or weekly activities which are frequent sequences of commands, system calls, etc. These repetitive command sequences are called normative patterns. These patterns reveal the regular or normal behavior of a user [1], [2]. Command sequences continuously arrive as a stream and evolve over time. Due to the stream nature, when we extract normative patterns, we will develop one pass algorithms to find normative patterns [3]–[6]. In other words, the algorithms cannot go over the same command sequences more than once. In addition, the normative patterns may change over time; new normative pattern may evolve. Hence, the algorithm needs to be adaptive or incremental in nature. For example, a novice programmer can develop his skills to become an expert programmer over time.

Finding normative patterns is an important problem. This is because normative patterns of a user can be used as a user profile. A user profile can be used for targeted advertisement or insider threat detection. In particular, when a user suddenly demonstrates unusual activities that indicate a significant excursion from normal behavior, an alarm is raised for potential insider threat [2], [7]–[9]. So, in order to identify insider threats, first we need to find normal user behavior. For that, we need to collect sequences of commands and find the potential normative patterns observed within these command sequences in an unsupervised fashion.

In our prior work, an unsupervised learning approach is used to find normative patterns [1], [10], [11]. During the learning process, we store the repetitive sequence patterns from a users actions or commands in a model called a Quantized Dictionary. In particular, longer patterns with higher weights due to frequent appearances in the stream are considered in the dictionary. To cope with changes, our approach can exploit incremental learning where the dictionary will be continuously updated with new incoming sequences.

Construction of a quantized dictionary is time consuming. We would like to address scalability issues of this algorithm. One possible solution is to adopt parallel/distributed computing. Here, we would like to exploit cloud computing based on commodity hardware [12]–[15]. Cloud computing is a distributed parallel solution. For our approach, we utilize a Hadoop and MapReduce based framework to facilitate parallel computing.

Our primary contributions are as follows. First, we propose a framework for an unsupervised learning to find pattern sequences from successive user actions or commands using unsupervised quantized dictionary construction. Second, we propose a scalable solution to construct quantize dictionary using the Hadoop and MapReduc framework. Finally, we compare our approach with other alternatives and show the effectiveness of our approach in terms of speed on a benchmark dataset.

The rest of the paper is organized as follows. Section II presents our proposed unsupervised sequence learning. Section III presents complexity of our approach. Section IV addresses scalability issues of our approach using MapReduce framework. Section V presents experimental setup and results of our approach. Finally, Section VI concludes and suggests future work.

II. Unsupervised Sequence Learning (USSL)

This unsupervised approach needs to identify normal user behavior in a single pass [10], [11], [16]. One major challenge with these repetitive sequences is their variability in length. To combat this problem, we need to generate a dictionary which will contain any combination of possible normative patterns existing in the gathered data stream.

COLLABORATECOM 2013, October 20-23, Austin, United States

DOI 10.4108/icst.collaboratecom.2013.254135

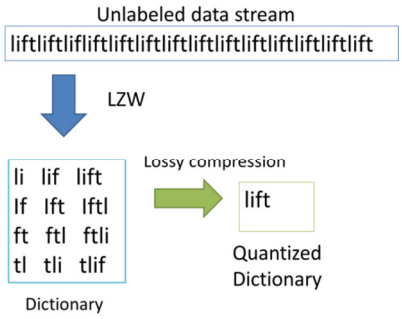

Figure 1. Quantization of dictionary

Potential variations that could emerge within the data include the commencement of new events, the omission or modification of existing events, or the reordering of events in the sequence. *Eg., liftliftliftliftliftcomcomecomecomecomecome,* is a sequence of commands represented by the alphabets given in a data stream. We will consider all patterns *li,if,ft,tl, lif, ift, ftl, lift, iftl etc.*, as our possible normative patterns. However, the huge size of the dictionary presents another significant challenge.

We have addressed the above two challenges in the following ways. First, we extract possible patterns from the current data chunk using single pass algorithm (e.g., LZW, Lempel-Ziv- Welch algorithm [17]) to prepare a dictionary. We called it LZW dictionary. LZW dictionary has a set of patterns and their corresponding weights according to

$$w_i = \frac{f_i}{\sum_{i=1}^{n} f_i} \quad (1)$$

where w_i is the weight of a particular pattern p_i in the current chunk, f_i is the number of times the pattern p_i appears in the current chunk, and n is the total number of distinct patterns found in that chunk.

Next, we compress the dictionary by keeping only the longest and frequent unique patterns according to their associated weight and length, while discarding other subsumed patterns. This technique is called compression method (CM), and the new dictionary is a Quantized dictionary (QD). The Quantized dictionary has a set of patterns and their corresponding weights. Here, we use edit distance to find the longest pattern. Edit distance is a measure of similarity between pairs of strings [18]. It is the minimum number of actions required to transfer one string to another where an action can be substitution, addition, or deletion of a character into the string. As in case of the earlier example mentioned, the best normative pattern in the quantized dictionary would be lift, come, etc.

This process is a lossy compression, but is sufficient enough to extract the meaningful normative patterns. The reason behind this is the patterns that we extract are the superset of the subsumed patterns. Moreover, as frequency is another control parameter in our experiment, the patterns which do not appear often cannot be regular user patterns.

A. Construct the LZW dictionary by selecting the patterns in the data stream

At the beginning, we consider that our data is not annotated (i.e., unsupervised). In other words, we dont know the possible sequence of future operations by the user. So, we use LZW algorithm [17] to extract the possible sequences that we can add to our dictionary. These can also be commands like *liftliftliftliftliftcomcomecomecomecomecome,* where each unique letter represents a unique system call or command. We have used Unicode to index each command. E.g, ls, cp, find are indexed as *l*, *c*, and *f*. The possible patterns or sequences are added to our dictionary would be *li, if,ft,tl, lif,ift, ftl, lift, iftl, ftli, tc, co, om, mc,com, come* and so on. When the sequence *li* is seen in the data stream for the second time, in order to avoid repetition it will not be included in the LZW dictionary. Instead, we increase the frequency by 1 and extend the pattern by concatenating it with the next character in the data stream, thus turning up a new pattern *lif*. We will continue the process until we reach the end of the current chunk. Fig. 1 demonstrates how we generate an LZW dictionary from the data stream.

linesnumbered 1 Quantized Dictionary

Input: $D = \{P, W\}$ (LZW dictionary)
Output: QD (quantized dictionary)
2 **while** $D \neq 0$ **do**
4 $X \leftarrow D_1$ `// first pattern`
6 **foreach** $i \in D$ **do** `// for each pattern`
8 **if** $edit_{distance(X,D_i)} = 1$ **then** $P \leftarrow P \cup i$
10 $D \leftarrow D$ 11 X **if** $P \neq 0$ **then**
13 $X \leftarrow choose(\arg\max_i(P_i length(i)))$ $QD \leftarrow QD \cup$
15 X $D \leftarrow D - X$
17 $X \leftarrow D_1$ `// next pattern`

B. Constructing the Quantized Dictionary

Once we have our LZW dictionary, we keep the longest and most frequent patterns and discard all their subsumed patterns. Algorithm 1 shows step by step how a quantized dictionary is generated from LZW dictionary. Inputs of this algorithm are as follows: LZW dictionary D which contains a set of patterns P and their associated weight W. Line 4 picks a pattern (e.g., li). Lines 6 to 8 find all the closest patterns that are 1 edit distance away. Lines 11 to 15 keep the pattern which has the highest weight multiplied by its length and discard the other patterns. We repeat the steps (line 4 to 15) until we find the longest, frequent pattern (*lift*). After that, we start with a totally different pattern (*co*) and repeat the steps until we have explored all the patterns in the dictionary. Finally, we end up with a more compact dictionary which will contain many meaningful and useful sequences. We call this dictionary our quantized dictionary. Fig. 1 demonstrates how we generate a quantized dictionary from the LZW dictionary.

Once we identify different patterns *lift*, *come*, etc., any pattern with $X\%(\geq \%30$ in our implementation) deviation

from all these patterns would be considered as anomaly. Here, we will use edit distance to identify the deviation.

III. Complexity Analysis

Here, we will report time complexity of quantized dictionary construction. In order to calculate edit distance between two patterns of length K (in worst case maximum length would be K), our worst case time complexity would be $O(K^2)$.

Suppose we have n patterns in our LZW dictionary. We have to construct quantized dictionary from this LZW dictionary. In order to do this, we need to find patterns in LZW dictionary which have 1 edit distance from a particular pattern (say p). We have to calculate edit distance between all the patterns and the pattern p. Recall that time complexity to find edit distance between two patterns is $O(K^2)$. Since there are total n number of distinct patterns, total time complexity between p and the rest of patterns is $O(n \times K^2)$. Note that p is one of the member of n patters. Therefore, total time complexity between pair of patterns is $O(n^2 \times K^2)$. This is valid for a single user. If there is u of distinct users, total time complexity across u user is $O(u \times n^2 \times K^2)$ (see Table I)

IV. Scalability using Hadoop and MapReduce

This section presents proposed scalable LZW and quantized dictionary construction algorithm using Map Reduce (MR) [12]–[15]. We address scalability issues using the following two approaches. Approaches are illustrated in Fig. 2. Our proposed approach exploits in one case two map reduce jobs (2MRJ) and the other case it exploits a single map reduce job (1MRJ).

Here, we consider a number of users and their command sequence stream as input. Hence, a set of user's command sequence patterns will be the input for mapper program. In this work, we have not considered one individual particular user command sequence as input to the mapper program. To generate a large dataset, we consider all users together.

1) Two Map Reduce Jobs Approach (2MRJ): This is a simple approach and requires two map reduce (MR) jobs. It is illustrated Fig. 3 and Fig. 4. The first MR job is dedicated for LZW dictionary construction in (Fig. 3) and the second MR job is dedicated for quantized dictionary construction in (Fig. 4). In the first MR job, Mapper takes userid along with command sequence as an input to generate intermediate (key, values) pair having the form ((userid, css), 1). Note that css is a pattern which is a command sub-sequence. In Reduce phase intermediate key (userid, css) will be the input. Here, keys are grouped together and values for the same key (word count) are added. For example, a particular user 1, has command sequences "liftlift". Map phase emits ((u1, li), 1) ((u1, lif), 1) value as intermediate key value pairs (see middle portion of Fig. 3). Recall that the same intermediate key will go to a particular reducer. Hence, a particular user id along with pattern/css, i.e., key will arrive to the same reducer. Here, reducer will emit (user id, css) as key and value will be how many times (aggregated one) pattern appears in the command sequence for that user (see bottom portion of Fig. 3).

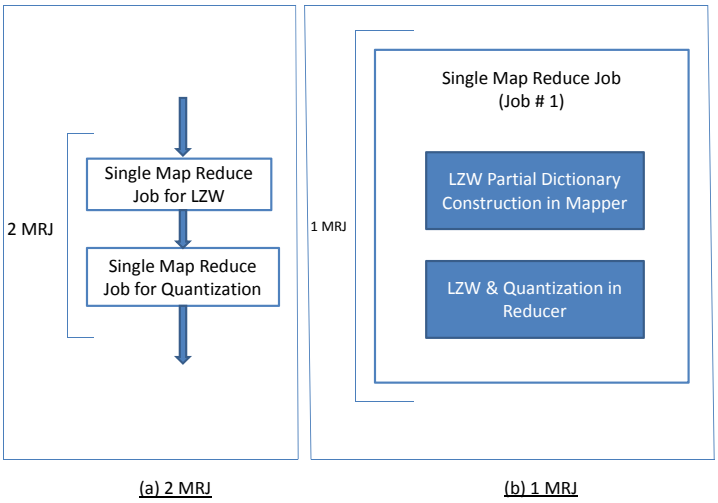

Figure 2. Approaches for Scalable LZW and Quantized Dictionary Construction using Map Reduce Job

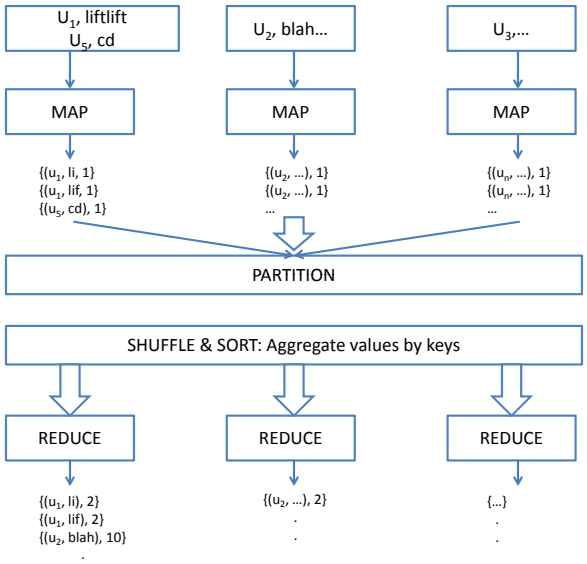

Figure 3. First Map Reduce Job for Scalable LZW Construction in 2MRJ approach

TABLE I
TIME COMPLEXITY OF QUANTIZATION DICTIONARY CONSTRUCTION

Description	Time Complexity
Pair of Patterns	$O(n^2 \times K^2)$
u number of user	$O(u \times n^2 \times K^2)$

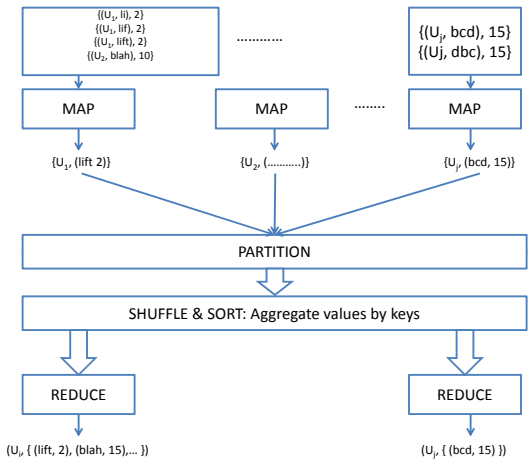

Figure 4. Second Map Reduce Job for Quantized Dictionary Construction in 2MRJ approach

2) 1MRJ:1 MR Job: Here, we will utilize 1 MR job. It is illustrated in Fig. 5. We are expecting that by reducing the number of jobs we can reduce total processing costs.

Running a job in Hadoop takes significant overhead. Hence, by minimizing the number of jobs, we can construct the dictionary quickly. The overhead for a Hadoop job is associated with disk I/O and network transfers. When a job is submitted to Hadoop cluster the following actions will take place:

1) The Executable file is moved from client machine to Hadoop JobTracker[1],
2) The JobTracker determines TaskTrackers[2] that will execute the job,
3) The Executable file is distributed to the TaskTrackers over the network,
4) Map processes initiates reading data from HDFS,
5) Map outputs are written to local discs,
6) Map outputs are read from discs, shuffled (transferred over the network to TaskTrackers which would run Reduce processes), sorted and written to remote discs,
7) Reduce processes initiate reading the input from local discs,
8) Reduce outputs are written to discs.

Therefore, if we can reduce the number of jobs, we can avoid expensive disk operations and network transfers. That is the reason we prefer 1MRJ over 2MRJ.

[1]http://wiki.apache.org/hadoop/JobTracker
[2]http://wiki.apache.org/hadoop/TaskTracker

linesnumbered 2 Dictionary Construction and Compression using single Map-Reduce(1MRJ)

Input: $gname : groupname, cseq : commandsequences$
Output: $Key : gname, commandpattern(css)$

$map(stringgname, stringcseq)$
$start \leftarrow 1$, $end \leftarrow 2$
$css = (cs_{start} \cdots cs_{end})$
if $css \notin dictionary$ **then**
  $Add\ css\ to\ thedictionary$
  $emit(gname, css)$
  $start \leftarrow end$
  $end \leftarrow end + 1$
else
  $emit(gname, css)$
  $end \leftarrow end + 1$
end if

$reduce(gname, (css_1, css_2, \cdots)$
$H \leftarrow 0$
for all $css_i \in ((css_1, css_2, \cdots)$ **do**
  if $css \notin H$ **then**
    $H \leftarrow H + (css_i, 1)$
  else
    $count \leftarrow getfrequency(H(css_i))$
    $count \leftarrow count + 1$
    $H \leftarrow H + (css_i, count)$
  end if
end for
$QD = QuantizedDictionary(H)$
for all $css_i \in QD$ **do**
  $emit(gname, pair(css_i, count(css_i)))$
end for

Mapper will emit user id as key and value will be pattern (see Algorithm 2). The same user id will arrive at the same reducer since user is the intermediate key. For that user id, a reducer will have a list of patterns. In the reducer, Edit distance operation will be implemented as described in Section II. Parallelization will be achieved at the user level (inter user parallelization) instead of within users (intra user parallelization). In mapper, parallelization will be carried out by dividing large files into a number of chunks and process a certain number of files in parallel.

Algorithm 2 illustrates the idea. Input file consists of line by line input. Each line has entries namely, gname (userid) and command sequences (cseq). Next, mapper will take gname

(userid) as key, and values will be command sequences for that user. In mapper we will look for patterns having length 2, 3, etc. Here, we will check whether patterns exist in the dictionary (line 6). If the pattern does not exist in the dictionary, we simply add that in the dictionary(line 7), and emit intermediate key value pairs (line 8) having keys as gname and values as patterns with length 2, 3 etc. At line 9 and 10, we increment pointer so that we can look for patterns in new command sequences (cseq). If the pattern is in the dictionary, we simply emit at line 12 and cseq's end pointer is incremented so that we can look for super-set command sequence.

At the reducer, each user (gname) will be input and list of values will be patterns. Here, compression of patterns will be carried for that user. Recall that some patterns will be pruned using Edit distance. For a user each pattern will be stored into Hashmap, H. Each new entry in the H will be pattern as key and value as frequency count. For existing pattern in the dictionary, we will simply update frequency count(line 18). At line 20 dictionary will be quantized, and H will be updated accordingly. Now, from quantized dictionary all distinct patterns from H will emitted as values along with key gname.

V. Experimental Setup and Results

A. Hadoop Cluster

Our hadoop cluster (cshadoop0-cshadoop9) is compromised of virtual machines that run in the Computer Science vmware esx cloud - so there are 10 VM's. Each VM is configured as a quad core with 4GB of ram and a 256GB virtual hard drive. The virtual hard drives are stored on the CS SAN (3PAR).

There are three ESX hosts which are Dell Poweredge R720's with 12 cores @2.99GHZ, 128GB of RAM, and fiber to the 3PAR SAN. The VM's are spread across the three ESX hosts in order to balance the load.

cshadoop0 is configured as the "name node". A"cshadoop1" through "cshadoop9" are configured as the slave "data nodes".

We have implemented our approach using Java JDK version 1.6.0.39. For MapReduce implementation we have used Hadoop version 1.0.4.

B. Big Dataset for Insider Threat Detection

The Data sets used are created from the trace files from the University of Calgary project. 168 files have been collected from different levels of users of UNIX as described in [9], [19]. The different levels of users are:

- Novice programmers (56 users)
- Experienced programmers (36 users)
- Computer scientists (52 Users)
- Non-programmers (25 users).

Each file contains the commands used by each of the users over weeks.

Now, in order to get the big data, first we replicated the user files randomly so that we have:

- Novice programmers - (1320 Users) i.e. File starting from "novice-0001" till "novice-1320"

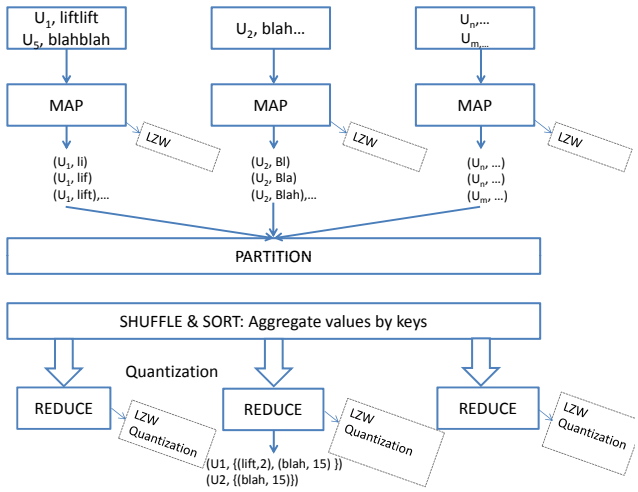

Figure 5. 1MRJ: 1 MR Job Approach for Scalable LZW and Quantized Dictionary Construction

- Experienced programmers - (576 Users)
- Computer scientists - (832 Users)
- Non-programmers - (1600 Users)

Total Number of Users = 4328; size is 430 MB; and one command file is for one user.

Next, we gave these files as input to our program (written in Python) which gave unique unicode for each distinct command provided by all users. The output file for all users is 15.5 MB. We dubbed it as *original data (OD)*.

Finally, we replicated this data 12 times for each user. And we ended up 187 MB of input file which was given as an input to Map Reduce job of LZW and Compression. We dubbed as *duplicate big data (DBD)*.

C. Results for Big Data Set Related to Insider Threat Detection

First, we experiment on OD, and next, we concentrate on DBD.

On OD Dataset:

We have compared our approaches namely, 2MRJ and 1MRJ on OD dataset. Here, we have varied number of reducers and fixed number of mappers (e.g., HDFS block size equals 64MB). In case 2MRJ, we have varied a number of reducers in 2nd job's reducer and not in first map reduce job. 1MRJ outperforms 2MRJ in terms of processing time on a fixed number of reducers except in the first case (number reducer equals to 1). With the latter case, parallelization is limited at the reducer phase. Table II illustrates this. For example, for number of reducer equals 9, total time taken is 3.47 sec and 2.54 sec for 2MRJ and 1MRJ approaches respectively.

With regard to 2MRJ case, Table IV presents input-output statistics of both MapReduce jobs. For example, for first map reduce job mapper emits 65,37,040 intermediate key value

TABLE II
TIME PERFORMANCE OF 2MRJ VS 1MRJ FOR VARYING NUMBER OF REDUCERS

# of Reducer	Time for 2MRJ (M:S)	Time for 1MRJ (M:S)
1	13.5	16.5
2	9.25	9.00
3	6.3	5.37
4	5.45	5.25
5	5.21	4.47
6	4.5	4.20
7	4.09	3.37
8	3.52	3.04
9	3.47	2.54
10	3.38	2.47
11	3.24	2.48
12	3.15	2.46

TABLE III
TIME PERFORMANCE OF MAPPER FOR LZW DICTIONARY CONSTRUCTION WITH VARYING PARTITION SIZE IN 2MRJ

Partition block size	Map (Sec)	No of mappers
1MB	31.3	15
2MB	35.09	8
3MB	38.09	5
4MB	36.06	4
5MB	41.01	3
6MB	41.03	3
7MB	41.01	3
8MB	55.0	2
64MB	53.5	1

pairs and Reducer emits 95.75MB output. This 95.75MB will be the input for mapper for the second MapReduce job.

Here, we will show how HDFS block size will have an impact on LZW dictionary construction in 2MRJ case. First, we vary HDFS block size that will control the number of mappers. With 64 MB HDFS block size and 15.5MB input size, number of mapper equals to 1. For 4MB HDFS block size number of mapper equals 4. Here, we assume that input File split size equals HDFS block size. Smaller HDFS block size (smaller file split size) increases performance (reduce time). More mappers will be run in various nodes in parallel.

Table III presents total time taken by mapper (part of first MapReduce job) in 2MRJ case on OD dataset. Here, we have varied partition size for LZW Dictionary Construction. For 15.498MB input file size with 8MB partition block size, MapReduce execution framework used 2 mappers.

on DBD Dataset: Table V shows the details of the value comparisons of 1MRJ across a various number of reducer and HDFS block size values. Here, we have used DBD dataset.

In particular, in Fig. 6, we show total time taken for a varying number of reducers with a fixed HDFS block size. Here, X axis represents the number of reducer and Y axis represents the total time taken for 1MRJ with a fixed HDFS block size. We demonstrate that with an increasing number of reducers, total time taken will drop gradually. For example, with regard to reducer 1, 5, and 8, total time taken in 1MRJ approach is 39.24, 13.04, 10.08 minutes respectively. This number validates our claim. With more reducers running

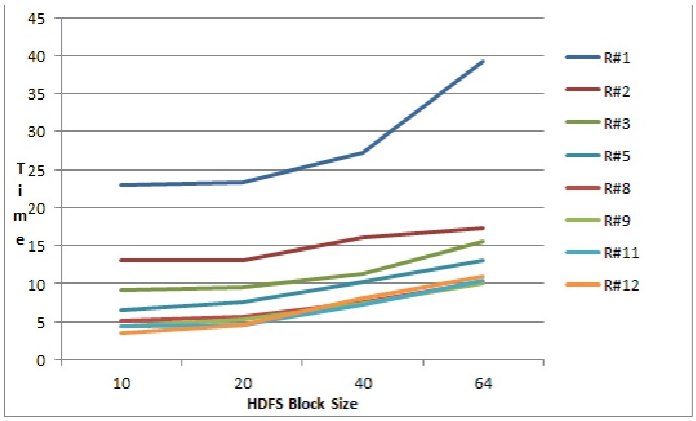

Figure 6. Time Taken for Varying Number of HDFS block size in 1MRJ

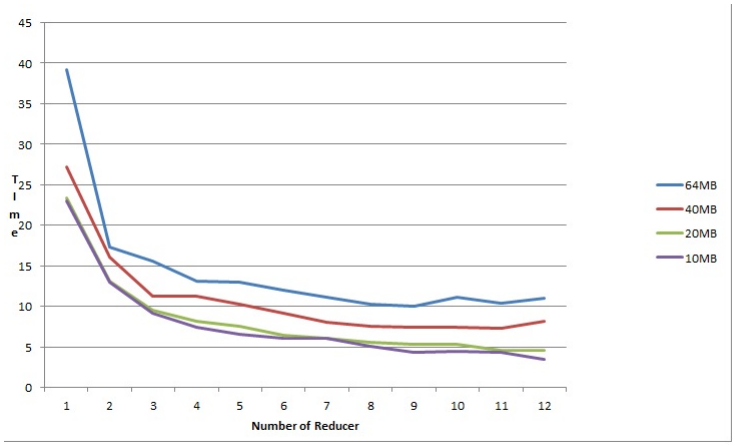

Figure 7. Time Taken for Varying Number of Reducer in 1MRJ

in parallel, we can run quantization/compression algorithms for various users in parallel. Recall that in 1MRJ reducer will get each distinct user as key and values will be LZW dictionary pattern. Let us assume that we have 10 distinct users and their corresponding patterns. For compression with 1 reducer, compression for 10 user patterns will be carried out in a single reducer. On the other hand for 5 reducers, it is expected that each reducer will get 2 users' patterns. Consequently, 5 reducers will run in parallel and each reducer will execute compression algorithm for 2 users serially instead of 10. Therefore, with an increasing number of reducers, performance(decreases time) improves.

Now, we will show how the number of mappers will affect total time taken in 1MRJ case. The number of mappers is usually controlled by the number of HDFS blocks (dfs.block.size) in the input files. Number of HDFS blocks in the input file is determined by HDFS block size. Therefore, people adjust their HDFS block size to adjust the number of maps.

Setting the number of map tasks is not as simple as setting up the number of reduce tasks. Here, first we determine whether input file is isSplitable. Next, three

TABLE IV
DETAILS OF LZW DICTIONARY CONSTRUCTION AND QUANTIZATION USING MAP REDUCE IN 2MRJ ON OD DATASET

Description	Size/Entries in Second Job	Size/Entries in First Job
Map Input	95.75MB (size)	15.498MB (size)
Map Output	45,75,120 (entries)	65,37,040 (entries)
Reduce Input	17,53,590 (entries)	45,75,120 (entries)
Reduce Output	37.48 MB	95.75 MB (size)

TABLE V
TIME PERFORMANCE OF 1MRJ FOR VARYING REDUCER AND HDFS BLOCK SIZE ON DBD

No of Reducer	64MB	40MB	20MB	10MB
1	39:24	27:20	23:40	24:58
2	17:36	16:11	13:09	14:53
3	15:54	11:25	9:54	9:12
4	13:12	11:27	8:17	7:41
5	13:06	10:29	7:53	6:53
6	12:05	9:15	6:47	6:05
7	11:18	8:00	6:05	6:04
8	10:29	7:58	5:58	5:04
9	10:08	7:41	5:29	4:38
10	11:15	7:43	5:30	4:42
11	10:40	7:30	4:58	4:41
12	11:04	8:21	4:55	3:46

variables, mapred.min.split.size, mapred.max.split.size, and dfs.block.size, determine the actual split size. By default, min split size is 0 and max split size is Long.MAX and block size 64MB. For actual split size, minSplitSize & blockSize set the lower bound and blockSize & maxSplitSize together sets the upper bound. Here is the function to calculate:

max(minsplitsize, min(maxsplitsize, blocksize))

For our case we use min split size is 0; max split size is Long.MAX and blockSize vary from 10MB to 64MB. Hence, actual split size will be controlled by HDFS block size. For example, 190MB input file with DFS block size 64MB, the file will be split into 3 with each split having two 64 MB and the rest with 62 MB. Finally, we will end up with 3 maps.

In Fig. 7 we show the impact of HDFS block size on total time taken for a fixed number of reducers. Here, X axis represents HDFS block size and Y axis represents total time taken for 1MR approach with a fixed number of reducers. We demonstrate that with increasing number of HDFS block size, total time taken will increase gradually for a fixed input file. For example, with regard to HDFS block size 10, 20, 40, 64 MB total time taken in 1MRJ approach were 7.41, 8.17, 11.27, 13.12 minute respectively for a fixed number of reducers (=4). On one hand, when HDFS block size of 10 MB, and input file is 190 MB, 19 maps run where each map processes 10MB input split. On the other hand, for HDFS block size=64MB, 3 maps will be run where each map will process a 64MB input split. In the former case (19 maps with 10 MB) each map will process a smaller file and in the latter case (3 maps with 64MB) we process a larger file which consequently consumes more time. In the former case, more parallelization can be achieved. In our architecture, more than 10 mappers can be run in parallel. Hence, for a fixed input file and fixed number of reducers, total time increases with increasing HDFS block size.

VI. CONCLUSIONS AND FUTURE WORK

Compressed/quantized dictionary construction is computationally expensive. It does not scale well with a number of users. Hence, we look for distributed solution with parallel computing with commodity hardware. For this, all users quantized dictionary is constructed using a MapReduce framework on Hadoop. A number of approaches are suggested, experimented on benchmark dataset, and discussed. We have shown with 1 map reduce job that quantized dictionary can be constructed and demonstrates effectiveness over other approaches.

We could extend the work in the following directions. First, we will build a full fledge system to capture user input as stream using apache flume and store it on the Hadoop distributed file system (HDFS) and then apply our approach as discussed previously. Second, we will apply MapReduce to calculate edit distance between patterns. Finally, we will update the quantized dictionary using ensemble based techniques instead of the incremental approach.

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 & FA9550-08-1-0260. We thank Dr. Robert Herklotz for his support.

REFERENCES

[1] P. Parveen, N. McDaniel, J. Evans, B. Thuraisingham, K. W. Hamlen, and L. Khan, "Evolving insider threat detection stream mining perspective," *Journal International Journal on Artificial Intelligence Tools World Scientific Publishing (to appear in).*

[2] K. Wang and S. J. Stolfo, "One-class training for masquerade detection," in *Proc. ICDM Workshop on Data Mining for Computer Security (DMSEC)*, 2003.

[3] M. M. Masud, Q. Chen, L. Khan, C. C. Aggarwal, J. Gao, J. Han, A. N. Srivastava, and N. C. Oza, "Classification and adaptive novel class detection of feature-evolving data streams," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 7, pp. 1484–1497, 2013.

[4] T. Al-Khateeb, M. M. Masud, L. Khan, and B. M. Thuraisingham, "Cloud guided stream classification using class-based ensemble," in *IEEE CLOUD*, 2012, pp. 694–701.

[5] M. M. Masud, C. Woolam, J. Gao, L. Khan, J. Han, K. W. Hamlen, and N. C. Oza, "Facing the reality of data stream classification: coping with scarcity of labeled data," *Knowl. Inf. Syst.*, vol. 33, no. 1, pp. 213–244, 2011.

[6] P. Domingos and G. Hulten, "Mining high-speed data streams," in *Proc. ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, 2000, pp. 71–80.

[7] R. C. Brackney and R. H. Anderson, Eds., *Understanding the Insider Threat*. RAND Corporation, March 2004.

[8] M. Schonlau, W. DuMouchel, W.-H. Ju, A. F. Karr, M. Theus, and Y. Vardi, "Computer intrusion: Detecting masquerades," *Statistical Science*, vol. 16, no. 1, pp. 1–17, 2001.

[9] R. A. Maxion, "Masquerade detection using enriched command lines," in *Proc. IEEE International Conference on Dependable Systems & Networks (DSN)*, 2003, pp. 5–14.

[10] P. Parveen, N. McDaniel, B. Thuraisingham, and L. Khan, "Unsupervised ensemble based learning for insider threat detection," in *Proc. of 4th IEEE International Conference on Information Privacy, Security, Risk and Trust (PASSAT), Amsterdam, Netherlands*, September 2012.

[11] P. Parveen and B. Thuraisingham, "Unsupervised incremental sequence learning for insider threat detection," in *Proc. IEEE International Conference on Intelligence and Security (ISI), Washington DC*, June 2012.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, 2008.

[13] M. M. Masud, T. Al-Khateeb, K. W. Hamlen, J. Gao, L. Khan, J. Han, and B. M. Thuraisingham, "Cloud-based malware detection for evolving data streams," *ACM Trans. Management Inf. Syst.*, vol. 2, no. 3, p. 16, 2011.

[14] A. Haque, B. Parker, and L. Khan, "Labeling instances in evolving data streams with mapreduce," in *BigData*, 2013.

[15] M. F. Husain, J. P. McGlothlin, M. M. Masud, L. R. Khan, and B. M. Thuraisingham, "Heuristics-based query processing for large rdf graphs using cloud computing," *IEEE Trans. Knowl. Data Eng.*, vol. 23, no. 9, pp. 1312–1327, 2011.

[16] S.-L. Chua, S. Marsland, and H. W. Guesgen, "Unsupervised learning of patterns in data streams using compression and edit distance," in *IJCAI*, 2011, pp. 1231–1236.

[17] T. A. Welch, "A technique for high-performance data compression," *IEEE Computer*, vol. 17, no. 6, pp. 8–19, 1984.

[18] L. Vladimir, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[19] S. Greenberg, "Using unix: Collected traces of 168 users," in *Research Report 88/333/45, Department of Computer Science, University of Calgary, Calgary, Canada*, 1988, http://grouplab.cpsc.ucalgary.ca/papers/.