SAND2015-8150J

Statistical Techniques For Real-time Anomaly Detection Using Spark Over Multi-source VMware Performance Data

Mohiuddin Solaimani, Mohammed Iftekhar, Latifur Khan, Bhavani Thuraisingham

Department of Computer Science
The University of Texas at Dallas
Richardson, TX, USA
{*mxs121731, mxi110930, lkhan, bhavani.thuraisingham*}*@utdallas.edu*

Joe Ingram

Sandia National Laboratories
Albuquerque, NM, USA
jbingra@sandia.gov

Abstract

Anomaly detection refers to the identification of an irregular or unusual pattern which deviates from what is standard, normal, or expected. Such deviated patterns typically correspond to samples of interest and are assigned different labels in different domains, such as outliers, anomalies, exceptions, or malware. Detecting anomalies in fast, voluminous streams of data is a formidable challenge.

This paper presents a novel, generic, real-time distributed anomaly detection framework for heterogeneous streaming data where anomalies appear as a group. We have developed a distributed statistical approach to build a model and later use it to detect anomaly. As a case study, we investigate group anomaly detection for a VMware-based cloud data center, which maintains a large number of virtual machines (VMs). We have built our framework using Apache Spark to get higher throughput and lower data processing time on streaming data. We have developed a window-based statistical anomaly detection technique to detect anomalies that appear sporadically. We then relaxed this constraint with higher accuracy by implementing a cluster-based technique to detect sporadic

and continuous anomalies. We conclude that our cluster-based technique outperforms other statistical techniques with higher accuracy and lower processing time.
Keywords: Real-time anomaly detection, Chi-square test, multinomial goodness-of-fit test, Resource scheduling, Data center, Apache Spark

1. Introduction

Anomalies do not conform to the normal or expected behavior. It inevitably signifies some exceptional occurrence or abnormalities in need of corrective measures. Real-time anomaly detection [1, 2, 3, 4, 5, 6] aims to capture abnormalities in system behavior in real-time. They may appear in the form of malicious network intrusions [5, 7], malware infections, abnormal interaction patterns of individuals or groups in social media [8], over-utilized system resources due to design defects, etc. It is a challenging task to efficiently process and detect anomalies from continuous, high-volume, high-speed streams of data in real-time. Failure to accomplish this task in a timely manner may lead to catastrophic consequences with a severe impact on business continuity.

A real-time anomaly detector could be the heart of dynamic resource scheduling in enterprise data centers. Virtualizaion techniques enable data centers to host hundreds to thousands of virtual machines (VM). Dynamic resource scheduling plays a crucial role for the operational efficiency of these data centers. In response to varying demands for various resources–e.g., CPU, memory, I/O–the scheduler must allocate or re-allocate resources dynamically. This necessitates real-time monitoring of resource utilization and performance data for the purpose of detecting abnormal behavior.

A real-time anomaly detector for data centers requires scalable frameworks capable of handling extremely large amounts of data (so called Big Data) with low latency. Big Data frameworks (e.g., Hadoop [9], MapReduce [10], HBase [11], Mahout [12], Google Bigtable [13]) are highly scalable to accommodate massive data sets but lack real-time processing capabilities. Apache Storm [14]

and Apache S4 [15] are distributed frameworks that can process stream data. Apache Spark [16, 17] is another distributed framework that offers streaming integration with time-based in-memory analytics for the live data as they come in stream. Spark runs a streaming computation as a series of micro batch jobs. It minimizes batch size in order to achieve low latency. It keeps job states in memory between batches so that they can be recovered without significant delay. Spark has an advanced DAG (Direct Acyclic Graph) execution engine that supports cyclic data flow and in-memory computing. This makes Spark faster than other distributed frameworks. Spark is 100 times faster than Hadoop MapReduce in memory, or 10 times faster on disk [16]. It is also faster than Storm and S4 [18]. Overall, it generates low latency, real-time results.

We have identified some key challenges in our real-time anomaly detection framework: (i) Identifying a data point as anomalous may increase the false alarm when, for example, sudden spike in any resource usage due to temporal hardware malfunctioning in a data center may not indicate its anomalous or over usage condition. A simple extension can be detecting anomalies on windowed data instead of individual data points. As data comes in stream, we set a time window to collect data from the stream periodically. This approach requires setting a time period for the window and collecting data within that window. Later, a non-parametric histogram has been used to generate the distribution for the window. If a window's distribution substantially deviates from the model of longer-term benign distributions, the window is flagged as anomaly. (ii) Heterogeneous data co-exists in the data center. For example, each VM in the data center generates separate statistics for each of its resources (e.g., CPU, memory, I/O, network usages statistics). Their balanced usages exhibit the data center's operational efficiency. When demands for various resources go high, their balanced usages will no longer exist. Therefore, we capture these balanced usage distributions. Later, we build a separate model of distributions for each resource from them. We have observed that there exists no relation among distributions of difference resources. For example, higher CPU usage may not increase the memory usage & vice versa. So, we considered no rela-

tion among different resources when we build models. (iii) Synchronized data communication and efficient parallel design are the main focuses for designing distributed real-time anomaly detection framework [19, 20]. We use a message broker (Apache Kafka [21]) to transfer the data to the distributed framework without any loss of data. We set a fixed time interval to capture and process data from the message broker.

Considering the above challenges, we have made the following contributions in this paper:

- We have developed a novel, generic, window-based statistical real-time framework for heterogeneous stream data using Apache Spark [16, 17] and Kafka [21].
- We have developed a novel cluster-based statistical framework for heterogeneous stream data using Apache Spark and Kafka. We have set a time window to capture data and generated a non-parametric distribution for each of the feature in data. We have built a model of benign distributions for each feature in training and used that to identify anomaly during testing. We will consider a window as anomaly if any of its feature's distribution fails to fit its corresponding model during testing.
- We have implemented the distributed two-sample Chi-square test in Apache Spark where each distribution has unequal number of data instances.
- We have compared our cluster-based approach with our previous window-based approach [22] and also another online statistical method proposed by Chengwei *et al.* [23]. It uses a multinomial goodness of fit test where each window has a fixed number of data instances. We use Chi-square two sample test where the number of data instances vary. Moreover, in our scenario, anomalies may be distributed across multiple windows. Our cluster-based approach outperforms both approaches with higher TPR (true positive rate) because they assume that anomaly will be distributed within a window.

The rest of the paper is organized as follows: Section 2 describes our statistical real-time anomaly detection frameworks in detail. Section 3 shows our case study of real-time anomaly detection on VMware performance data using Apache Spark with implementation details. Section 4 presents our experimental results with our framework's accuracy. Section 5 discusses related works. Finally, Section 6 concludes our paper and discusses future work.

2. Real-time Anomaly Detection Framework

In real-time anomaly detection, preprocessing of raw stream data serves as a precursor to machine learning or data mining algorithms. K-means is a popular unsupervised data mining technique for stream data analytics that builds clusters of data points from training data with the assumption that the clusters are benign. If a test data point falls into any one of the clusters, it is labeled as benign, otherwise it is flagged as anomalous. In order to accommodate the evolution of stream data, the training model needs to be updated periodically.

Data center automation [24] (e.g., dynamic resource management) may require analyzing the performance data in real-time to identify anomalies. As the stream data come continuously and the volume of data is also huge, we require a scalable distributed framework. To address the scalability issue, we can use a distributed solution like Hadoop or MapReduce. Hadoop runs on batch mode and cannot handle real-time data. As we are looking for real-time data analysis in a distributed framework, we have decided to use Apache Spark [16], which is fault-tolerant and supports distributed real-time computation system for processing fast, large streams of data.

Apache Spark [16] is an open-source distributed framework for data analytics. It avoids the I/O bottleneck of the conventional two-stage MapReduce programs. It provides the in-memory cluster computing that allows a user to load data into a cluster's memory and query it efficiently. This increases its performance faster than Hadoop MapReduce [16].

Spark has two key concepts: Resilient Distributed Data set (RDD) and

directed acyclic graph (DAG) execution engine.

- *Resilient Distributed Data set (RDD)*

  RDD [25] is a distributed memory abstraction. It allows in-memory computation on large distributed clusters with high fault-tolerance. Spark has two types of RDDs: parallelized collections that are based on existing programming collections (like list, map, etc.) and files stored on HDFS. RDD performs two kinds of operations: transformations and actions. Transformations create new data sets from the input or existing RDD (e.g. map or filter), and actions return a value after executing calculations on the data set (e.g. reduce, collect, count, saveAsTextFile, etc.).

- *Directed acyclic graph (DAG) execution engine*

  Whenever the user runs an action on RDD, a directed acyclic graph is generated considering all the transformation dependencies.

Spark supports both batch process and also processing of streaming data [26]. Its streaming component is highly scalable, and fault-tolerant. It uses a micro-batch technique which divides the input stream as a sequence of small batched chunks of data for a small time interval. It then delivers these small packed chunks of data to batch system to be processed.

Spark Streaming has two types of operators:

- *Transformation operator*

  It creates a new DStream [18] from one or more parent streams. It can be either stateless (independent on each interval) or stateful (share data across intervals).

- *Output operator*

  It is like action operator and allows the program to write data to external systems (e.g., save or print DStream).

Like MapReduce, map is a transformation function that takes each data set element and returns a new RDD. On the other hand, reduce is an action

function that aggregates all the elements of the RDD and returns the final result (reduceByKey is an exception that returns a RDD).

2.1. Chi-square test

Given two binned data sets, let R_i be the number of items in bin i for the first data set and S_i be the number of items in bin i for the second data set. The Chi-square statistic is:

$$\chi^2 = \sum_i \frac{(\sqrt{S/R} \times R_i - \sqrt{R/S} \times S_i)^2}{R_i + S_i} \tag{1}$$

where $R \equiv \sum_i R_i$ and $S \equiv \sum_i S_i$.

It should be noted that the two data sets can be of different sizes. A threshold T is computed against which the test statistic χ^2 is compared. T is usually set to that point in the Chi-squared cdf with N_{bins} degrees of freedom(for data sets of unequal sizes) that corresponds to a confidence level of 0.95 or 0.99 [23]. If $\chi^2 < T$ the data sets follow the same distribution.

2.2. Statistical anomaly detection

A *window* W can be defined as a data container which collects data $d_1,\ d_2,\ ...,\ d_n$ from a stream periodically at a fixed time interval t from multiple VMs (virtual machines) in a data center. The length of a Window is the total number of data instances that it contains within that time interval.

A feature f is a property to describe a physical computing hardware or resource used in data center. For example, CPU usage, memory usage, I/O block transfer, etc.

Clustering techniques aim to classify a single data point. In cases of stream data, it seems more appropriate to consider a point as part of a distribution and determine its characteristics (anomaly/benign) with reference to the remaining points in the distribution. Statistical approaches are a natural fit for this kind of scenario, since they offer ways to compare two distributions. Parametric statistical techniques assume prior knowledge of the data distribution and verify

whether the test distribution conforms to the known model. Non-parametric techniques, on the other hand, make no assumptions on the probability distribution of data. Hence, non-parametric techniques are less restrictive than parametric ones, and have wider applicability compared to their parametric counterparts [27, 28].

Algorithm 1 Statiscal anomaly detection

```
1:  procedure DETECTANOMALIES(windows, TH, C_TH)
2:      w_i ← dataOf i^th TimeInterval
3:      AddToWindows(windows,w_i)
4:      status ← anomalous
5:      for j ← i − 1 to i − N + 1 do
6:          T_j ← ComputeChi-SqrStatistic(w_i,w_j)
7:          T ← T ∪ T_j
8:      T_min = argmin_j{j | T_j ∈ T}
9:      if T_min < TH then
10:         c_j ← c_j +1
11:         if c_j > c_TH then
12:             status ← benign
```

In Algorithm 1, we provide an outline of the anomaly detection process by a non-parametric statistical method. It periodically collects data from incoming stream. Data collected at the i^{th} time interval will be stored in the i^{th} window w_i. Each window has a score that represents the number of times this window matches with another windows. w_i is compared against past *N - 1* windows using a Chi-square test (line 6); the test statistics from this test are stored in a list *T*. If the list is empty, window w_i is anomaly. Otherwise, we choose the window that has minimum test statistics T_{min}(line 8). If the test statistic T_{min} is below a pre-defined threshold, the score c_j of the j^{th} window w_j is incremented. If c_j exceeds a pre-defined threshold C_{TH}, w_i is declared as benign. w_i is flagged as anomalous if we do not find any window whose score exceeds C_{TH} (line 9-12).

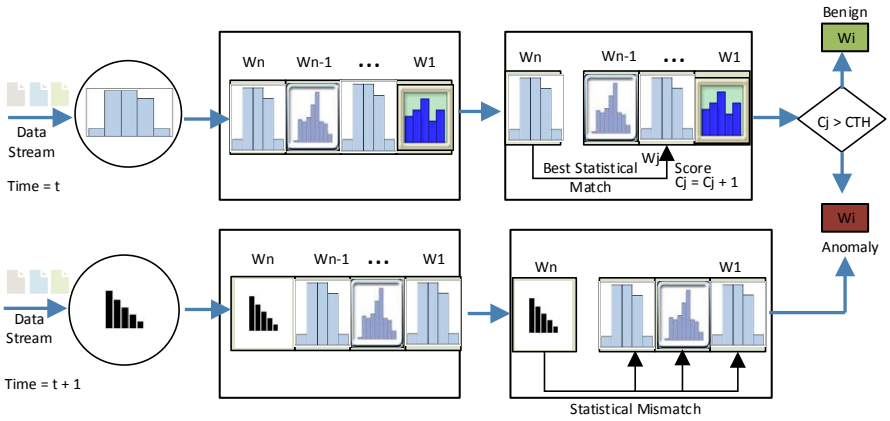

Figure 1: Statistical technique for anomaly detection

In Figure 1, a sliding buffer always stores the recent N windows. Each window has a distribution. When a window comes at time t, it is inserted as a most recent window w_n. It is matched with distributions of the past $n-1$ windows. A Chi-square two sample test has been used to check if two distributions are same or not. It gives the Chi-square statistics. The lower statistic indicates the closest match. If the window w_n matches window w_j, then its score c_j is incremented. The score defines the number of window matches. This technique assumes that the stream data will not change abruptly. So, anomalies will appear rarely and will not come consecutively. If a window has high score, then it represents a benign window. Moreover, a new window matches this high scored window will also be considered as a benign window. We set a threshold C_{TH}. If $c_j > C_{TH}$, then w_n is benign, else it is anomaly. At time $t+1$, a window w_n comes and if it does not find any closest distribution from the past $n-1$ windows, it is immediately termed as anomaly.

In the real word, consecutive anomalous windows may come frequently. Moreover, because of using the past N windows, some benign windows may

be interpreted as anomalous, increasing the false positive rate. To overcome these, we have introduced a cluster-based statistical technique, where we have applied unsupervised clustering on benign distributions during training to build our model. Later, we use this model to detect anomaly while testing. We use a Chi-square two sample test to compare two distributions. During testing, if the distribution of a window fails to match with any distribution in the training model, we will consider it an anomaly.

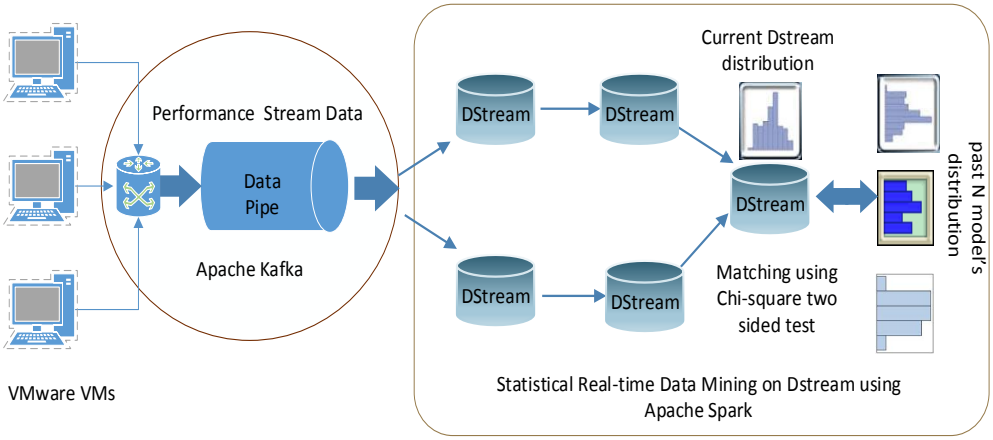

Figure 2: Technical Approach

2.3. Statistical Stream Data Mining Module Implementation Using Spark

Figure 2 shows the statistical real-time framework for anomaly detection using Apache Spark. Each virtual machine in the network continuously sends its performance data through Kafka. The Spark framework receives the stream as small micro batches called a DStream. After that, it uses several transformation and action operations to build the statistical model. Here, we have focused more on a set of data points in a window rather than an individual data point. An anomalous window carries more information and represents a pattern or distribution of abnormal characteristics of data. After receiving a window, the

framework generates a non-parametric distribution. It stores the most recent distribution. Later anomalies will be detected by matching the current window's distribution with its stored distributions.

In algorithm 2, DETECTANOMALY illustrates our distributed statistical procedure for anomaly detection using Spark. It describes a general procedure to find anomalies for each feature. In this case, we say a window is benign if it is not anomalous for all features; otherwise it is anomalous.

i *Generating Non-parametric Distribution*

As we do not have any prior knowledge of data, we have used a bin-based histogram technique to derive this non-parametric distribution by using the following formula.

$$Bin_{current} = \left\lceil \frac{d - d_{min}}{d_{max} - d_{min}} \times totalBin \right\rceil \tag{2}$$

where d_{min} and d_{max} are the minimum and maximum value of the data d. These can be calculated experimentally. We have used percentage of resource usage like CPU / memory usage. So, their minimum and maximum values are 0 and 100.

GETDISTRIBUTION in algorithm 3, describes how to generate distribution. Here, we have used Spark's map and reduce functions. The map function (lines 5-8) takes an index i as key and each instance of the DStream D_i as value. Later it uses equation 2 to generate appropriate bin for each feature and finally emits (bin , frequency) as output. The reduce function (lines 9-10) takes bin as key and frequency list as value. It counts the total number of frequencies for each bin of the corresponding feature. Finally, it returns a list of bin with its frequency.

ii *Chi-square Test*

The Chi-square Test is the ideal non-parametric technique to match two unknown distributions. Here, we are collecting data as a window. VMs

Algorithm 2 Anomaly detection

1: **procedure** DETECTANOMALY

Inputs

2: $InputDStream\ D$ (n data, l $features$)

3: $Window\ W$

4: $models_l \leftarrow \{\}$ ▷ empty models collection

5: $bin_{k,l} \leftarrow$ GETDISTRIBUTION($InputDStream$, $totalBin$)

6: $m_{l,c} \leftarrow createModel(bin_{k,l},\ score = 1)$

7: $models_l.insert(m_{l,c})$ ▷ insert current model to models collection

8: **if** $models.size = 1$ **then**

9: $current\ first\ model\ is\ benign$

10: **else**

11: $chi - models_l \leftarrow$ MODELSWITHCHIVALUE($bin_{k,l}$, $models_l$)

12: $(m_{l,c}, (m_{l,match}, ch_l)) \leftarrow$ GETMATCHEDMODEL($chi - models_l$, $m_{l,c}$)

13: $W \leftarrow benign$

14: **for** $j \leftarrow 1$ to l **do**

15: **if** $m_{j,match}.isEmpty()$ **then**

16: $current\ model\ m_{j,c}\ is\ anomaly$

17: $W \leftarrow anomaly$

18: **else**

19: **if** $m_{j,match}.getScore() > Sc_{TH}$ **then**

20: $current\ model\ m_{j,c}\ is\ benign$

21: **else**

22: $current\ model\ m_{j,c}\ is\ anomaly$

23: $W \leftarrow anomaly$

24: **return** W

Algorithm 3 Generate non-parametric distribution

1: **procedure** GETDISTRIBUTION

Inputs

2: $InputDStream\ D\ (n\ data,\ l\ features)$

3: $totalBin$

Outputs

4: $List\ (bin,\ f)$ ▷ f = frequency

5: **Map** $(i,\ D_i),\ where\ i = 1\ to\ n$

6: **for** $j \leftarrow 1\ to\ l$ **do**

7: $bin_{k,j} \leftarrow \left\lceil \frac{D_{ij} - D_{jmin}}{D_{jmax} - D_{jmin}} \times totalBin \right\rceil$, $where\ k \in 1\ ..\ totalBin$

8: **collect**$(bin_{k,j},\ 1)$

9: **Reduce** $(bin_{k,j},\ [f_1, f_2, ...]),$ $where\ k = 1\ to\ totalBin,\ j = 1\ to\ l$ ▷ f = frequency

10: **return sum**$([f_1, f_2, ...])$

send their data as a stream. Due to network delay or packet loss, some data might not reach the receiving end at their scheduled time. So, the window size of the Spark framework may vary. Therefore, we use Chi-square test for matching two distributions where the number of instances are not equal.

Figure 2 shows the matching of the current distribution with its recent $N-1$ distributions. After generating the distribution, we store it as a model (line 5-6 in algorithm 2) for each feature. This is our current model and we score it as 1. We assume our first model as benign. If the current model is not the first model, then we compute Chi-square value for the rest of the models and list them (line 11 in algorithm 2). In algorithm 4, MODELSWITHCHIVALUE computes and lists the model for each feature with their Chi-square values. The map function (lines 5-13) takes bin and frequency as a key-value pair. It extracts each bin of a feature and its frequency from the tuple. Later, it computes the frequencies of that bin for that feature from all the past $N-1$ models. After that, it calculates the partial Chi-square statistics for that

Algorithm 4 Models with Chi-square value

1: **procedure** ModelsWithChiValue

Inputs

2: $List\ (bin,\ f)$ ▷ f = frequency

3: $Models\ M_{l,n}$ (l *models, each has* n *distributions*)

Output

4: $Chi - Models\ M_{l,n-1}$(l *models, each has* $n-1$ *chi values*)

5: **Map** $(bin_{k,j},\ f_{k,j})$, *where* k *= 1 to totalBin,* j *= 1 to* l

6: $m_{j,current} \leftarrow M_{j,n}$ ▷ current model for feature j

7: $R_T \leftarrow m_{j,current}.getTotalFreq$

8: $R \leftarrow f_{k,j}$ ▷ frequency of $bin_{k,j}$

9: **for** $i \leftarrow 1$ *to* $n-1$ **do**

10: $S_T \leftarrow M_{j,i}.getTotalFreq$

11: $S \leftarrow M_{j,i}.getBinFreq(bin_{k,j})$

12: $chi\_sqr \leftarrow \frac{(\sqrt{S_T/R_T} \times R - \sqrt{R_T/S_T} \times S)^2}{R_T + S_T}$

13: **collect**$(M_{j,i},\ chi\_sqr)$

14: **Reduce** $(M_{j,i},\ [c_1, c_2, ...])$, *where* j *= 1 to* l, i *= 1 to n-1*

▷ c = chi-square value

15: **return** **sum**$([c_1, c_2, ...])$

bin of corresponding model. Finally, it emits the (model, Chi-square value) as a tuple list. The reduce function (lines 14-15) receives model as a key and Chi-square value list as a list of value and sums up the Chi-square statistics for that model. At the end, we will get Chi-square statistics for all models.

Now, we have calculated all the feature model's Chi-square statistics and we have to filter them by using a Chi-square threshold TH-CHI. GetMatchedModel (line 12 in algorithm 2) finds each feature's model which has the lowest Chi-square value. Here, we can use any statistical tool like R [29] to find out this threshold. R has Chi-squared test of comparing two distributions which assumes they have equal number of data instances but in our case, we have unequal number of data instances. $qchisq(cf, df)$ in R gives the threshold, where cf = confidence level and df = degrees of freedom. Here, $df = Number\ of\ bins$. In algorithm 5, GetMatchedModel, the map function (lines 5-8) takes model and Chi-sqaure value as key-value pair. It also knows the current model. It finds all the possible closest models using chi-square threshold and emits current model with minimum matching Ch-square value as a key value pair. The reduce function (lines 7-8) receives current model with a list of Ch-square values as key-value list and returns the model which has minimum Chi-square statistic. When we match our current model with past models, several models have lower Chi-square statistics than the threshold. So, we have to select that model which has the lowest statistics. In algorithm 6, MinModel (line 1-12) selects the model which has the lowest Chi-square statistics. If several models have the same lowest Chi-square statistics, then it selects the model which has highest score. The score means how many times it matches a model. Initially all models have a score = 1. If any past model is matched by current model, then we increase the score of that past model.

iii *Detecting Anomaly*

The current window has multiple distributions of different features. Therefore, the window will be flagged as benign if all its distribution are benign,

otherwise it is anomaly. Lines 13-24 in algorithm 2, describe this.

Algorithm 5 Find closest model

1: **procedure** GETMATCHEDMODEL

Inputs

2: $Chi - Models\ M_{l,n-1}(l\ models,\ each\ has\ n-1\ chi\ values\ c)$

3: $CurrentModel\ m_{l,c}(l\ models)$

Outputs

4: $ModelsWithLowestChiValue\ (m_{l,min},\ \ c_{l,min}),$ where j = 1 to l

5: **Map** $(M_{j,i},\ c_{j,i}),\ where\ j = 1\ to\ l, i = 1\ to\ n-1$

6:

7: **if** $c_{j,i} \leq TH - CHI_j$ **then** ▷ Calculated in R for 95% confidence

8: **collect**$(m_{j,c},\ \ (M_{j,i},\ \ c_{j,i}))$

9: **Reduce** $(m_{j,c},\ [(M_{1,1},\ c_{1,1}), ...]),$

10: $(m_{j,min}, c_{j,min}) \leftarrow MINMODEL([(M_{1,1},\ c_{1,1}), ...])$

11: $m_{j,min}.incrementScore()$

12: **return** $(m_{j,min}, c_{j,min})$

2.4. Cluster-based Statistical Stream Data Mining Module Implementation Using Spark

In Figure 3, we show the cluster-based statistical real-time framework for anomaly detection using Apache Spark. Each virtual machine in the network continuously sends its performance data through Kafka. We cluster on the benign distributions and build a model, which will be used later to detect anomaly.

In algorithm 7, we illustrate our distributed statistical procedure for anomaly detection using Spark. It uses an semi–supervised technique for training unknown distributions. We assume that all VMs are in stable state and their performance data do not contain anomalies in training. We have taken a time window to collect all the data and build non-parametric distribution for each feature. During clustering, we build a training model for each feature. If a

Algorithm 6 Model with minimum Chi-square value

procedure MinModel

Inputs

[(m_1, c_1), (m_2, c_2) ..] (*List of models with Chi − square value*)

Outputs

(m_{min}, c_{min}) (*matched model with min with Chi − square value*)

$c_{min} \leftarrow \infty$

for *each (m, c) in [(m_1, c_1), (m_2, c_2) ..]* **do**

  $sc \leftarrow m.getScore()$

  if $c < c_{min}$ **then**

    $(c_{min}, m_{min}, sc_{high}) \leftarrow (c, m, sc)$

  else

    if $c = c_{min}$ **and** $sc > sc_{high}$ **then**

      $(m_{min}, sc_{high}) \leftarrow (m, sc)$

  return (m_{min}, c_{min})

Algorithm 7 Training

procedure Training

Inputs

InputDStream D (n data, l *features*)

$models_l \leftarrow \{\}$

$bin_{k,l} \leftarrow$ GetDistribution($InputDStream$, $totalBin$)

$m_{l,c} \leftarrow createModel(bin_{k,l}, score = 1)$

$chi - models_l \leftarrow$ ModelsWithChiValue($bin_{k,j}$, $models_l$)

$(m_{l,c}, (m_{l,match}, ch_l)) \leftarrow$ GetMatchedModel($chi - models_l$, $m_{l,c}$)

if $m_{l,match}.isEmpty()$ **then**

  $models_l.insert(m_{l,c})$

else

  $m_{l,match}.incrementScore()$

Algorithm 8 Testing

1: **procedure** TESTING

Inputs

2: $InputDStream\ D$ (n data, $l\ features$)

3: $Window\ W$

4: $models_l \leftarrow \{\}$

5: $bin_{k,j} \leftarrow$ GETDISTRIBUTION($InputDStream$, $totalBin$)

6: $m_{l,new} \leftarrow createModel(bin_{k,l}, score = 1)$

7: $chi - models_l \leftarrow$ MODELSWITHCHIVALUE($bin_{k,j}$, $models_l$)

8: $(m_{l,new}, (m_{l,match}, ch_l)) \leftarrow$ GETMATCHEDMODEL($chi - models_l$, $m_{l,c}$)

9: $W \leftarrow benign$

10: **for** $j \leftarrow 1$ *to* l **do**

11: **if** $m_{j,match}.isEmpty()$ **then**

12: *current model* $m_{j,new}$ *is anomaly*

13: $W \leftarrow anomaly$

14: **else**

15: **if** $m_{j,match}.getScore() > Sc_{TH}$ **then**

16: *current model* $m_{j,new}$ *is benign*

17: **else**

18: *current model* $m_{j,new}$ *is anomaly*

19: $W \leftarrow anomaly$

20: **return** W

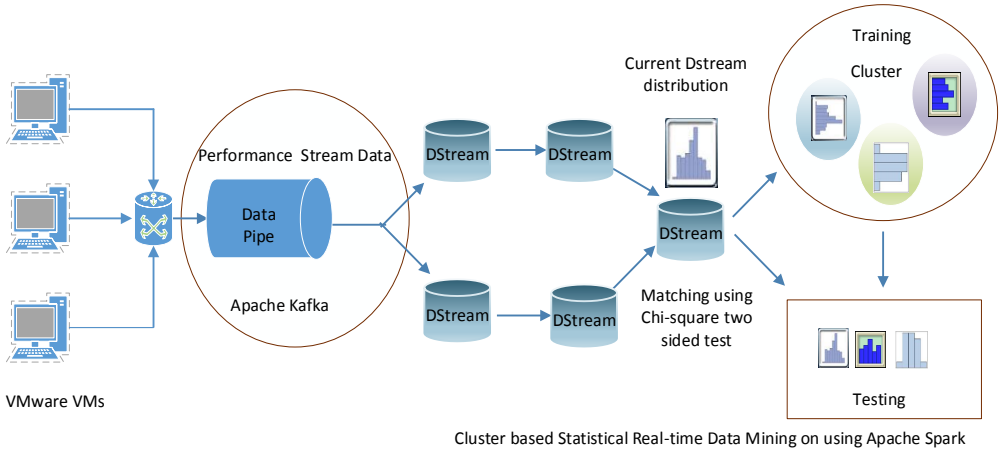

Figure 3: Technical Approach

distribution does not fit to any cluster, it starts with a new cluster. If it fits to a cluster, then we increment the score of that cluster. The score describes the confidence level of the cluster. TRAINING illustrates the details.

During testing, we take a time window to collect data and generate the unknown distribution for each feature. Later, if the distribution does not fit to any clusters of the relevant feature, then it is called anomalous, otherwise, it is benign. We can say a window is benign if it is not anomalous for that feature; otherwise it is called anomalous. In algorithm 8, TESTING illustrates the details.

3. Case Study: Real-Time Anomaly Detection In VMware-based Data Center

We have developed a generic framework using a statistical technique. We can implement this framework in variety of fields like in social media, real-time traffic analysis, intrusion detection in sensor data, and so on. In this case, an anomaly corresponds to over-utilization of the resources in a data center. More precisely, we can say that an abnormal distributions of its resources usage. For

example, when CPU or memory intensive applications are running, the overall CPU or memory usage goes high and these generate abnormal distributions, which can be considered as anomalous.

In this section, we will describe data centers and dynamic resource scheduling and later our framework implementation details.

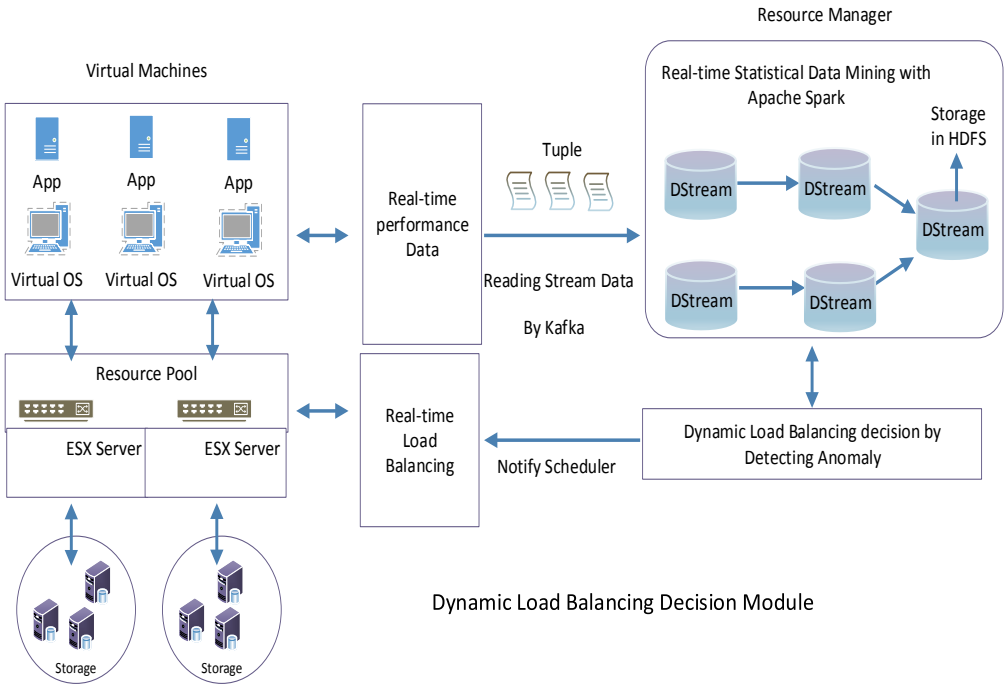

Figure 4: Dynamic resource management using Apache Spark

3.1. Data Center

A data center is the store house of data. It provides computer systems and associated components, such as telecommunications and storage systems. It offers dynamic resource management software to manage the storage infrastructure and also provides cloud facilities with the help of VMware, OpenStack, Microsoft, etc.

3.2. Dynamic Resource Scheduler

In Figure 4, we have shown the data flow for VMware dynamic resource management. The resource manager periodically reads the VMware performance data and sends it to the Spark cluster model to analyze it. The resource manager then sends the data analysis information to the resource pool [30] to dynamically allocate resources if necessary.

3.3. Implementation Details

3.3.1. VMware Cluster Setup

Our VMware cluster consists of 5 VMware ESXi [31] (VMware hypervisor server) 5.5 systems. Each of the systems has Intel(R) Xeon(R) CPU E5-2695 v2 2.40GHz processor, 64 GB DDR3 RAM, 4 TB hard disk and dual NIC card. Each processor has 2 sockets and every socket has 12 cores. So there are 24 logical processors in total. All of the ESXi systems contain 3 virtual machines. Each of the virtual machines is configured with 8 vCPU, 16 GB DDR3 RAM and 1 TB Hard disk. As all the VM's are sharing the resources, performance may vary in run time. We have installed Linux Centos v6.5 64 bit OS in each of the VM along with the JDK/JRE v1.7. We have designed a real-time outlier detection system on this distributed system using Apache Spark. We have installed Apache Spark version 1.0.0. We have also installed Apache Hadoop NextGen MapReduce (YARN) [32] with version Hadoop v2.2.0 and formed a cluster. We ran multiple MapReduce jobs on Hadoop cluster to put extra load so that we can monitor the performance data of the VMware-based cluster.

3.3.2. VMware Performance Stream Data

It is imperative for Data Center operations team to be able to track resource utilization of various hardware and software in order to maintain high quality of service and client satisfaction. The performance counters generated by the data center operation management system need to be of sufficient granularity to facilitate the detection of anomalies.

In this experiment, we have used percentage of CPU usage and percentage of memory usage to build our statistical model. Our framework can accommodate further attributes as well. We resorted to *top* [33], a Linux tool which reports CPU related statistics, *vmstat* [34] for memory related statistics and we integrated their output with that of the *vSphere Guest SDK* [35] using Kafka API.

3.3.3. Message Broker

We are continuously monitoring each VMware performance data using the vSphere Guest SDK [35]. We have also integrated top [33] to get the current percentage of CPU usage and vmstat [34] to get the current percentage of memory usage. Several message brokers (e.g., Apache Kafka [21], RabbitMQ [36], etc.) are available to integrate with Spark. We have chosen Kafka 3.3.4 because it is stable and also compatible with Apache Spark. Kafka creates a dedicated queue for message transportation. It supports multiple source and sink on the same channel. It ships (in Figure 2) those performance data to Spark's streaming framework. Kafka ensures that the messages sent by a producer to a particular topic partition are delivered to the consumer in the order they are sent. In addition to high-speed, large-scale data processing, Kafka clusters offer safeguards against message loss due to server failure. Kafka is a natural fit for our real-time anomaly detection initiative where it serves as an intermediary between multiple VMware virtual machines and the Spark cluster, transferring multi-source stream data from the VMs to Spark.

3.4. Building Statistical Training Model and Prediction

Our framework captures CPU and memory data periodically and generates distributions for each of them. It then clusters the distributions separately for both CPU and memory usages during clustering. While testing, it again generates distributions for both CPU and memory usages. If any of the distribution does not fit into the cluster of distributions, then we consider them as anomaly. More details are described in Section 2.

3.5. Scalability

Our framework can easily adapt to an increasing number of VMware virtual machines. For each new VM, we simply plug in one more data collection instance to Kafka [21]. Furthermore, we can increase computational parallelism by adding more threads to Spark executors/workers.

4. Experimental Results

4.1. Data set

We continuously monitor 12 VMs' performance data. Each VMware sends data to Kafka server at twelve second intervals. Apache Spark periodically collects stream data from Kafka server at two min intervals. The length of the window is not fixed here. The number of data points in each windows varies from 95-105. In this experiment, we have trained our model with clustering by 200 windows. So, 200 x 100 (average) = 20,000 data points have been considered during training. We collect 400 windows during testing both for our cluster-based statistical method and window-based method. So, 400 x 100 (average) = 40,000 data points have been considered during testing.

We have compared our result with a non-distributed, online statistical approach [23]. For this, we dump our real-time testing data to a CSV file. It uses a fixed window and it uses multinomial goodness-of-fit [37] for statistical analysis while we are using a Chi-square two-sided test.

In this experiment, we consider any VM resource hungry if its usage goes beyond 95%. For example, if a VM has more than 95% CPU usage, then it is called CPU hungry.

4.2. Results

Table 1 shows the training statistics. We use 200 windows for training each window has 100 data points on an average. We have found seven clusters for CPU usage and five clusters for memory usage.

Table 1: Training

Number of Windows	200
Number of data points	20,000
Number of cluster	7(CPU), 5 (Memory)

Next, we have used our data set for both non-distributed and distributed experiments during testing. The online distributed method is used to detect anomaly for cluster-based statistical model and window-based model. The non-distributed method is used for online multinomial goodness-of-fit based model. Table 2 shows the details.

Table 2: Testing

	Distributed	Non-distributed
Number of Windows	400	400
Number of data points	40,000	40,000

Table 3: Accuracy

Data set	TPR	FNR	TNR	FPR
Multinomial goodness-of-fit based model	20.00%	80.00%	82.16%	17.84%
Window-based model	60.00%	40.00%	99.80%	00.20%
Cluster-based model	96.00%	03.33%	95.67%	04.32%

Our framework has higher accuracy to identify anomaly. In table 3, TPR is the proportion of actual anomaly which are correctly identified and TNR is the proportion of actual benign which are correctly identified. Moreover, FNR is the proportion of actual anomaly which are misclassified as benign and FPR is the proportion of actual benign which are misclassified as anomaly. We take 20 bins for building histogram/distribution and 400 windows out of which 30 are anomaly. We calculate the Chi-square threshold $TH_{chi} = 31.41$. We have

used the function $qchisq(0.95, df = 20) = 31.41$ in R [29], where confidence level = 95% and degrees of freedom = $numberofbins$. Here, we assume that an anomaly can be spread across multiple windows. During testing, we found that our framework can successfully detect most of the injected anomalies. Only 1 of them is misclassified and the rest 29 are correctly classified. Moreover, it also detects 354 benign windows as benign and misclassifies only 16 as anomaly.

On the other hand, our window-based statistical model correctly identifies 18 anomaly out of 30 although it correctly classifies 369 benign out of 370. The reason of its low accuracy is that it assumes anomaly will come abruptly and spread within a window and if it comes across multiple windows, then it identifies the first one, but misclassifies the rest of the anomalous windows. The online Multinomial goodness of fit test based approach has low TPR rate. We take only 400 windows and 30 of them are anomalous. It detects only 06 of them as anomaly and misclassifies 24 as benign. Moreover, it detects 304 benign windows as benign and misclassifies 66 as anomaly. It assumes equal window length which may increase some false alarm.

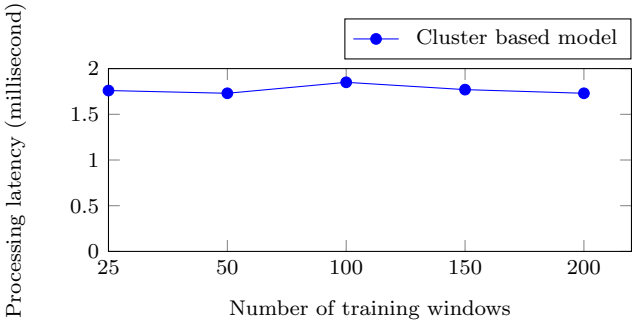

Figure 5: Average window processing latency during training

Figure 5 shows the average execution time per window for 200 windows while training. It has a consistent execution time (on average 1.75ms).

Figure 6 shows the comparison of execution time per window of three approaches during testing. We plot the average execution time for window against the total number of windows. Here, we can see that our cluster-based model

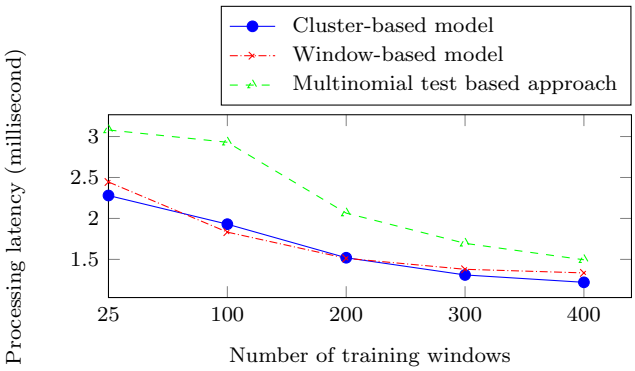

Figure 6: Comparing average window processing latency during testing

and Chi-square-based technique have a lower execution time. It varies from 1.22-2.28ms for cluster-based model and 1.33-2.45ms for window-based model. The base-line approach has average execution time between 3.08-1.49ms. Initially the Multinomial test based approach has higher execution time because it operates in batch mode. It reads the whole file in to memory before doing the computation. So, it has higher I/O time. Therefore, it will affect the average execution time of a window when we have billions of data.

5. Related Work

Our related work covers anomaly detection techniques, then the scalability issue with existing distributed frameworks.

Clustering has been widely used for the anomaly detection problem. K-means has produced better accuracy but it has greater time complexity for a very large data set. K-means also have an initial centroid problem. K-medoids [38] overcomes this problem. It initially selects k centers but the centers are repeatedly changed randomly and thus it improves the sum of squared error. Assent *et al.* [39] have proposed an AnyOut algorithm for detecting outlier on stream data. They use hierarchical clustering and determine an outlier score based on the deviation between object and cluster. They can smoothly detect outliers but they are not addressing the scalability issue. As hierarchical clus-

tering has a complex data structure, it is time consuming if we deploy it in a distributed system.

Yu *et al.* [40] have proposed a non-parametric cluster-based anomaly detection framework in the distributed system. It uses Hadoop[9] and MapReduce. It outperforms existing static anomaly detection techniques. Gupta *et al.* [41] also propose a framework for an anomaly detection system using Hadoop and MapReduce. They extract context from system operational log files. After that, they use K-means to build a model and generate score for anomaly. Apache Mahout [12] is a machine learning tool in a distributed system. It uses Hadoop and MapReduce. It has K-means and StreamingKMeans implementations. These implementations run on batch mode and they are also timeconsuming. So, they are not ideal for clustering real-time data.

All of the above distributed techniques use Hadoop and MapReduce. Hadoop works well in batch where we have the data in advance. It does not work well with real-time systems, where data comes continuously. If we use Hadoop for building the training model, then for each new instance it will repeatedly build the model. That will be a waste of time and resources. So, we cannot use Hadoop with stream or real-time data. Similar distributed systems like HBase [11] and BashReduce [42] have the same problem. Apache Storm [14] and Apache S4 [15] are also available for stream data processing, but Spark is much faster than these [26]. So, we have decided to use Spark [14]. It is scalable and fault tolerant. It has guaranteed data processing and finally it is quite faster than other frameworks like Hadoop, Storm, etc.

Most of the anomaly detection techniques focus on point anomalies but we know that anomalies can form a group. Rose *et al.* [43] implemented a group-based anomaly detection system on social media. Liang *et al.* [44] proposed a Flexible Genry Model (a model that addresses both point and group data characteristic) to capture the group anomaly. All these works have used typical topic models [45] to identify group behavior. Our cluster-based model also captures group behavior but we do not consider the relation among all the resources. For example, we experimentally find that many CPU hungry applications do not

consume higher memory & vice versa. So, we capture all the benign distributions for each resource during training within a particular time frame.

Chengwei *et al.* [23] implements statistical techniques to identify anomalies using a non-distributed framework. They use a multinomial goodness-of-fit test to match two unknown distributions but they assume that, the two distributions have equal size widow. In practice, the window size may vary due to many reasons. In that case, we found their their technique does not give good results. Mohiuddin *et al.* [22] performed real-time statistical anomaly detection for VMware performance data using Apache Spark [16] which performs well under the constraint that a certain window contains all anomalies. Moreover, it also uses a buffer to store past N models. So, an initial benign model might be identified as anomaly over time if the buffer does not contain that model. So it increases false alarms. We demonstrated that the approach proposed in this paper outperforms the one in [22] by addressing those issues. Apache Spark 1.1.0's MLib library [46] supports Pearson's Chi-square goodness of fit test and independence test of paired observations of two variables.

6. Conclusion and Future work

In this paper we have implemented a cluster-based statistical technique for real-time anomaly detection using Apache Spark that uses a Chi-square test as distance metric. It segments performance data streams of VMware virtual machines into windows of variable lengths and performs window-based comparisons against a trained model of benign windows. Our experiments confirm that it is capable of detecting anomalous CPU and memory utilizations of VMware virtual machines that span across multiple windows collected at different time intervals. Our framework is generic and can be adapted to a wide array of use cases. In the future, we intend to apply it to a complex heterogeneous streaming environment where each feature of the heterogeneous data interrelated and we can explore either statistical approaches or graphical model to capture anomaly.

7. Acknowledgement

Funding for this work was partially supported by the Laboratory Directed Research and Development program at Sandia National Laboratories and The National Science Foundation (NSF). Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. NSF grant is contracted under NSF award No. CNS 1229652 and NSF Award No. DUE 1129435.

References

[1] M. Solaimani, L. Khan, B. Thuraisingham, Real-time anomaly detection over vmware performance data using storm, The 15th IEEE International Conference on Information Reuse and Integration, San Francisco, USA.

[2] M. Solaimani, M. Iftekhar, L. Khan, B. Thuraisingham, J. B. Ingram, Spark-based anomaly detection over multi-source vmware performance data in real-time, In the proceeding of 2014 IEEE Symposium Series on Computational Intelligence, Orlando, Florida, USA.

[3] A. Mustafa, M. Solaimani, L. Khan, K. Chiang, J. Ingram, Host-based anomalous behavior detection using cluster-level markov networks, Tenth Annual IFIP WG 11.9 International Conference on Digital Forensics, Vienna University of Technology.

[4] A. Mustafa, A. Haque, L. Khan, M. Baron, B. Thuraisingham, Evolving stream classification using change detection, 10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, October 2225, 2014 Miami, Florida, USA.

[5] Y. Yao, A. Sharma, L. Golubchik, R. Govindan, Online anomaly detection for sensor systems: A simple and efficient approach, Performance Evaluation 67 (11) (2010) 1059–1075.

[6] W. Lee, S. J. Stolfo, P. K. Chan, E. Eskin, W. Fan, M. Miller, S. Hershkop, J. Zhang, Real time data mining-based intrusion detection, in: DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01. Proceedings, Vol. 1, IEEE, 2001, pp. 89–100.

[7] G. R. Abuaitah, Anomalies in sensor network deployments: Analysis, modeling, and detection, Ph.D. thesis, Wright State University (2013).

[8] D. Savage, X. Zhang, X. Yu, P. Chou, Q. Wang, Anomaly detection in online social networks, Social Networks 39 (0) (2014) 62 – 70. doi:http://dx.doi.org/10.1016/j.socnet.2014.05.002.
URL http://www.sciencedirect.com/science/article/pii/S0378873314000331

[9] Apache hadoop.
URL http://hadoop.apache.org/

[10] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Communications of the ACM 51 (1) (2008) 107–113.

[11] Apache hbase.
URL https://hbase.apache.org/

[12] Apache mahout.
URL https://mahout.apache.org/

[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber, Bigtable: A distributed storage system for structured data, ACM Transactions on Computer Systems (TOCS) 26 (2) (2008) 4.

[14] Storm - distributed and fault-tolerant realtime computation.
URL http://storm.incubator.apache.org/

[15] S4.
URL http://incubator.apache.org/s4

[16] Apache spark.
URL http://spark.apache.org/,

[17] Apache spark.
URL http://spark.apache.org/streaming/

[18] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, I. Stoica, Discretized streams: A fault-tolerant model for scalable stream processing, Tech. rep. (2012).

[19] B. Balasingam, M. Sankavaram, K. Choi, D. Ayala, D. Sidoti, K. Pattipati, P. Willett, C. Lintz, G. Commeau, F. Dorigo, J. Fahrny, Online anomaly detection in big data, in: Information Fusion (FUSION), 2014 17th International Conference on, 2014, pp. 1–8.

[20] J. Camacho, G. Macia-Fernandez, J. Diaz-Verdejo, P. Garcia-Teodoro, Tackling the big data 4 vs for anomaly detection, in: Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on, 2014, pp. 500–505. doi:10.1109/INFCOMW.2014.6849282.

[21] Apache kafka.
URL http://kafka.apache.org/

[22] M. Solaimani, M. Iftekhar, L. Khan, B. Thuraisingham, Statistical technique for online anomaly detection using spark over heterogeneous data from multi-source vmware performance data, in: Proceedings of the 2014 IEEE International Conference on Big Data (IEEE BigData 2014), IEEE, 2014.

[23] C. Wang, K. Viswanathan, L. Choudur, V. Talwar, W. Satterfield, K. Schwan, Statistical techniques for online anomaly detection in data centers, in: Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on, 2011, pp. 385–392. doi:10.1109/INM.2011.5990537.

[24] VMware, Automating the virtual datacenter.
URL https://www.vmware.com/files/pdf/avd_wp.pdf

[25] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing.

[26] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. Mccauley, M. Franklin, S. Shenker, I. Stoica, Fast and interactive analytics over hadoop data with spark.

[27] N. Jaques, Fast johnson-lindenstrauss transform for classification of high-dimensional data.

[28] T. Hoskin, Parametric and nonparametric: Demystifying the terms.

[29] R, The r project for statistical computing.
URL http://www.r-project.org/

[30] Why use resource pools?
URL http://pubs.vmware.com/vsphere-4-esx-vcenter/index.jsp?topic=/com.vmware.vsphere.resourcemanagement.doc_40/managing_resource_pools/c_why_use_resource_pools.html

[31] VMware, vsphere esx and esxi info center.
URL http://www.vmware.com/products/esxi-and-esx/overview

[32] Apache hadoop nextgen mapreduce (yarn).
URL http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[33] top, Top command in linux.
URL http://linux.about.com/od/commands/l/blcmdl1_top.htm

[34] vmstat, vmstat.
URL http://linuxcommand.org/man_pages/vmstat8.html

[35] VMware, Vmware vsphere guest sdk documentation.
URL https://www.vmware.com/support/developer/guest-sdk/

[36] Rabbitmq tm.
URL https://www.rabbitmq.com/

[37] N. Cressie, T. R. Read, Multinomial goodness-of-fit tests, Journal of the Royal Statistical Society. Series B (Methodological) (1984) 440–464.

[38] L. Kaufman, P. Rousseeuw, Finding groups in data: an introduction to cluster analysis, Wiley series in probability and mathematical statistics. Applied probability and statistics, Wiley, 2005.
URL http://books.google.com/books?id=yS0nAQAAIAAJ

[39] I. Assent, P. Kranen, C. Baldauf, T. Seidl, Anyout: Anytime outlier detection on streaming data, in: Database Systems for Advanced Applications, Springer, 2012, pp. 228–242.

[40] L. Yu, Z. Lan, A scalable, non-parametric anomaly detection framework for hadoop, in: Proceedings of the, ACM, 2013, p. 22.

[41] M. Gupta, A. B. Sharma, H. Chen, G. Jiang, Context-aware time series anomaly detection for complex systems.

[42] E. Frey, Bashreduce (2009).
URL http://rcrowley.org/2009/06/27/bashreduce

[43] R. Yu, X. He, Y. Liu, Glad: group anomaly detection in social media analysis, in: Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2014, pp. 372–381.

[44] L. Xiong, B. Póczos, J. G. Schneider, Group anomaly detection using flexible genre models, in: Advances in Neural Information Processing Systems, 2011, pp. 1071–1079.

[45] M. Steyvers, T. Griffiths, Probabilistic topic models, Handbook of latent semantic analysis 427 (7) 424–440.

[46] Mllib - basic statistics.

URL https://spark.apache.org/docs/latest/mllib-statistics.html