SAND2014-18521C

Spark-based Anomaly Detection Over Multi-source VMware Performance Data In Real-time

Mohiuddin Solaimani, Latifur Khan, Bhavani Thuraisingham
Department of Computer Science
The University of Texas at Dallas
Richardson, TX
{mxs121731, lkhan, bhavani.thuraisingham}@utdallas.edu

***Abstract*—Anomaly detection refers to identifying the patterns in data that deviate from expected behavior. These nonconforming patterns are often called outliers, malware, anomalies or exceptions in different application domains. This paper presents a novel, generic real-time distributed anomaly detection framework for multi-source stream data. As a case study, we have decided to detect anomalies for a multi-source VMware-based cloud data center. The framework monitors streaming VMware performance data (e.g., CPU load, memory usage, etc.) continuously. It collects data simultaneously from all the VMware instances connected to the network and notifies the resource manager to dynamically reschedule its resources when it identifies any abnormal behavior from its collected data. We have utilized Apache Spark, a distributed framework for processing stream data and making predictions without any delay. Spark is chosen over a traditional distributed framework (e.g., Hadoop and MapReduce, Mahout, etc.) that is not ideal for stream data processing. We have implemented a flat incremental clustering algorithm to model the benign characteristics in our distributed Spark-based framework. We have compared the average processing latency of a tuple during clustering and prediction in Spark with Storm, another distributed framework for stream data processing. We experimentally find that Spark processes a tuple much quicker than Storm on average.**

***Index Terms*—Anomaly detection; Real-time anomaly detection; Incremental clustering; Resource scheduling; Data center**

I. Introduction

Anomalies deviate from the normal behavior of a system. Typically, the anomalous items indicate warnings or exceptions such as system malfunctions, system overloads, malicious attacks in a network, structural defects or errors in a program. Sometimes they are called intrusions, malware or outliers, etc. An online system generates data continuously. The volume of data is so large that it is not possible to monitor the behavior of the data with traditional means. Moreover, the data may be complex in nature and have several attributes to monitor. That is why a real-time anomaly detection [1], [2] system is so crucial. It monitors the data and automatically takes the corrective actions when an anomaly is detected, thus preventing it from causing any damage to the system.

A data center builds a private cloud-based infrastructure as a service (IaaS) [3], [4]. These centers provide a secure, on-demand ability for end users to deploy their workloads. Resource management for data centers requires examining user requirements and available resources to fulfill its customer demands. As the user demands may vary, the system must scale its infrastructure resources dynamically. To accomplish this, a data center must have a real-time monitoring system. The system monitors the resource's performance data continuously and detects abnormal behavior. It then reschedules its resources to keep the system in balance. For example, a data center allocates a fixed amount of CPU and memory for a number of users. After a certain time, if the users run more computationally expensive programs, the CPU or memory usage will be high and require more resources to do their job. The real-time monitoring system of the data center detects these abnormal resource statistics and allocates more resources dynamically.

Online data processing is a time-sensitive operation. It must convert continuous high volume, high velocity data into real-time, actionable information. Furthermore, these "Big Data" volumes should be processed with ease, and delivered with low latency, even when data velocity is high. To build such a system, a scalable real-time operational capability is needed. Recently, several Big Data frameworks have been introduced (e.g., Hadoop [5], MapReduce [6], HBase [7], Mahout [8], Google Bigtable [9], etc.) that address the scalability issue. However, they excel at batch-based processing. Apache Storm [10] and Apache S4 [11] are distributed frameworks that process stream data. Apache Spark [12], [13] is another distributed framework that offers streaming integration with time-based in-memory analytics for live, real-time data. Spark runs a streaming computation as a series of micro-batch jobs, using a batch size as low as possible to achieve low latency. It keeps the states between batches in memory so that it can recover quickly. Spark has an advanced DAG (Direct Acyclic Graph) execution engine that supports cyclic data flow and in-memory computing. These features make Spark faster than other distributed frameworks (100$\times$ faster than Hadoop MapReduce in memory, or 10$\times$ faster on disk) [12]. It is also faster than Storm and S4 [14]. Overall, it generates low latency, real-time results. It has instant productivity and no hidden obstacles. It also has an easy cluster setup procedure.

In this paper, we make the following contributions:

- We have developed a novel generic real-time framework for multi-source stream data using Apache Spark [12], [13] and Kafka [15] (a message broker). Kafka provides guaranteed message delivery with proper ordering. This means messages sent by a producer to a particular topic partition will be delivered in the order they are sent, and a consumer will see messages in the order they are stored. Moreover, Kafka can form a cluster to handle a high volume of data with low processing latency and server failures without losing messages. As a case study, we attempt to detect anomalies for a VMware-based data center. Our real-time framework is generic so that it can handle continuous performance data (CPU load, memory usage, disk storage information, etc.) from multiple VMware instances simultaneously. This means that all VMwares can send their performance data to our framework, which collects them as a data "chunk". Kafka ships this continuous multi-source stream data from all VMwares to a Spark cluster. Inside Spark, machine learning techniques are applied to analyze the data.

- A flat incremental clustering technique [16] has been used to model online benign data. The model is built on benign training data only, and the number of clusters are not fixed before building the model, rather they are generated dynamically.
- We experimentally show that our real-time framework detects anomalies without sacrificing accuracy. It also has a low latency to analyze data. We have reduced the prediction time subsequently so that the overall data processing time is reduced.
- We have also implemented our framework using Apache Storm [10]. We have compared the average processing latency of a tuple during training and testing for both a Spark and Storm-based implementation. We found that Spark outperforms Storm substantially.

The rest of the paper is organized as follows: Section II describes key concepts used in the paper. Section III describes details about our anomaly detection framework. Section IV shows more technical details of our novel Spark-based anomaly detection framework. Section V shows the experimental results using our framework. Section VI covers the related works. Finally, Section VII states the conclusion and future work.

II. BACKGROUND

The following topics are the basic building blocks of our real-time anomaly detection framework.

A. Data Center

A data center is the store house of data. It provides computer systems and associated components, such as telecommunications and storage systems. A data center has a cloud storage appliance for its storage infrastructure. It also has resource management software to manage its storage and infrastructure with VMware, OpenStack and Microsoft. Some advanced data centers have a dynamic and compatible operating environment allowing users to leverage internal and external resources to mitigate their extra demands.

In this paper, we design a real-time framework to detect anomalies on a VMware-based data center.

B. Dynamic Resource Scheduling

VMware Dynamic Resource Scheduling (DRS) [17] allows users to define the rules and policies that decide how virtual machines share resources and how these resources are prioritized among multiple virtual machines. Some other rules like affinity and anti-affinity rules [18] have also been defined to improve the scheduling. All the rules are static and cannot be changed dynamically. In order to do real-time scheduling, we need machine learning techniques. In the core of the scheduler, unsupervised (no training data), and supervised (training data) learning can be used to analyze stream data.

For unsupervised learning, data instances associated with normal load must first be gathered. Then, a clustering algorithm (e.g., k-means [19], incremental clustering, etc.) is applied to these instances. Finally, to classify new, unseen instances (e.g., a "normal" VM or a "resource intensive" VM), an outlier detection approach can be applied. For example, for each test instance, if it is inside any cluster, then it can be treated as a VM "normal" load. Otherwise, it is classified as a "resource intensive" VM. An instance is considered inside a cluster if the distance to its centroid is smaller than its radius.

For supervised learning, first, training data is collected for classes of interest namely, "normal" and "resource intensive" categories. Next, a classification model (or rules) is built from these training data using a supervised learning algorithm (e.g., support vector machines, decision trees, k-nearest neighbor). Finally, testing (unseen) instances are predicted using the classification model.

C. VMware Performance Stream Data

The performance metrics of VMware reflect its resources usage. If a user runs a resource intensive application (e.g., CPU or memory hungry), the respective counters in the performance metrics rise beyond their threshold. As a result, the application requires more resources to complete its task smoothly. Our real-time distributed framework should diagnose this event and reschedule the appropriate resources dynamically.

VMware has some administrative commands like [TOP], and [ESXTOP], which provide performance data of the virtual machines (VMs). Periodically, we capture these statistics as a snapshot. After preprocessing and feature selection, a feature vector is generated that acts as a data point for the proposed data analytics algorithm.

We have used [ESXTOP] for collecting VMware statistics. It gives several performance statistics for CPU, Memory, and Disk storage. Our performance metrics contain average CPU load, percentage CPU usage of the physical cores (%Processor Time), percentage utilization of the logical cores (%Util Time), and percentage utilization of the physical cores (%Core Util Time).

D. Apache Spark and Real-time Issues

Data center automation [20] (e.g., dynamic resource management) may require analyzing the performance data in real-time to identify anomalies. As the stream data comes continuously and is voluminous, we require a scalable distributed framework. To address the scalability issue, we can use a distributed solution like Hadoop, MapReduce, etc. Hadoop runs in 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 [12], which is fault-tolerant, and supports distributed real-time computation system for processing fast, large streams of data.

Apache Spark [12] is an open-source distributed framework for data analytics with a simple architecture. It uses Hadoop [5] for the distributed file system and can work on the top of YARN, a next generation Hadoop cluster [21]. Spark avoids the I/O bottleneck of conventional two-stage MapReduce programs. It also provides in-memory cluster computing that allows a user to load data into a cluster's memory and query it efficiently. This increases the performance up to 100 times faster than Hadoop MapReduce [12].

Spark has two key concepts: Resilient Distributed Dataset (RDD) and directed acyclic graph (DAG) execution engine.

- Resilient Distributed Dataset (RDD) [22] - 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 datasets from the input or existing RDD (e.g. map or filter), and actions return a value after executing calculations on the dataset (e.g. reduce, collect, count, saveAsTextFile, etc.). Transformations are the lazy operation that define only the new RDD, while actions perform the actual computation and calculate the result or write to the external storage.
- Directed acyclic graph (DAG) execution engine - when the user runs an action on the RDD, a directed acyclic graph is generated that considers all the transformation dependencies. This eliminates the traditional MapReduce multi-stage execution model and improves performance.

Spark supports both batch and stream processing [23]. 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 from a small time interval. It then delivers these small packed chunks of data to a batch system for processing.

Spark Streaming has two types of operators:

- Transformation operator - creates a new DStream [14] from one or more parent streams. It can be either stateless (independent on each interval) or stateful (share data across intervals).
- Output operator - an action operator that 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 dataset 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).

Spark Streaming inherits all the transformations and actions operations of typical batch frameworks, including map, reduce, groupBy, and join, etc.

III. Real-time Anomaly Detection framework

A real-time anomaly detection framework consists of data preprocessing, model training, prediction, model updating, and resource scheduling.

The stream data comes in a raw format, which needs to be processed in order to be analyzed. After processing, it is converted to multi-dimensional time series data, such as the CPU usage or load of user programs in a data center over time. Sometimes preprocessing needs data normalization and noise elimination. The real-time framework applies machine learning algorithms to analyze stream data. It uses unsupervised clustering algorithms (e.g., k-means, incremental clustering) to build clusters on training data. Furthermore, the clustering applies only to benign data. After clustering, it predicts data using its cluster information. If any data fails to fall in any cluster, it is considered an anomaly; otherwise, it is considered benign. As the stream data evolve in nature, the training model should be updated periodically. Finally, the framework sends a request to the resource scheduler to update its resources if it detects any abnormal behavior.

Algorithm 1 Real-time framework

```
1: procedure IdentifyAnomaly(dataChunk, operation)
2:     if operation == MODEL_BUILD then
3:         m ← BuildModelWithClustering(dataChunk)
4:     if operation == PREDICT_DATA then
5:         hasAnomaly ← PredictData(dataChunk, m)
6:         NotifyManagerForScheduning(hasAnomaly)
7:     if operation == MODEL_UPDATE then
8:         m ← UPDATEModelWithClustering(dataChunk)
```

Algorithm 1 describes the basic flow of the real-time framework. Initially we build the training model from the incoming stream data (line 2-3). After building the model, it predicts (line 4-5) data using that model. Here, it is important to periodically update the training model (line 7-8) because of the dynamic nature of data. For example, after building the training model, some new applications might be executed in the VMs which are not anomalous, but still change the CPU metrics. These altered CPU metrics may not fit into any clusters in the existing model, causing them to be treated as anomalous during prediction. Therefore, the model need to be updated periodically to accommodate these changes. Finally, after prediction, if the framework finds any deviation from the usual behavior, it notifies (line 6) the resource scheduler for rescheduling.

A. Incremental Clustering

Formally, the clustering problem is defined as follows: we are given N points and we have to group these N points into k clusters so that points of similar behavior group together. A similarity or distance measure (like cosine similarity, Euclidean distance, etc.) is used to assign the points to each cluster. Each cluster has some attributes like centroid, maximum distance or radius, etc. The centroid is the center point of each cluster. The radius is the maximum-distanced point to its centroid. In this paper, we will use these properties to detect anomalies.

Incremental clustering [16] works on continuous stream data. It processes each data point sequentially and incrementally assigns them to their respective clusters. The following algorithm, described in [16], shows incremental clustering that is generated from N data points while maintaining a collection of k clusters. For each data point, it is either assigned to one of the current k clusters or it starts a new cluster. If the number of clusters exceeds k, then two existing clusters are merged into one to fix the number of clusters to k. In this paper, we have also implemented a flat incremental clustering algorithm where the cluster size may vary according to the input data.

Algorithm 2 Incremental clustering

```
1:  procedure CLUSTERING
2:      $k \leftarrow numOfcluster$
3:      $totalCluster \leftarrow 0$
4:      for each data point $n$ do
          $isFit \leftarrow FitsInAnyCluster(n)$
5:          if $isFit == false$ then
6:              $CreateCluster()$
7:              $totalCluster \leftarrow totalCluster + 1.$
8:              if $totalCluster > k$ then
9:                  $MergeTwoNearestClusterToOne()$
10:         else
11:             $UpdateClusterInfo()$
```

For data center automation in virtual environments [20] , we envision the following framework. The virtual resource management software consists of an additional dynamic load balancing decision module. This module will be connected to our proposed Spark framework. It monitors the real-time performance data generated from virtual machines. It collects the data and splits it to DStream [23], [14]. It builds the model by applying transformation and action operations to the input DStream. After building the training model, it passes the model for prediction. In the prediction unit, again transformation and action operations have been performed to identify the anomalies in the input stream data. We will implement machine learning algorithms (described in algoritms 3, 4) here to analyze data. After analyzing, the framework will report the result to the virtual center resource manager. Finally, the resource manager will allocate the resources (network bandwidth, storage capacity) to the overused VMware machines on-the-fly.

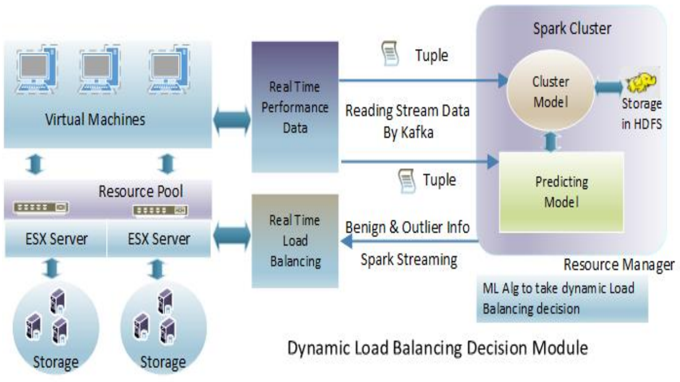

Fig. 1. Dynamic Resource Management using Apache Spark

In Fig. 1, 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 [17] to dynamically allocate resources, if necessary.

IV. SPARK-BASED ANOMALY DETECTION FRAMEWORK

Our VMware cluster consists of 5 VMware ESXi [24] (VMware hypervisor server) 5.x systems. Each of the systems has a Intel(R) Xeon(R) CPU E5-2695 v2 2.40GHz processor, 64 GB DDR3 RAM, a 4 TB hard disk and a 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 VMs are sharing the resources, performance may vary during runtime. We have installed Linux Centos v6.5 64 bit OS in each of the VMs along with Java JDK/JRE v1.7. We have designed a real-time outlier detection system on this distributed system using Apache Spark version 1.0.0. We have also installed Apache Hadoop NextGen MapReduce (YARN) [21] with Hadoop v2.2.0 to form a cluster. Apache Spark uses this YARN cluster for the distributed file system.

Our framework is divided into two components. The first one is the message broker and the second is the stream data mining module.

A. Message Broker

We are continuously monitoring the performance data from each VMware ESXi server [24]. The VMware utility tool [EXSTOP] is utilized to continuously write the performance data of its server to a CSV (Comma Separated File) file. Those CSV files are read and the data is sent to the message broker. Several message brokers (e.g., Apache Kafka [15], RabbitMQ [25], 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 Fig. 2) the performance data to Spark's streaming framework. It provides guaranteed message delivery with proper ordering and a Kafka cluster can be formed to handle large volumes of data.

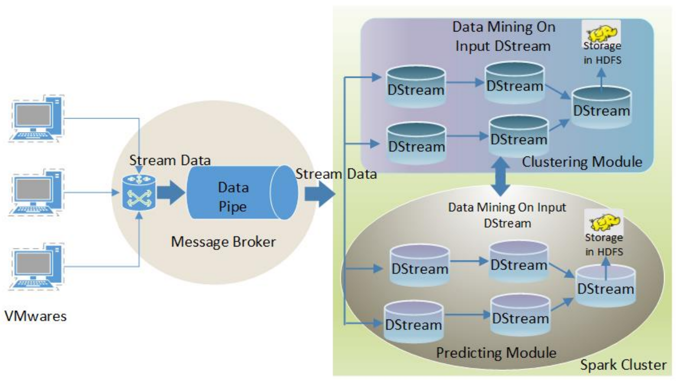

Fig. 2. Technical Approach

B. Stream Data Mining Module

As in Fig. 2, we have implemented a clustered network using Apache Spark. The VMware performance data is transported through a Kafka queue continuously as a stream. It is split into small micro-batches (DStreams). Several transformation and action operations are performed on these DStreams to build the training model and predict anomalies. At each micro-batch processing step, the generated immutable dataset is stored in-memory and the results are propagated to the next micro-batch processing module. The framework builds the model from the benign stream. After that, it predicts the incoming stream data to identify anomalies.

Algorithm 3 Training

```
1: procedure TRAINING($InputDStream$)
2:     $trainingModel \leftarrow InputDStream.map(Tuple\ t)\{$
3:     $n \leftarrow t.getDataPoint()$
4:     $closestCluster \leftarrow$ FINDNEARESTCLUSTER($n$)
5:     if $closestCluster == Empty$ then
6:         return $Map(n, n)$
7:     else
8:         return $Map(closestCluster.Centroid, n)$
9:     $\}.reducedByKey(centroid, [dPoint1, dPoint2..])\{$
10:    $index \leftarrow$ FITSINANYCLUSTER($centroid$)
11:    if $index == -1$ then
12:        CREATECLUSTER($centroid$)
13:    else
14:        UPDATE($index, [dPoint1, dPoint2..]$)
15:    }
16: procedure UPDATE($index, [dPoint1, dPoint2..]$)
17:     $cluster \leftarrow$ FINDCLUSTER($index$)
18:     CALWGTAVGCENTER($cluster, [dPoint1, dPoint2..]$)
19:     CALNEWRADIUS($cluster$)
```

(i) **Training**

Algorithm 4 Testing

```
1: procedure TESTING(InputDStream)
2:     $testingModel \leftarrow InputDStream.map(Tuple\ t)\{$
3:     $n \leftarrow t.getDataPoint()$
4:     for each cluster $k$ do
5:         $distance \leftarrow EuclideanDistance(n, centroid(k))$
6:         if $distance \geq Radius(k)$ then
7:             $n$ is anomaly
8:         else
9:             $n$ is benign
10:    }
11:    $testingModel.print()$
```

The Spark framework builds the initial training model. Algorithm 3 shows that it has two components: map and reducedByKey. The map function (lines 3-8) takes each instance of the DStream (called a tuple) and extracts data points from this tuple. Later, it finds the nearest cluster for that data point. If there is no nearest cluster, then it uses the data point as a new cluster's centroid. It uses this nearest cluster's centroid (line 8) or new centroid (line 6) as a key and the data point as a value. The reducedByKey acts as a reducer. It receives the centroid as a key and a list of data points as a value. If the centroid does not belong to any cluster, then it creates a new cluster with that centroid. Otherwise, it updates the cluster. Lines 9-15 describes this process. All the data points with the same centroid go to the same reducer. Data points with different centroids will go to different reducers. In the UPDATE function (line 16-19), the cluster is fetched from its index. After that, the cluster's centroid is updated by taking a weighted average of the list of data instances.
As Spark has a stateful property, all the clusters generated in training should be kept in memory. Furthermore, it does not need to explicitly store the data points, only cluster information. Here, a flat incremental clustering algorithm is used for training the model, and the number of clusters is not fixed. For each data point, if it fits to any cluster, then the cluster is updated. Otherwise, a new cluster is created with that point as the centroid. So, the number of clusters is not fixed and may vary according to the training data. Moreover, the Euclidean distance is used to measure the distance of two data points, which can be converted to a similarity measure via the following:

$$Similarity = \frac{1}{1 + Distance} \tag{1}$$

The above similarity function has a lowest value of zero (when Distance = ∞) and a highest value of one (when Distance = 0). It is easy to see that the more distant points would have lower similarity and closer points would have higher similarity. A threshold of 70% is used when assigning points to a cluster. Furthermore, a weighted average is used when calculating the centroid of the cluster.

(ii) **Testing**
During testing, each data point is extracted from the DStream and checked against the model. If any data point does not fit in any cluster, it is considered anomalous, otherwise it is benign. If the distance between the data point and the centroid of the clusters is less than the cluster's radius, then it is considered as belonging to that cluster. A detailed algorithm is given from line 1-11 in the algorithm 4.

C. Update Training Model

The stream data is dynamic in nature. Therefore, to maintain accuracy, it is necessary to update the prediction model. At a fixed interval, the predictor signals the clustering model to update the model. When the clustering model receives this signal, it uses the incoming tuples to modify its clustered model. It updates the cluster's centroids or it may add new clusters. We also save the whole cluster model information to HDFS for backup.

D. Generating Anomalous Data

Generating anomalous behavior is very tedious. As we are using CPU performance data, we have to find when the VMware instances become more CPU hungry. By conducting several experiments, we have found that they become CPU hungry when CPU metrics in a VMware increases. We have pragmatically increased CPU load to simulate anomalous behavior in the data. For example, a program with an infinite loop may increase the CPU load to 100%. Again we have also done some expensive database read/write operations which also increases the value of several counters in CPU metrics (e.g., CPU usage, processor time, core utility time, etc.).

E. Design Challenge

We faced some design challenges like gathering domain knowledge of VMware performance data, integrating a stable message broker into the framework, and handling message synchronization and message ordering, etc.

F. Scalability

Our framework is more generic. We can build a message broker cluster using Kafka [15]. It is highly available and scalable. It helps us to add more VMware sources to collect their performance data. Moreover, we can add more threads to Spark executors/workers to increase parallelism. Thus we can accommodate these large volumes of data.

V. Experimental Results

A. Dataset

We have used two different datasets for our experiments. For the first dataset D1, we run several jobs on a Spark cluster in several VMware instances. We then monitor the real-time CPU performance metrics from all the VMware instances. We capture the stream and build the benign model. We then programmatically increase the CPU metrics to generate anomalous stream data.

For the second dataset D2, we utilize a benchmark analysis framework called Yahoo Cloud Serving Benchmark (YCSB) [26]. We run the benchmark with different workloads and runtime properties continuously to capture stream data and build our benign model. The database system is a simple MySQL database with billions of rows inserted. Data is loaded in the load phase of the YCSB run. Later, transactions such as read and update are run on the database system to increase the workload. The client data is the performance data that captures the network, I/O, CPU and memory usages during the workload run. We have used millions of read/write operations on the database to generate anomalous stream data.

B. Result

In this section, we present the accuracy of our real-time framework. TABLE I shows the Apache Spark cluster setup.

TABLE I
Spark Cluster

Component	Number of parallelism
Worker for emitting tuples	05
Worker for clustering	08
Worker for prediction	08

As data arrive continuously, a time window is used to collect the training instances for building the model. The flat incremental clustering algorithm is run on 10,000 data instances. At the end, there are 63 clusters for dataset 1 (D1) and 134 for dataset 2 (D2). These clusters are generated dynamically. TABLE II shows the training data model.

TABLE II
Training

	D1	D2
Number of data points	10,000	10,000
Number of clusters	63	134

TABLE III
Testing

Dataset	TPR	FNR	TNR	FPR
D1	98.0%	2.00%	99.83%	0.17%
D2	99.20%	0.80%	99.06%	0.94%

TABLE III shows the accuracy of the framework. A total of 3,500 held-out testing instances for both datasets are used for evaluation (3,000 benign and 500 anomalous). Our framework correctly predicts 2,995 benign data out of 3,000 and also identifies 490 anomalous data out of 500 for dataset 1 (D1). It also correctly predicts 2,972 benign data out of 3,000 and also identifies 496 anomalous data out of 500 for dataset 2 (D2). TABLE III reflects the overall accuracy statistics. So our framework has higher accuracy to identify anomalies. In this table, the true positive rate (TPR) is the proportion of actual anomalies which are correctly identified and the true negative rate (TNR) is the proportion of actual benign instances which are correctly identified. Moreover, the false negative rate (FNR) is the proportion of actual anomalies which are misclassified as benign and the false positive rate (FPR) is the proportion of actual benign instances which are misclassified as anomalous.

C. Comparison with Storm

We have also implemented this framework using Storm with the same environment and the same data.

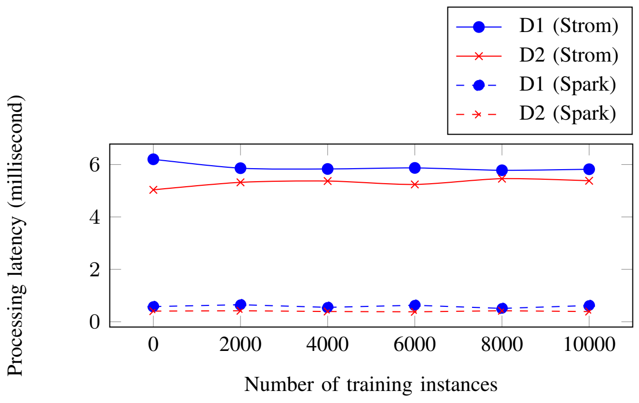

Fig. 3. Average Tuple Processing Latency During Clustering

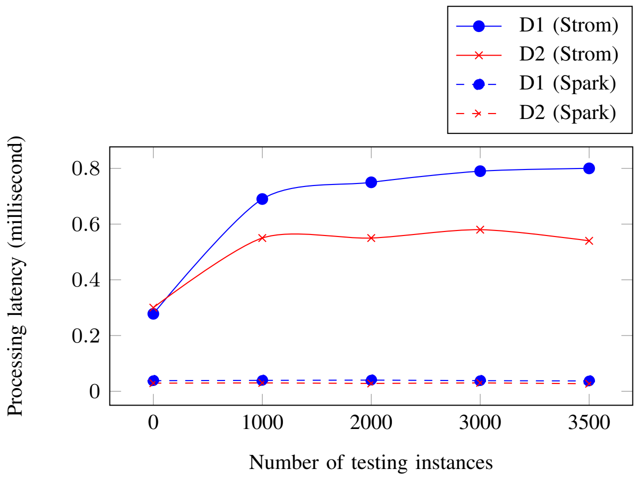

Fig. 4. Average Tuple Processing Latency During Prediction

Fig. 3 shows the average process latency of a tuple for clustering during training for both the Spark and Storm implementations. We have taken the average process latency of a tuple after a certain number of training instances have been processed and plotted them. The graph shows that the clustering takes almost 5.85 ms (dataset 1) and 5.30 ms (dataset 2) on average for Storm and almost 0.6 ms (dataset 1) and 0.4 ms (dataset 2) on average for Spark. From the result we see that for dataset 1 Spark is almost 10 times faster than Storm and for dataset 2, Spark is almost 13 times faster than Storm.

Fig. 4 shows the average process latency of a tuple for predicting anomalies during testing for both the Spark and Storm implementations. We have taken the average process latency of a tuple after a certain number of training instances have been processed and plotted them. The graph shows that the prediction takes almost 0.78 ms (dataset 1) and 0.55 ms (dataset 2) on average for Storm (when the graph reaches saturation) and almost 0.038 ms (dataset 1) and 0.03 ms (dataset 2) on average for Spark. From the result we see that for dataset 1 Spark is almost 20 times faster than Storm and for dataset 2, Spark is almost 18 times faster than Storm.

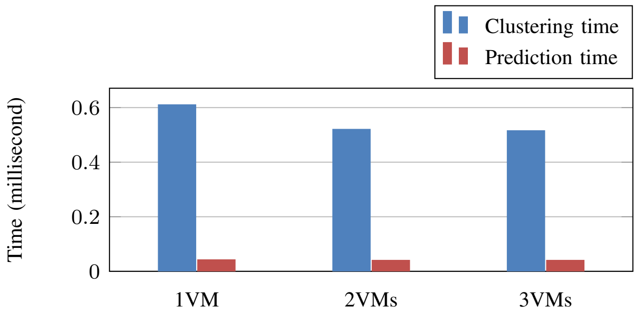

Fig. 5. Average Tuple Processing Latency for Different Input VM Instances

D. Experiment of Parallelism

We vary the number of input VMware instances when collecting the data. From Fig. 5, we see that the average process latency of a tuple during clustering does not increase when the number of input VMware instances increases. For a single VM, the latency is almost 0.61 ms and then it decreases to almost 0.52 ms for double and triple VMs. The average process latency during prediction is almost 0.04 ms for all three cases.

The limitation of this framework is using the appropriate threshold. We have experimentally selected the similarity threshold at 70% during clustering. Finally, as we do not fix the number of clusters, it might be possibile to generate many clusters consisting of single data points. However, these clusters would not be influential during testing because they do not have a radius. Unless an exact match to these data points occurs, no testing data points will fall in these clusters. Additionally, these single point clusters can be easily removed.

VI. 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 generally produced better accuracy, but it also has a greater time complexity for very large data sets. K-means also has an initial centroid problem. K-medoids [27] can overcome this problem. It initially selects k centers but the centers are repeatedly changed randomly and thus it improves the sum of the squared error. Assent et al. [28] have proposed an AnyOut algorithm for detecting outliers on stream data. They use hierarchical clustering and determine an outlier score based on the deviation between the object and the cluster. They can smoothly detect outliers but they are not addressing the scalability issue. As hierarchical clustering has a complex data structure, it is time consuming when deployed in a distributed system.

Yu et al. [29] have proposed a non-parametric cluster-based anomaly detection framework for distributed systems. It uses Hadoop[5] and MapReduce. It outperforms existing static anomaly detection techniques. Gupta et al. [30] 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 context-based cluster and generate an anomaly score. Apache Mahout [8] is a machine learning tool that uses Hadoop and MapReduce. It has k-means and StreamingKMeans implementations. These implementations run in batch-mode and are time consuming. Therefore, these implementations are not ideal for clustering real-time data.

All of the above distributed techniques use Hadoop and MapReduce. Hadoop works well in batch-mode where we have the data in advance. It does not work well with real-time systems where data comes continuously. If Hadoop is used for building the training model, then for each new instance it must rebuild the model using *all* data instances, which is a waste of time and resources. So, we cannot use Hadoop with streaming or real-time data. Similar distributed systems like HBase [7], BashReduce [31], etc. also have the same problem. Furthermore, although Apache Storm [10] and Apache S4 [11] are also available for stream data processing, Spark is much faster than these [23]. So, we have decided to use Spark [10]. It is scalable and fault tolerant. It has guaranteed data processing and finally it is quite faster than other frameworks like Hadoop, Storm, etc.

No work has been done before on real-time anomaly detection for VMware performance data using a distributed framework. Our real-time anomaly detection framework using Spark is the first development. There has been a lot of work on anomaly detection, which has mainly been focused on accuracy in batch data sources while ignoring potential performance issues (speed). Moreover, they have not considered distributed frameworks with real-time data. Furthermore, Spark has a StreamingKMeans implementation. It generates initial centroids randomly. In a real-time framework, these randomly generated centroids may lead to a corrupted training model that may not correctly identify anomalous data. Moreover, if in order to randomly generate the centroids, domain knowledge of the data is necessary. In our real-time framework, we have no idea about the VMware's performance data. We know only the state of the VMwares ("normal" or "resource intensive") when we build the training model. So, we prefer to use our flat incremental clustering instead of Spark's StreamingKMeans.

VII. Conclusion and Future work

In this paper we have implemented a real-time anomaly detection framework using Apache Spark. It has the ability to manage streaming data and guarantees data delivery. It is also fault tolerance and is flexible to increasing parallelism.

We envision our real-time system to be the key part of a dynamic resource management system for VMware. Our framework is generic and can be integrated into other systems such as monitoring system logs to measure a system's operational performance, analyzing sensor data from the embedded devices, etc.

In the future, we will implement other machine learning algorithms in our framework. Moreover, we will also consider other performance metrics (e.g., memory and storage statistics, etc.) and try to correlate them for detecting anomalies.

Acknowledgment

Funding for this work was partially supported by the Laboratory Directed Research and Development program at Sandia National Laboratories.

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.

References

[1] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys (CSUR)*, vol. 41, no. 3, p. 15, 2009.
[2] A. Mustafa, M. Solaimani, L. Khan, K. Chiang, and 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*, 2014.
[3] Interoute. Iaas. [Online]. Available: http://www.interoute.com/what-iaas
[4] Oracle. Making infrastructure-as-a-service in the enterprise a reality. [Online]. Available: http://www.oracle.com/us/products/enterprise-manager/infrastructure-as-a-service-wp-1575856.pdf
[5] Apache hadoop. [Online]. Available: http://hadoop.apache.org/
[6] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[7] Apache hbase. [Online]. Available: https://hbase.apache.org/
[8] Apache mahout. [Online]. Available: https://mahout.apache.org/
[9] 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 Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.
[10] Storm - distributed and fault-tolerant realtime computation. [Online]. Available: http://storm.incubator.apache.org/
[11] S4. [Online]. Available: http://incubator.apache.org/s4
[12] Apache spark. [Online]. Available: http://spark.apache.org/,
[13] Apache spark. [Online]. Available: http://spark.apache.org/streaming/
[14] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: A fault-tolerant model for scalable stream processing," Tech. Rep., 2012.
[15] Apache kafka. [Online]. Available: http://kafka.apache.org/
[16] M. Charikar, C. Chekuri, T. Feder, and R. Motwani, "Incremental clustering and dynamic information retrieval," *SIAM Journal on Computing*, vol. 33, no. 6, pp. 1417–1440, 2004.
[17] Why use resource pools? [Online]. Available: 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
[18] VMware. Using drs affinity rules. [Online]. Available: http://pubs.vmware.com/vsphere-51/index.jsp
[19] K. Tan, Steinbach, *Introduction to Data Mining*, 2005, no. 3.
[20] VMware. Automating the virtual datacenter. [Online]. Available: https://www.vmware.com/files/pdf/avd_wp.pdf
[21] Apache hadoop nextgen mapreduce (yarn). [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

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

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

[24] VMware. vsphere esx and esxi info center. [Online]. Available: http://www.vmware.com/products/esxi-and-esx/overview

[25] Rabbitmq tm. [Online]. Available: https://www.rabbitmq.com/

[26] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

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

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

[29] L. Yu and Z. Lan, "A scalable, non-parametric anomaly detection framework for hadoop," in *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*. ACM, 2013, p. 22.

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

[31] E. Frey. (2009) Bashreduce. [Online]. Available: http://rcrowley.org/2009/06/27/bashreduce