SOFTWARE: PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2016)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2390

Online anomaly detection for multi-source VMware using a distributed streaming framework

Mohiuddin Solaimani[1,*,†], Mohammed Iftekhar[1], Latifur Khan[1], Bhavani Thuraisingham[1], Joe Ingram[2] and Sadi Evren Seker[3]

[1]*Department of Computer Science, The University of Texas at Dallas, Dallas, TX 75080, USA*
[2]*Sandia National Laboratories, Albuquerque, NM 87123, USA*
[3]*Department of Business, Istanbul Medeniyet University, Istanbul, Turkey*

Anomaly detection refers to the identification of patterns in a dataset that do not conform to expected patterns. Such non-conformant patterns typically correspond to samples of interest and are assigned to different labels in different domains, such as outliers, anomalies, exceptions, and malware. A daunting challenge is to detect anomalies in rapid voluminous streams of data.

This paper presents a novel, generic real-time distributed anomaly detection framework for multi-source stream data. As a case study, we investigate anomaly detection for a multi-source VMware-based cloud data center, which maintains a large number of virtual machines (VMs). This framework continuously monitors VMware performance stream data related to CPU statistics (e.g., load and usage). It collects data simultaneously from all of the VMs connected to the network and notifies the resource manager to reschedule its CPU resources dynamically when it identifies any abnormal behavior from its collected data. A semi-supervised clustering technique is used to build a model from benign training data only. During testing, if a data instance deviates significantly from the model, then it is flagged as an anomaly.

Effective anomaly detection in this case demands a distributed framework with high throughput and low latency. Distributed streaming frameworks like Apache Storm, Apache Spark, S4, and others are designed for a lower data processing time and a higher throughput than standard centralized frameworks. We have experimentally compared the average processing latency of a tuple during clustering and prediction in both Spark and Storm and demonstrated that Spark processes a tuple much quicker than storm on average.

Received 4 May 2015; Revised 30 November 2015; Accepted 10 December 2015

KEY WORDS: real-time anomaly detection; incremental clustering; resource scheduling; data center; Apache Spark; Apache Storm

1. INTRODUCTION

Real-time anomaly detection [1–6] aims to capture abnormalities in system behavior in real time. These abnormalities or anomalies may appear in the form of malicious network intrusions [5, 7–9], malware infections, abnormal interaction patterns of individuals/groups in social media [10], over-utilized system resources due to design defects, and so on. Anomaly detection in this regard involves absorbing continuous streams of data, performing analysis on them, and initiating corrective measures when necessary. Therefore, it is a challenging task given the volume, velocity, and possibly complex nature of the input data. Failure to accomplish this task in a timely manner may lead to catastrophic consequences with a severe impact on business continuity.

*Correspondence to: Mohiuddin Solaimani, Department of Computer Science, The University of Texas at Dallas, Dallas, TX 75080, USA.
†E-mail: mxs121731@utdallas.edu

The most rigorous form of real-time anomaly detection is observed in enterprise data centers which host tens of thousands of virtual machines (VMs). Dynamic resource scheduling is a crucial differentiator for the operational efficiency of these data centers. In response to varying demands for various resources, for example, CPU and memory, the scheduler must allocate or reallocate resources dynamically. This necessitates real-time monitoring of resource utilization and performance data for the purpose of detecting abnormal behavior.

Real-time anomaly detectors for data centers must be built upon scalable frameworks capable of handling extremely large amounts of data (so called Big Data) with low latency. Popular big data frameworks, for example, Hadoop [11], MapReduce [12], HBase [13], Mahout [14], and Google Bigtable [15] are highly scalable but are geared more toward batch processing. Examples of frameworks with built-in stream processing capabilities are Apache Storm [16], Apache S4 [17], and Apache Spark [18, 19]. Spark performs in-memory analytics on stream data by running streaming computations as a series of micro-batch jobs. It makes the batch size as small as possible to achieve low latency. Each batch's processed data can be stored in any file system (including Hadoop distributed file system), database (including Hbase), or live dashboards. Spark also supports stateful operations which mean the operations have a dependency on previous batches of data. It keeps the states (intermediate batch result) between batches in memory so that it can recover them quickly. Spark uses an advanced directed acyclic graph execution engine that supports cyclic data flow and in-memory computing, which makes it faster than other distributed frameworks. Spark is 100 times faster than Hadoop MapReduce in memory, and ten times faster on disk [18]. Overall, it generates low-latency real-time results. It also has an easy cluster setup procedure.

Designing an online anomaly detection framework in a distributed system [20, 21] requires solving some key challenges. First, how should the data be handled within the distributed framework in order to process it efficiently and reliably? We utilize a message broker (Apache Kafka [22]) to transfer the data to the distributed framework without any loss of data. A fixed-time interval is used to capture and process data from the message broker. Second, how should machine learning algorithms be implemented in the distributed system? Streaming distributed frameworks like Apache Spark [18] and Apache Storm [16] come with a few standard machine learning algorithms. In order to use custom algorithms, they must be implemented in such a way that fully utilizes the benefits of a distributed framework. For our case study, we have implemented an adaptive incremental clustering algorithm.

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

First, we have developed a novel generic real-time framework for multi-source stream data using Apache Spark [18, 19] and Kafka [22]. Kafka is a message broker [23]. A message broker is an intermediate program which communicates messages or data between sources and destinations. It can route messages to one or more destinations, and similarly, it can also collect messages from multiple sources. It performs message aggregation, decomposing messages into multiple messages and sending them to their destination, then recomposing the responses into one message to return to the user. It uses topic-based message routing like publish–subscribe pattern [24]. Kafka is well compatible with Spark. It provides guaranteed message delivery with proper ordering. This means that messages sent by a producer to a particular topic partition will be delivered in the order they are sent, and a consumer also sees messages in the order they are stored. Moreover, a Kafka cluster can be formed to lower processing latency and provide fault tolerance without loss of data. As a case study, we consider anomaly detection for a VMware-based data center. Our real-time framework monitors continuous performance data (CPU load, memory usage, disk storage information, and so on) from multiple VMs periodically. We have built a model from benign training data using a semi-supervised clustering technique. During testing, if a data instance deviates significantly from the model, then, it is identified as an anomaly and notifies the resource manager to reschedule its CPU resources dynamically.

Second, we have also developed a novel real-time framework using Apache Storm [16] and Kafka. Kafka ships the continuous performance stream data from all VMs to Storm. Inside Storm, a machine learning technique is applied to analyze the data. Our real-time framework monitors a continuous flow of VMware CPU usage statistics and analyzes it by applying a semi-supervised

DOI: 10.1002/spe

incremental clustering technique. The framework has an option to accommodate other VMware performance data like memory usage and storage statistics.

Third, a semi-supervised flat-adaptive incremental clustering technique [25] has been implemented in both frameworks to model benign data in an online manner. Clusters are derived from benign training data only. The number of clusters are not fixed in advance but are generated dynamically. It will be monotonically increased over time.

Fourth, we have also experimentally established the accuracy and low latency of our real-time frameworks for anomaly detection. The overall processing time has been reduced due to the reduction in prediction time.

Finally, we experimentally compared the average processing latency of a tuple during training and testing for both the Spark and Storm-based implementation. We found that Spark outperforms Storm substantially.

The rest of the paper is organized as follows: Section 2 describes the anomaly detection framework in detail. Section 3 shows the case study of online anomaly detection framework with its implementation. Section 4 describes the experimental results. Section 5 covers related work in the area of distributed and streaming anomaly detection. Finally, Section 6 states the conclusion and proposed future work followed by an acknowledgement and bibliography.

2. 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 come 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 also involves data normalization and noise elimination. The real-time framework applies machine learning algorithms to analyze the stream data. It uses clustering algorithms (e.g., k-means and incremental clustering) to build clusters on data considered to be benign. 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, which optimistically reduces the potential for false alarms and missed anomalies. Finally, the framework sends a request to the resource scheduler to update its resources if it detects any abnormal behavior. Figure 1 describes the basic flow of the real-time framework. Initially, the model is built from the incoming stream data.

Data center automation [26] (e.g., dynamic resource management) may require analyzing the performance data in real time to identify anomalies. As the stream data come continuously and is quite voluminous, a scalable distributed environment is required for analysis. However, many distributed solutions (e.g., Hadoop and MapReduce) run in batch mode, which makes it difficult to handle real-time data. Fortunately, there are a few distributed frameworks capable of handling data in real time. Two such distributed systems are Apache Storm and Apache Spark.

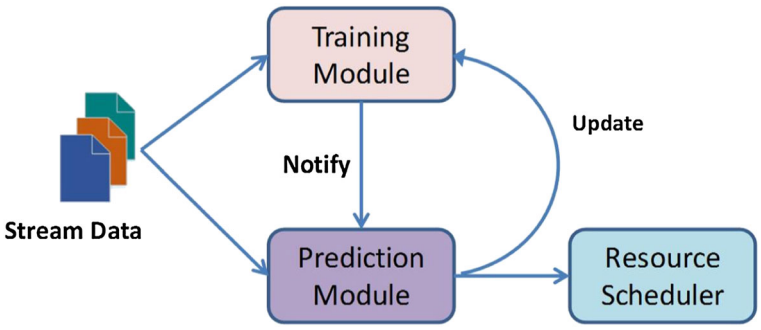

Figure 1. Real-time anomaly detection framework

DOI: 10.1002/spe

2.1. Apache Storm

Storm has a simple architecture (in Figure 2) consisting of three sets of nodes:

1. Nimbus node – a master node similar to the Hadoop JobTracker [11] . It distributes jobs and launches workers across the cluster. It also monitors jobs and reallocates workers as needed.
2. ZooKeeper node [27] communicates and coordinates the Storm cluster.
3. Supervisor node communicates with Nimbus through Zookeeper. It starts and stops workers according to Nimbus.

Storm also consists of Spout, Bolt, and Topology (in Figure 3). Spout provides an input stream into Storm and is capable of reading external data sources. Bolt is a processor unit that can process any number of input streams and produce any number of output streams. Crucial analysis logic is typically written in Bolt. Finally, Topology is a network of Spouts and Bolts. It is a complex multi-stage stream computation that runs indefinitely when it is deployed. In a Storm cluster, the master node coordinates and communicates with all the worker nodes.

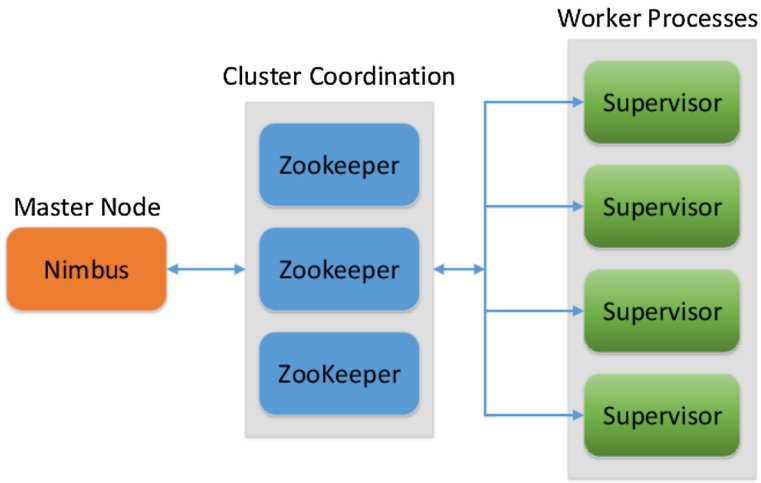

Figure 2. Storm architechture

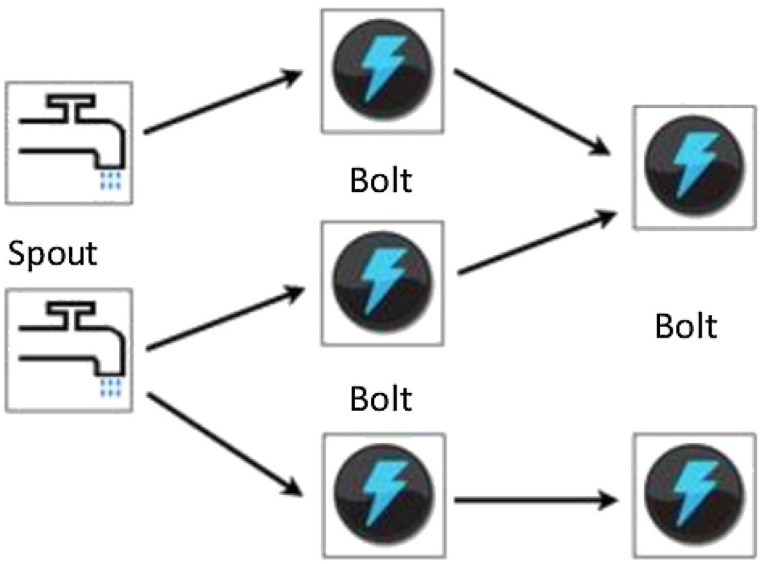

Figure 3. Storm topology

DOI: 10.1002/spe

2.2. Apache Spark

Apache Spark [18] is an open-source distributed framework for data analytics and also has simple architecture. It uses Hadoop [11] for the distributed file system and is capable of working on top of a next generation Hadoop cluster called YARN [28]. Spark avoids the I/O bottleneck of the conventional two-stage MapReduce programs by providing in-memory cluster computing that allows a user to load data into a cluster's memory and query it efficiently. This architecture increases the performance of up to 100 times over that of Hadoop MapReduce [18].

Spark has two key concepts: resilient distributed dataset (RDD) and directed acyclic graph execution engine.

1. Resilient distributed dataset [29] – 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 and map) and files stored on Hadoop distributed file system. 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, and count). Transformations are the lazy operations that only define a new RDD, while actions perform the actual computation and calculate the result or write to external storage.
2. Directed acyclic graph execution engine – whenever the user runs an action on an RDD, a directed acyclic graph is generated that considers all the transformation dependencies. This eliminates the traditional MapReduce multi-stage execution model and also improves the performance.

Spark also has a streaming implementation [30]. It is highly scalable and fault tolerant. It uses a micro-batch technique, which splits the input stream into a sequence of small chunks of data. It then processes these small chunks as a batch and sends its results to the next batch process.

Spark streaming has two types of operators:

1. Transformation operator – maps a new DStream [31] from one or more parent streams. It has both a stateless (independent on each interval) and stateful (share data across intervals) property.
2. Output operator – an action operator that allows the program to write data to external systems (e.g., save or print a DStream).

Like MapReduce, map is a transformation operation that takes each dataset element and returns a new RDD/DStream. On the other hand, reduce is an action operation that aggregates all the elements of the RDD/DStream and returns the final result (reduceByKey is an exception that returns an RDD) as a RDD/DStream. Spark streaming inherits all the transformation and action operations like map, reduce, groupBy, and join. These operators are used to build a model for prediction. In the prediction unit, transformation and action operations are also performed to identify the anomaly from the input stream data. Machine learning algorithms (described in Algorithms 4, 5) are implemented here to analyze the data. After analysis, the framework reports the results to the resource manager for the virtual center. Finally, the resource manager will allocate the resources to the overused VMware machines dynamically.

2.3. Stream data mining module

2.3.1. Clustering. It is a data mining technique that attempts to group objects so that objects in each group have similar characteristics. For example, given N data points (each with attributes or features), the objective is to group these N points into k clusters such that points with similar attributes group together. Clustering uses several proximities (like cosine similarity and Euclidean distance) to assign objects to each cluster. Each cluster also has a centroid or center and a radius.

Incremental clustering [25] is an unsupervised technique that is well suited for grouping continuous stream data. It takes the data points and incrementally assigns them to their respective clusters. The algorithm is described in detail in [1, 25]. For each data point, it is either assigned to one of the

DOI: 10.1002/spe

current k clusters or it starts a new cluster. If the number of clusters exceeds k, then two existing clusters are merged in order to fix the number of clusters to k.

In this paper, we have implemented a semi-supervised technique where a flat adaptive clustering technique has been used to build the training model, and later, it is used to identify anomalies. It is different from K-means as we do not fix the number of K (clusters). Moreover, we do not merge the two clusters like incremental clustering. Here, the number of cluster will be monotonically increased over time with the arrival of new training data.

Algorithm 1 Adaptive incremental clustering

```
1: procedure CLUSTERING
2:     k ← numOfcluster
3:     totalClusters ← 0
4:     for each data point n do
5:         if FitsInAnyCluster(n) then
6:             UpdateCluster()
7:         else
8:             CreateCluster()
9:             totalClusters ← totalClusters +1.
```

2.3.2. Stream data mining using Apache Storm. Figure 4 describes our network cluster using Storm. Storm's Spout receives the VMware performance data as tuples through Kafka, which ensures guaranteed delivery of the tuples. Currently, only the VMware CPU performance data are being monitored. Spout emits those tuples to Bolts. We have developed a Bolt called a clustering bolt for building the cluster model and several predicting bolts to predict anomalies.

1. Training
   Initially, the system has no information about the input data. The framework gathers stream data coming from multiple VMs through message broker (Kafka). The utility tool [ESXTOP] in VM gives the performance data (CPU, memory, and statistics). We assume that each VM sends only benign performance data during clustering and those are shipped via message broker. The clustering bolt first builds a training model from benign training data. The predicting bolt ignores all tuples coming from Spout until a model has been generated. Storm keeps the clusters information in memory. Furthermore, it does not store any tuples. A flat

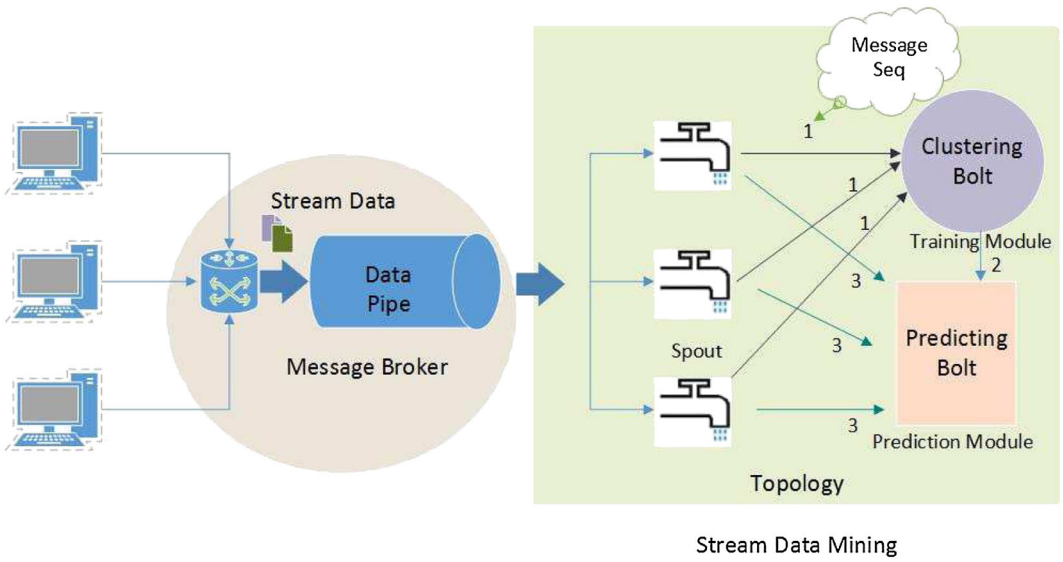

Figure 4. Technical approach for Storm-based framework

DOI: 10.1002/spe

Algorithm 2 Training in Apache Storm

```
1:  procedure TRAINING(trainingData)
2:      for each n in trainingData do
3:          BUILDCLUSTER(n)
4:  procedure BUILDCLUSTER(n)
5:      totalClusters ← 0 fitClusterIndex ← FITSINANYCLUSTER(n)
6:      if fitClusterIndex ==-1 then
7:          CREATECLUSTER ()
8:          totalClusters ← totalClusters +1.
9:      else
10:         UPDATECLUSTERINFO(fitClusterIndex, n)
11: procedure FITSINANYCLUSTER(n)
12:     maxSimilarity ← MAX_NEG
13:     maxSimilarClusterIndex ← −1
14:     for eachcluster k do
15:         distance ← EuclideanDistance(n, centroid(k))
16:         similarity ← 1/(1+distance)
17:         if similarity ≥ TH_SIM then
18:             if similarity ≥ maxSimilarity then
19:                 maxSimilarity ← similarity
20:                 maxSimilarClusterIndex ← k
        return maxSimilarClusterIndex
21: procedure UPDATECLUSTERINFO(clusterIndex, dataPoint)
22:     cluster ← FindCluster(clusterIndex)
23:     CalculateWeightedAvgCenter(cluster, dataPoint)
24:     CalculateNewRadius(cluster)
```

semi-supervised incremental clustering algorithm is used for training the model. The number of clusters is not fixed and may vary monotonically increasing order according to the training data. For each data point, if it fits into any cluster, then the cluster information is updated. Otherwise, a new cluster is created from that point. Lines 4–10 in Algorithm 2 show the building of the cluster. Euclidean distance is used to measure the distance of two data points. The distance metric is converted to a similarity measure as follows:

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

The lowest possible value of the aforementioned similarity measure function is 0 (when Distance = ∞) and the highest is 1 (when Distance = 0). It is easy to see that points with a large distance have lower similarity and points that are very close have a higher similarity. We also consider a similarity threshold of .70 when assigning points to a cluster. Furthermore, a weighted average is used when calculating the centroid of the cluster. Lines 11–20 in Algorithm 2 show how a point fits into a cluster, and lines 21–24 show the weighted-centroid modification of a cluster.

2. Testing
When the clusters are built, the clustering bolt sends the cluster information to all predicting bolts. The predicting bolt performs anomaly detection based on that information. If a tuple does not fit to any cluster, it is considered anomalous, otherwise benign. If the distance between the tuple and the centroid of a given cluster is less than the cluster's radius, the tuple is considered to be inside (i.e., belonging to) that cluster. During prediction, the clustering bolt does nothing, and it discards all the tuples coming from Spout. The details of the algorithm are given in Lines 1–9 in Algorithm 3.

DOI: 10.1002/spe

Algorithm 3 Testing in Apache Storm

```
1: procedure TESTING(testing Data)
2:     for each n in testingData do
3:         for each cluster k do
4:             distance ← EuclideanDistance (n, centroid (k))
5:             if distance ≤ Radius(k) then
6:                 n is benign
7:                 break
8:         if n does not fall into any cluster then
9:             n is an anomaly
```

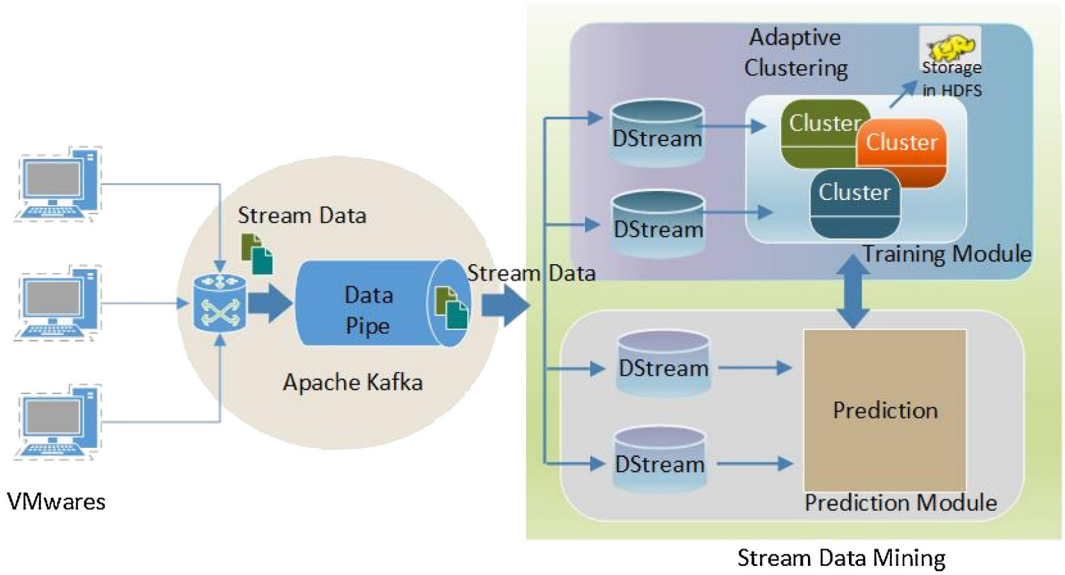

Figure 5. Technical approach for Spark-based framework

Each source or executor in Storm can be configured with multiple threads to increase parallelism. For this system, we have configured five parallel Spouts and eight parallel bolts. As the model is generated in a single bolt, modeling time cannot be reduced. However, prediction time can be easily reduced using parallelization. Furthermore, FITSINANYCLUSTER(n) method takes $O(k)$ to find the nearest cluster for a data point as there are k centroids to compare. In practice, there are typically a small number of clusters. So, the execution time in the clustering bolt heavily depends on the number of input data points that we collect parallelly during training.

2.3.3. Stream data mining using Apache Spark. Figure 5 shows the implementation in Apache Spark. The VMware performance data are transported through a Kafka queue continuously as a stream. It is split into small micro-batches called DStreams. Several transformation and action operations are performed on these DStreams to build a training model and also predict anomalies. The framework builds the model from the benign stream. After that, it predicts the incoming stream data to identify anomalies.

1. Training
   The Spark framework builds the initial training model from benign VM performance data coming from multiple VMs via message broker periodically. Algorithm 4 shows that it has two components: map and reducedByKey. The map function (Lines 3–8) takes each instance or tuple of the DStream and extracts the data point 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 centroid for a new cluster. 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

DOI: 10.1002/spe

Algorithm 4 Training in Apache Spark

```
1:  procedure TRAINING(InputDStream)
2:      trainingModel ← InputDStream.map (Tuple t){
3:      n ← t.getDataPoint()
4:      closestCluster ← 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 ← 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 ← FINDCLUSTER(index)
18:     CALWGTAVGCENTER(cluster, [dPoint1, dPoint2..])
19:     CALNEWRADIUS(cluster)
```

Algorithm 5 Testing in Apache Spark

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

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 describe this process. All the data points with the same centroid will go to a reducer. Data points with different centroids will go to different reducers. In the UPDATE function (Lines 16–19), the cluster is fetched from its index. After that, the cluster's centroid is updated by taking weighted average using the list of data instances.

Spark keeps all the clusters generated in training memory. Furthermore, it does not store any data point. A flat incremental clustering algorithm is used for training the model where we do not fixed the number of clusters. The cluster size is increased monotonically. Moreover, we have taken only the benign data coming from multiple VMs for building the training model. We have used the same training algorithm that we have mentioned in Storm-based framework (Section 2.3.2).

2. Testing

   During testing, each data point is extracted from the DStream. If any data point does not fit to any cluster, it is considered anomalous, otherwise benign. If the distance between the data point and the centroid of the clusters is less than the cluster's radius, then we consider it falling inside the cluster. A detailed algorithm is given from Lines 1–11 in Algorithm 5.

DOI: 10.1002/spe

3. CASE STUDY: ONLINE ANOMALY DETECTION IN VMWARE-BASED DATA CENTER

Our distributed framework is generic. It can be used in many fields like intrusion detection in social media, online monitoring of host health across a network, and real-time network traffic analysis. In this paper, we have applied this framework to identify anomalies in a VMware-based data center. In this case, an anomaly corresponds to over-utilization of the resources in a data center. For example, if a large number of CPU-intensive applications are running, the overall CPU utilization will be very high, and this phenomenon can be construed as anomalous.

In this section, we will briefly describe data centers and its dynamic resource scheduling using some machine learning techniques. Later, we will describe implementation details.

3.1. Data center

A data center centralizes an organization's information technology operations, infrastructure, and data storage. It houses computer systems and associated components, such as telecommunications and storage systems. Although data center designs are unique, they typically include a cloud appliance with VMware, OpenStack, Microsoft, and resource management software for managing the storage infrastructure. Some advanced data centers manage their resources dynamically to adapt to their customers' extra demands.

3.2. Distributed resource scheduler

VMware's distributed resource scheduler (DRS) [32] balances the computational workloads with available resources in a virtualized environment. It allows users to define the rules for allocating physical resources among virtual machines. DRS uses resource pools [32] which can be easily added, removed, or reorganized. If the workload among virtual machines changes drastically, DRS reschedules and redistributes the virtual machines among the physical servers. It uses affinity and anti-affinity rules [33] to improve the scheduling process. In order to do real-time dynamic scheduling, machine learning techniques (unsupervised, and supervised learning, and so on) can be applied in the scheduler to analyze stream data. Machine learning is necessary because the behavior/nature of data is not known *a priori*.

For unsupervised learning, data instances associated with a 'normal' load must first be gathered. Then, a clustering algorithm (e.g., k-means [34] and incremental clustering) 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 with '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 are 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, and k-nearest neighbor). Finally, testing (unseen) instances are predicted using the generated-classification model.

3.3. Dynamic resource management

Our distributed framework can be a key part of virtual data center automation [26]. As previously mentioned, a data center must have a virtual resource management module to control resources and provide dynamic load balancing. We envision a strong connection between our framework and this virtual module. Our framework monitors the real-time VMware performance data and reports the results to the virtual center resource manager. Finally, the resource manager can then allocate the resources to the overused VMware machines dynamically. Figures 6 and 7 show the Storm and Spark-based frameworks.

DOI: 10.1002/spe

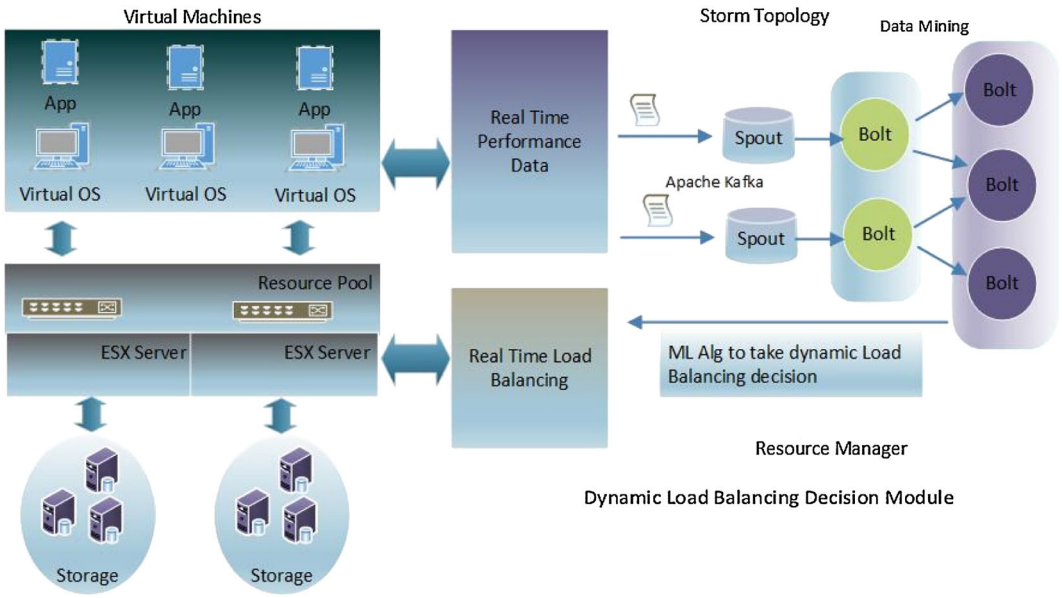

Figure 6. Dynamic resource management using Apache Storm

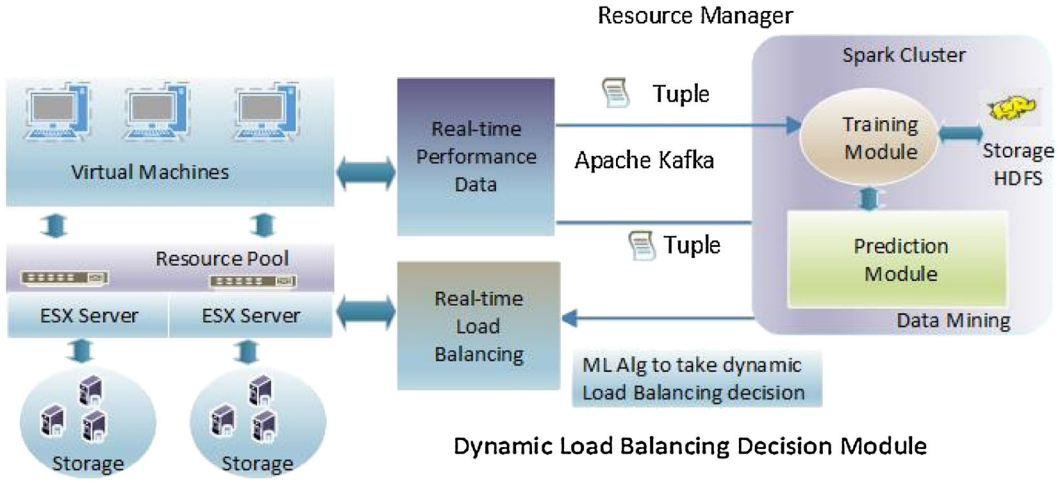

Figure 7. Dynamic resource management using Apache Spark

3.4. Implementation details

3.4.1. VMware cluster setup. Our VMware cluster consists of five VMware ESXi [35] (VMware hypervisor server) 5.5 systems. Each of the systems has an Intel(R) Xeon(R) CPU E5-2695 v2 2.40 GHz processor, 64 GB DDR3 RAM, a four TB hard disk and dual NIC card. Each processor has two sockets, and every socket has 12 cores. Therefore, there are 24 logical processors in total. Each ESXi system contains three virtual machines. Each of the virtual machines is configured with eight virtual CPUs, 16 GB DDR3 RAM and a one TB hard disk. As all the VMs are sharing the resources, performance may vary during runtime. Linux Centos v6.5 64 bit OS with the JDK/JRE v 1.7 is installed on each VM.

We have installed Apache Storm version 0.9.0-rc2 for Storm-based online anomaly detection framework. Additionally, we have installed Apache Spark version 1.0.0 for Spark-based online anomaly detection framework. We have also installed Apache Hadoop NextGen MapReduce (YARN) [28] with version Hadoop v2.2.0 and formed a cluster. Apache Spark uses this YARN cluster for its distributed file system.

Moreover, Apache ZooKeeper version 3.3.4 is used to control message flow between the message broker (Apache Kafka) and our distributed framework (both Spark and Storm).

DOI: 10.1002/spe

3.4.2. 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 intensive), more resources must be allocated to the task in order for it to complete properly. This resource increase would be reflected in the associated counters in the performance metrics. Our real-time distributed framework should diagnose this potential issue and reschedule the appropriate resources dynamically.

VMware ESXi server [35] has an administrative command [ESXTOP] which provides access to the performance data. Periodically, this raw data are captured and processed into a feature vector that acts as a data point for our framework. [ESXTOP] gives several performance statistics of CPU, memory, and disk storage. For our case study, only CPU data are considered. The 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).

3.4.3. Message broker. The VMware utility tool [EXSTOP] continuously writes the performance data to a comma separated value file. These comma separated value files are read and the data transported through the message broker. We have chosen Kafka 0.8.0 because it is stable and also compatible with both Apache Spark and Storm. Kafka creates a dedicated queue for message transportation. It supports multiple sources and sinks on the same channel. It also provides guaranteed message delivery with proper ordering, and a Kafka cluster can be formed to handle large volumes of data.

3.4.4. Training the model and prediction. Figure 8 shows the training of stream data. Our distributed framework uses a mapper and a reducer to build the training model. During training, we set an interval (e.g., 4–5 h) and assume that all the data (CPU data) in that interval are benign. During training, data from multiple VMs are coming periodically (e.g., 5 s). We have periodically updated

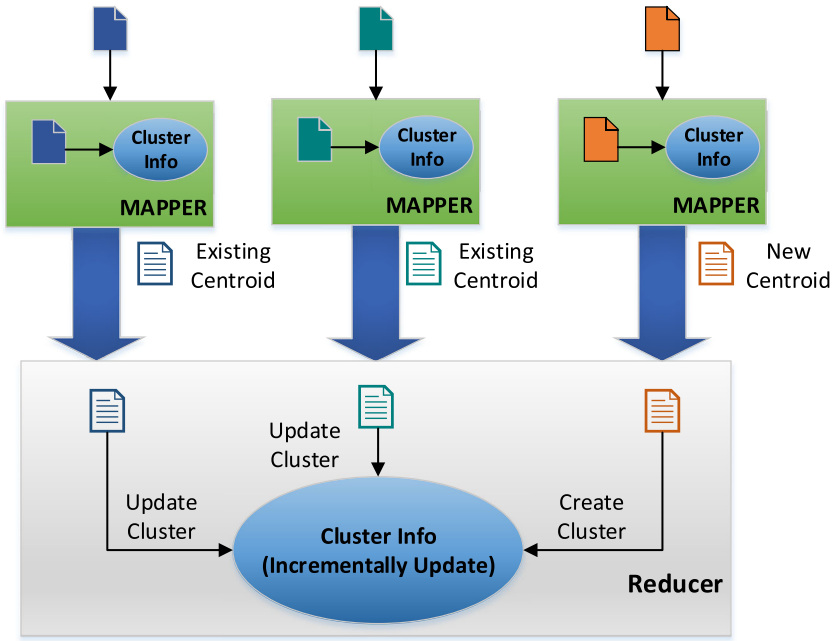

Figure 8. Training in distributed framework

DOI: 10.1002/spe

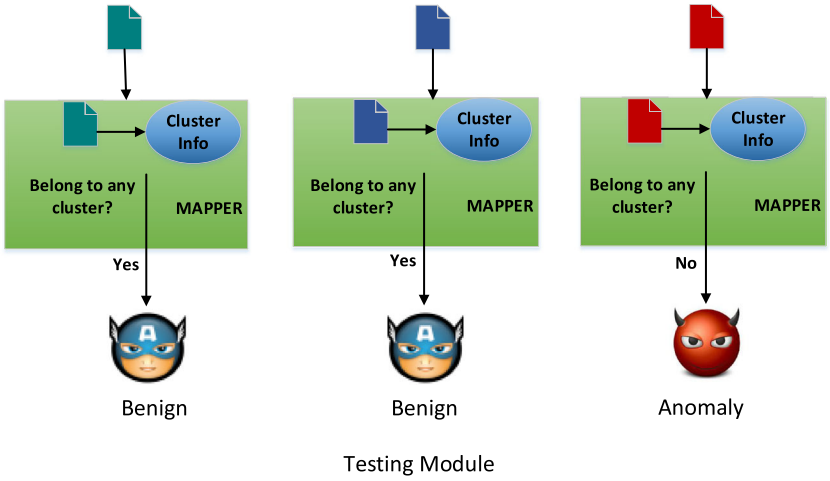

Figure 9. Testing in distributed framework

the cluster information, that is, the centroid of clusters, number of points, radius, and so on, and if any data point does not fit to any clusters, we introduce a new cluster with that data point. If a data instance matches a specific cluster, then that cluster information will be updated.

Figure 9 shows prediction of new stream data. Our distributed framework uses a mapper to predict new instances and identify anomalies. If an instance does not belong to any cluster, then it is called anomaly; otherwise, it is called benign. More specific implementation details have already been discussed in Section 2.

3.4.5. Updating the model. We build training model from benign data only, but this stream data may evolve over time. Therefore, the training model must be updated periodically in order to maintain its efficacy. The training model can be updated continuously or at a fixed time interval, which might be set experimentally. A cluster model can be easily updated to incorporate new data points. Additionally, it is easy to persist cluster information (e.g., to disk) so that the model does not need to be regenerated. In our experiments, we do not update the training model.

3.4.6. Scalability. Our generic framework is highly scalable. The message broker, Kafka [22], can be formed into a cluster to scale to increasing data demands. Moreover, more executors/workers can be added to increase parallelism and accommodate larger volumes of data.

3.4.7. Generating anomalies. Virtual machines become CPU intensive when their CPU metrics increase. We have programmatically increased CPU load to simulate anomalous behavior of data. For example, a program with an infinite loop may increase CPU load to 100%. We have also carried out some expensive database read/write operations to increase the value of several counters in CPU metrics (e.g., CPU usage, processor time, and core utility time).

4. EXPERIMENTAL RESULTS

Our real-time anomaly detection framework monitors continuous CPU performance data from multiple VMs periodically. It builds a model from benign training data. After that, it identifies a VM as anomalous if its CPU usages deviates substantially from the training model. In this section, we will describe the experiments with result that support the robustness of our framework.

4.1. Dataset

The framework does not need to know the data *a priori*. It will automatically train and build the model, and later, use the model to identify anomaly. We have considered a sufficient number of

DOI: 10.1002/spe

Table I. Spark Cluster

Component	Number of workers
Streaming source	5
Clustering	8
Prediction	8

Table II. Training

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

training instances empirically so that our model has a consistent number of clusters. We have created two scenarios to capture data for our framework during experiments. For the first scenario, we run several MapReduce/Spark jobs on our VM-based cluster. We then monitor the real-time CPU performance metrics from all the VMs. We have called this dataset D1. Dataset D1 has both training and testing data. The training data are benign only where testing data contain both benign and anomaly. We have increased the CPU metrics for generating anomalous stream data by using tools like stress and running programs.

For the second scenario, we have installed the Yahoo Cloud Serving Benchmark [36] framework to generate an extensible workload. We run the cloud service benchmark with different workloads on our VM-based cluster and monitor the run-time properties continuously to capture stream data. We have called this dataset D2. It has also separate training and testing data. The database system is a simple MySQL database with billions of rows inserted. Data are loaded in the load phase of the Yahoo Cloud Serving Benchmark run. Later, transactions such as read and update are run on the database system to increase the workload. Each machine in cloud emits the performance data that captures the network, I/O, CPU, and memory usages during the workload run. For our usage, we have only taken the CPU data. We have used millions of read/write operations on the database to generate anomalous stream data.

During training, we have selected the training duration and also the streaming window length by experiment. This helps us to obtain sufficient training data to avoid data over-fitting. Moreover, we have divided the datasets into training and testing phases which do not overlap. Furthermore, our framework reports with higher accuracy in both datasets.

4.2. Results

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

As data come continuously, a pre-specified time window is used to collect the training instances for building the model. So, the detection system is not affected by any non-synchronized samples from sources. Here, 10,000 data instances are fed to the flat incremental clustering algorithm to generate the model. At the end, the algorithm generates 63 clusters for dataset 1 (D1) and 134 for dataset 2 (D2). These clusters are generated dynamically. Table II shows statistics related to the generated model. The number of clusters depends on the number of sources, the nature of data, and so on. If we increase the number of sources, the number of clusters may vary. Here, we have not fixed the number of clusters. During training, we have also considered a sufficient number of training instances empirically, so that our model has a consistent number of clusters. We have considered two datasets D1 and D2. Although they contain the same CPU metric information, the statistics are different. D1 contains the CPU statistics for Spark jobs where D2 represents high volume of database operations. As the statistics are different, both D1 and D2 give different number of clusters.

Table III shows the accuracy of the framework. A time window is also used for data prediction and testing. A total of 3500 testing instances have been taken from both datasets. Three thousand of them are benign, and 500 are anomalous. Our framework correctly predicts 2995 benign data out of 3000 and also identifies 490 anomalous data out of 500 for Dataset 1 (D1). It also correctly predicts 2972 benign data out of 3000 and also identifies 496 anomalous data out of 500 for Dataset 2 (D2).

DOI: 10.1002/spe

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

FNR, false negative rate; FPR, false positive rate; TNR, true negative rate; TPR, true positive rate.

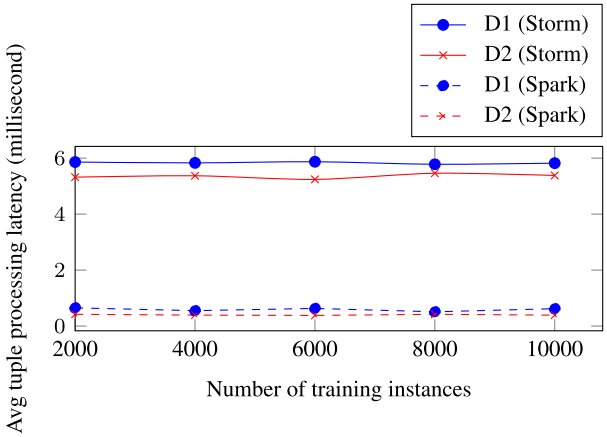

Figure 10. Comparing average tuple processing latency during clustering

Table III reflects the overall accuracy statistics. So our framework has higher accuracy in anomaly detection. In this table, TPR (True Positive Rate) is the proportion of actual anomalies which are correctly identified and TNR (True Negative Rate) is the proportion of actual benign data which are correctly identified. Moreover, FNR (False Negative Rate) is the proportion of actual anomalies which are misclassified as benign and FPR (False Positive Rate) is the proportion of actual benign which are misclassified as anomalous.

4.3. Comparison with Storm

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

Figure 10 shows the average process latency of a tuple for clustering during training for both 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 plot them. The graph shows that the clustering takes almost 5.85 (first dataset D1) and 5.30 ms (second dataset D2) on average for Storm and almost 0.6 (D1) and 0.4 ms (D2) on average for Spark. From this 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.

Figure 11 shows the average process latency of a tuple for predicting anomalies during testing for both the Spark and Storm implementations. We take 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 (first dataset D1) and 0.55 ms (second dataset D2) on average for Storm (when the graph reaches saturation) and almost 0.038 (D1) and 0.03 ms (D2) on average for Spark. From this 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. This is expected because Spark provides in-memroy computation which is faster than Storm.

4.4. Experiment of parallelism

We vary the number of slave nodes in Spark when we process data. In Figure 12, we see that the average processing latency (millisecond) of a tuple during clustering does not increase although

DOI: 10.1002/spe

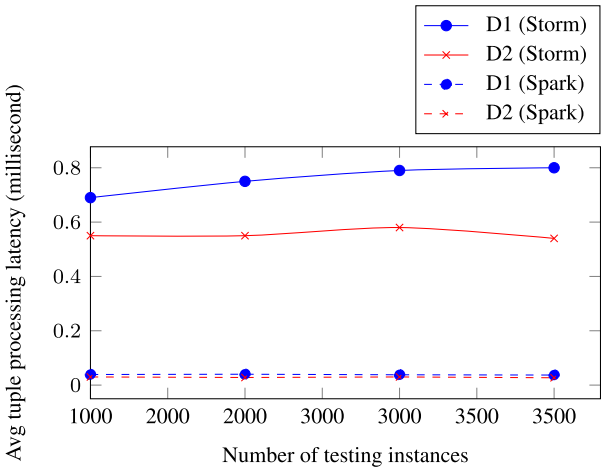

Figure 11. Comparing average tuple processing latency during prediction

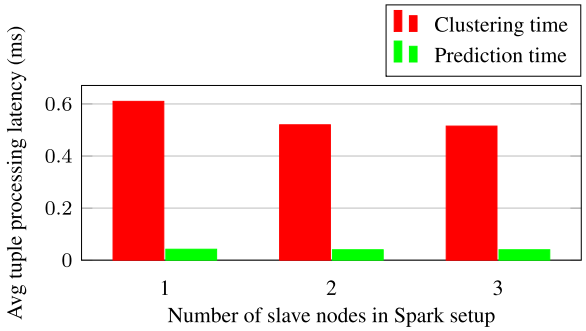

Figure 12. Average tuple processing latency for different number of slave nodes in Spark setup

we increase the number of slave nodes. For single node, the latency is almost 0.61 ms, and then, it decreases to almost 0.52 ms for double and triple nodes. 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 set the similarity threshold at .70 during clustering. Finally, as the number of clusters is not fixed, it is possible that many of the generated clusters will only have a single point. However, these clusters are not influential during testing because they do not have a radius. Unless exact matching to these data points occurs, no testing data points will fall inside these clusters. Post-pruning of these single point clusters can be carried out to eliminate this issue.

5. RELATED WORK

Anomaly detection naturally lends itself to clustering and therefore has been explored as a k-means clustering problem in the literature. However, for large datasets, the time complexity of k-means is prohibitive. Additionally, k-means' initial centroid problem where the initial centroids are chosen randomly. This may create bad clusters and also the algorithm converges very slowly. The initial centroid problem is circumvented by k-medoids [37] which starts with k centers, modifying the centers repeatedly and at random thereby improving the sum of squared error. Outlier detection on stream data has been addressed by Assent *et al.* [38]. Their algorithm, known as AnyOut, utilizes hierarchical clustering. The deviation between an object and a cluster is determined by an outlier score. However, structural complexity of hierarchical clustering translates to high processing time and poses a scalability challenge in case of distributed systems.

DOI: 10.1002/spe

In distributed systems, anomaly detection entails processing of large-scale datasets with the aid of distributed frameworks. Among them, Hadoop[11] and MapReduce perform extremely well on offline data but lack the capabilities for handling real-time stream data. Similar technologies, for example, HBase [13] and BashReduce [39], are not designed for stream data processing. Apache Mahout [14], a MapReduce-based machine learning framework, provides implementations for k-means and StreamingKMeans which only run in batch mode. Anomaly detection frameworks based on Hadoop and MapReduce have been proposed by Yu *et al.* [40] and Gupta *et al.* [41]. Because of the inherent restrictions of the underlying frameworks, their approaches are not readily extensible to real-time processing.

The state-of-the-art in real-time data processing includes Apache Storm [16] and Apache S4 [17]. Spark, being faster than Storm [30], has been used as the underlying framework of our approach. Mohiuddin *et al.*[1] performed a real-time anomaly detection framework for VMware performance data which is based on Apache Storm. Mohiuddin *et al.* [2] also presented the Spark-based anomaly detection technique. Our detailed-comparative analysis of Storm-based and Spark-based anomaly detection frameworks confirms that Spark incurs lower average tuple execution time. Traditional approaches to anomaly detection usually focus on accuracy and pay less attention to performance. However, performance is a key contributing factor to the efficacy of a real-time anomaly detector, and our approach aims to find a framework that excels both in accuracy and performance. Although Spark provides a StreamingKMeans implementation, it assumes pre-domain knowledge of data. In contrast, our approach is agnostic about the VMware performance data while building the model. Therefore, we elected to use our flat incremental clustering instead of Spark's StreamingKMeans.

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented a real-time anomaly detection framework based on Apache Spark. Powered by Spark's in-memory computing, parallel processing and fault tolerance, our framework can process large amounts of stream data in real time. We have also experimentally shown that Spark outperforms Storm substantially. Moreover, our real-time framework can be integrated with VMware's dynamic resource management system as well as various other monitoring systems to measure operational performance.

Our framework is generic and can play a key role in real-time intrusion detection system in Cyber security with higher accuracy. The framework can monitor network traffics periodically and builds the training model. Later it can be used to identify suspicious or anomalous network traffics that potentially indicate Cyber attacks.

In the future, we will aspire to implement other machine learning algorithms in our framework and compare them. We will explore the feasibility of incorporating other performance metrics, namely, memory usage, storage statistics, and performing a correlation analysis to detect anomalies without any degradation of accuracy or performance. Moreover, the training data may contain noise. We will build an ensemble-based model to minimize the effect of noise during training.

Real-time anomaly detection using statistical techniques is a good area to explore in the future. Point-based anomaly detection techniques attempt to flag a single point as anomalous or benign. In reality, a substantial deviation of data from multiple sources represents a true anomalous condition. For example, a single VM may generate high CPU/memory usage due to a malfunction for only a short amount of time. This condition can be considered a spike or sudden outlier but not the anomalous condition of a VM-based data center. On the other hand, if a number of VMs continuously show higher (unacceptable) CPU/memory usage, then it can be considered as an anomalous condition for the data center. Statistical techniques for detection anomaly in real time could be the ideal solution for this scenario.

Furthermore, we can attempt to fuse and analyze heterogeneous data from a variety of sources and in a variety of formats in order to detect anomalies. For example, unstructured data from sensor devices may need to be combined with structured data. From a data center perspective, resource usage data from different system components, for example CPU and memory, differ in format and semantics. So, we can design a framework that can reconcile these two different sources and also easily accommodate itself to higher levels of heterogeneity.

DOI: 10.1002/spe

ACKNOWLEDGEMENTS

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 US 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. Solaimani M, Khan L, Thuraisingham B. Real-time anomaly detection over VMware performance data using storm. *The 15th IEEE International Conference on Information Reuse and Integration*, San Francisco, USA, 2014; 458–465.
2. Solaimani M, Iftekhar M, Khan L, Thuraisingham B, Ingram JB. 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, 2014; 66–73.
3. Mustafa A, Solaimani M, Khan L, Chiang K, Ingram J. 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, Vienna, Austria, 2014.
4. Mustafa A, Haque A, Khan L, Baron M, Thuraisingham B. Evolving stream classification using change detection. *10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, October 22–25, 2014*, Miami, Florida, USA, 2014; 154–162.
5. Yao Y, Sharma A, Golubchik L, Govindan R. Online anomaly detection for sensor systems: a simple and efficient approach. *Performance Evaluation* 2010; **67**(11):1059–1075.
6. Lee W, Stolfo SJ, Chan PK, Eskin E, Fan W, Miller M, Hershkop S, Zhang J. Real time data mining-based intrusion detection. *In Darpa Information Survivability Conference & Exposition II, 2001. Discex'01. Proceedings*, Vol. 1: IEEE, Anaheim, CA, USA, 2001; 89–100.
7. Abuaitah GR. Anomalies in sensor network deployments: analysis, modeling, and detection. *Ph.D. dissertation*, Wright State University, OH, USA, 2013.
8. Hayes MA, Capretz MAM. Contextual anomaly detection in big sensor data. *In Big Data (BigData Congress), 2014 IEEE International Congress on*: IEEE, 2014; 64–71.
9. Janeja VP, Azari A, Namayanja JM, Heilig B. B-dids: mining anomalies in a big-distributed intrusion detection system. *In Big Data (Big Data), 2014 IEEE International Conference on*, 2014; 32–34.
10. Savage D, Zhang X, Yu X, Chou P, Wang Q. Anomaly detection in online social networks. *Social Networks* 2014; **39**(0):62 –70.
11. Apache hadoop. Available at: http://hadoop.apache.org/ [Accessed 15 October 2013].
12. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM* 2008; **51**(1):107–113.
13. Apache hbase. Available at: https://hbase.apache.org/ [Accessed 12 October 2015].
14. Apache mahout. Available at: https://mahout.apache.org/ [Accessed 1 February 2014].
15. Chang F, Dean J, Ghemawat S, Hsieh W.C, Wallach D.A, Burrows M, Chandra T, Fikes A, Gruber R.E. Bigtable: a distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 2008; **26**(2):4–4.
16. Storm – distributed and fault-tolerant realtime computation. Available at: http://storm.incubator.apache.org/ [Accessed 8 December 2013].
17. S4. Available at: http://incubator.apache.org/s4 [Accessed June 2013].
18. Apache Spark. Available at: http://spark.apache.org/ [Accessed 30 May 2014].
19. Apache Spark. Available at: http://spark.apache.org/streaming/ [Accessed 30 May 2014].
20. Balasingam B, Sankavaram MS, Choi K, Ayala DFM, Sidoti D, Pattipati K, Willett P, Lintz C, Commeau G, Dorigo F, Fahrny J. Online anomaly detection in big data. *Information Fusion (Fusion), 2014 17th International Conference*, Salamanca, Spain, 2014; 1–8.
21. Camacho J, Macia-Fernandez G, Diaz-Verdejo J, Garcia-Teodoro P. Tackling the big data 4 VS for anomaly detection. *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on*, Toronto, Canada, 2014; 500–505.
22. Apache Kafka. Available at: http://kafka.apache.org/.
23. Message broker. Available at: http://en.wikipedia.org/wiki/Message_broker [Accessed 7 December 2015].
24. Publish subscribe pattern. Available at: http://en.wikipedia.org/wiki/Publish-subscribe_pattern [Accessed 18 December 2015].
25. Charikar M, Chekuri C, Feder T, Motwani R. Incremental clustering and dynamic information retrieval. *SIAM Journal on Computing* 2004; **33**(6):1417–1440.
26. VMware. *Automating the virtual datacenter*. Available at: https://www.vmware.com/files/pdf/avd_wp.pdf.
27. Apache zookeeper TM. Available at: http://zookeeper.apache.org/doc/r3.3.4/ [Accessed 28 November 2011].
28. Apache hadoop nextgen mapreduce (YARN). Available at: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html [Accessed 29 June 2015].

DOI: 10.1002/spe

29. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. *9th USENIX Conference on Networked Systems Design and Implementation*, San Jose, CA, USA, 2012; 2–2.
30. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, Mccauley M, Franklin M, Shenker S, Stoica I. Fast and interactive analytics over hadoop data with spark. *USENIX;* ligin 37.4 (2012);45–51.
31. Zaharia M, Das T, Li H, Shenker S, Stoica I. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. *4th USENIX Conference on Hot Topics in Cloud Ccomputing, USENIX Association*, Boston, AA, USA, 2012; 10–10.
32. *Why use resource pools?* Available at: 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.
33. VMware. *Using DRS affinity rules*. Available at: http://pubs.vmware.com/vsphere-51/index.jsp.
34. Tan Kumar. *Introduction to Data Mining*. no. 3, Pearson Education, Limited: NY, USA, 2005.
35. VMware. *Vsphere ESX and ESXI info center.* Available at: http://www.vmware.com/products/esxi-and-esx/overview.
36. Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. *In Proceedings of the 1st ACM Symposium on Cloud Computing*: ACM, Indianapolis, IN, USA, 2010; 143–154.
37. Kaufman L, Rousseeuw PJ. *Finding Groups in Data: An Introduction to Cluster Analysis*, Wiley series in probability and mathematical statistics. Applied probability and statistics. Wiley, 2005.
38. Assent I, Kranen P, Baldauf C, Seidl T. Anyout: anytime outlier detection on streaming data. In *Database Systems for Advanced Applications*. Springer, 2012; 228–242.
39. Frey E. *Bashreduce*, 2009. Available at: http://rcrowley.org/2009/06/27/bashreduce [Accessed 27 June 2009].
40. Yu L, Lan Z. A scalable, non-parametric anomaly detection framework for hadoop. *In Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*: ACM, Miamy, FL, USA, 2013; 22–22.
41. Gupta M, Sharma AB, Chen H, Jiang G. Context-aware time series anomaly detection for complex systems. *In WORKSHOP NOTES*, Austin, TX, USA, 2013; 14–14.

DOI: 10.1002/spe