Context-Aware Time Series Anomaly Detection for Complex Systems

Manish Gupta[1], Abhishek B. Sharma[2], Haifeng Chen[2], Guofei Jiang[2]
[1]UIUC, [2]NEC Labs, America

Abstract

Systems with several components interacting to accomplish challenging tasks are ubiquitous; examples include large server clusters providing "cloud computing", manufacturing plants, automobiles, etc. Our relentless efforts to improve the capabilities of these systems inevitably increase their complexity as we add more components or introduce more dependencies between existing ones. To tackle this surge in distributed system complexity, system operators collect continuous monitoring data from various sources including hardware and software-based sensors. A large amount of this data is either in the form of time-series or contained in logs, e.g., operators' activity, system event, and error logs, etc.

In this paper, we propose a framework for mining *system operational intelligence* from massive amount of monitoring data that combines the time series data with information extracted from text-based logs. Our work is aimed at systems where logs capture the *context* of a system's operations and the time-series data record the *state* of different components. This category includes a wide variety of systems including IT systems (compute clouds, web services' infrastructure, enterprise computing infrastructure, etc.) and complex physical systems such as manufacturing plants. We instantiate our framework for Hadoop. Our preliminary results using both real and synthetic datasets show that the proposed context-aware approach is more effective for detecting anomalies compared to a time series only approach.

1 Introduction

Complex distributed systems with multiple components interacting to accomplish challenging tasks are drivers of our economic growth. Examples of such systems include large server clusters providing cloud computing services, critical infrastructure such as power plants and power grids, manufacturing plants, automobiles, etc. Our relentless efforts to improve the capabilities of these systems inevitably increase their complexity as we add more components or introduce more dependencies between existing ones to provide new functionality. The complex nature of such systems can lead to failures like the Amazon Outage [1]. Such unanticipated faults and unknown anomalies are a major source of service disruption in complex systems. System operators try to reduce such incidents through better visibility achieved by collecting continuous monitoring data from multiple vantage points. The ultimate goal is to transition from the current practice of reactive maintenance to a more pro-active approach where operators can predict system behavior and address possible service disruptions.

A significant barrier to making this leap from improved visibility to pro-active system management that goes beyond the current simple rules based solutions is the heterogeneity of monitoring data. Monitoring data can be in the form of time series and structured or semi-structured text logs (operators' activity, system events, and error logs), and unstructured text. Mining of log and time series data for fault and anomaly detection is an area of active research. However, previous work analyzes logs and time series data *separately* (Section 8) which has several shortcomings when detecting anomalies in distributed systems.

Anomaly detection using only logs. Fault detection using only logs suffers from two major shortcomings. (1) *Incomplete coverage*: It is challenging to *a priori* determine the right granularity for logging, and often programmers/designers do not want to log several relevant details fearing that the log sizes and the associated time overheads may be large. (2) *Lack of root-cause visibility*: Logs contain information at conceptual/application level and hence often do not contain sufficient information to identify the cause of faults at the component level.

Anomaly detection using only time series. System monitoring using just the time series data suffers from the *lack of a global semantic view*. For example, CPU utilization measurements of a server supply information that is context insensitive. In the absence of context, any significant change in the metrics might seem quite abnormal, even to a human, though some of them might be due to normal system events. Such scenarios can lead to a lot of false positives. Another drawback is the *proliferation of alarms*. A method that just uses time series data can raise multiple alarms (one for a fault in each time series) even when many of these faults are actually related to the same underlying problem. But the lack of a global view prevents identification of any commonality between these faults.

Our hypothesis is that performing data mining *jointly* on logs and time series data can address these shortcomings. We refer to it as *context-aware time series anomaly detection*. The following example provides an illustration.

Importance of Context. Figure 1 shows the variation in

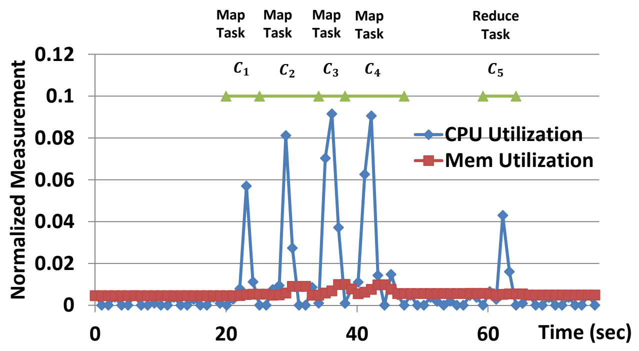

Figure 1: Logs and OS Metrics are Complementary

the normalized operating system (OS) performance metrics such as CPU and memory utilization for a server executing MapReduce [3][1] tasks. The measurements have been normalized for clear display. The contexts $C_1, \ldots, C_5$ (represented by green lines) denote the server's state. We assume that the server's state changes whenever a task starts or stops. Suppose we define an anomaly as unusual values in the time series data for CPU and memory utilization. If we detect anomalies based only on time series data on CPU and memory utilization, we will report one whenever a new task starts or stops. However, when we combine the information on the server's states (extracted from logs and characterized by variables like number of running tasks) along with the time series data, then we can explain the observed CPU and memory utilization behavior and not consider them as anomalous. For example, if we know that a new CPU resource intensive task has started on the server, we expect the CPU utilization to memory utilization ratio to go high, and so such an increase will not be flagged as an anomaly.

Summary of our Contributions

(1) We introduce the notion of context-aware anomaly detection in distributed systems by integrating the information from system logs and time series measurement data. (2) We propose a two-stage clustering methodology to extract context and metric patterns using a PCA-based method and a modified K-Means algorithm (see Figure 2). Anomalies are then detected based on these context and metric patterns. (3) We instantiate our framework for Hadoop, and show that our approach can detect interesting and meaningful anomalies from real Hadoop runs and can accurately discover injected anomalies from synthetic datasets.

The rest of this paper is organized as follows. We provide an overview of our work and highlight the main challenges that it addresses in Section 2. In Section 3, we provide background on Hadoop and define the context-aware time series anomaly detection problem. Section 4 describes the context and metric patterns, and the methodology used to extract them. In Section 5, we present our anomaly detection approach, and discuss various interesting aspects in

[1]We briefly describe the MapReduce framework in Section 3

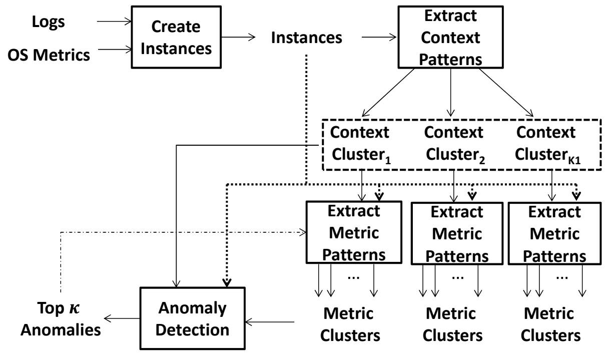

Figure 2: Anomaly Detection Methodology

Section 6. We present experimental results in Section 7 and the related work in Section 8. We conclude with a summary of our work in Section 9.

2 Overview: Challenges and the Proposed Solution

Challenges: There are three main challenges in combining log and time series for context-aware anomaly detection: (1) How do we *represent* the structured or semi-structured log data and the multivariate time series data together? (2) How do we *model* the combined data? (3) What is our *definition of an anomaly* and how do we *detect them*? Figure 2 shows the main steps of our proposed methodology to address these three challenges. We instantiate this framework for Hadoop.

Combining system logs and time series. We assume that system logs capture the current operational context, and define *context variables* and extract their time-varying values from the logs. This process exploits the fact that log data is often structured or semi-structured. We also define system events that signal a context change and assume that they are recorded in system logs. (In general, we may not have enough domain knowledge about a system to define context changing events. However, there exist tools such as Sisyphus [18] and PADS [5] that can help with this task.)

We combine logs and time series data through our definition of an *instance*. An instance spans the interval between two consecutive context changing events on a component. For example, we will define an instance for each of $C_1, \ldots, C_5$ in Figure 1. Each instance consists of context variables and multivariate time series data collected during the interval associated with it. For $C_1, \ldots, C_5$ in Figure 1, the context variables can be the number and type (map or reduce) of running tasks, and the time series data consists of the CPU and memory utilization measurements. Each component in a distributed system can have its own set of instances, i.e., for a large cluster of servers, we can extract instances at the granularity of individual servers.

Iterative clustering for anomaly detection. It is reasonable to assume that a system's components will display similar behavior under similar context. Hence, given the context values for an instance, we define an instance to be anomalous if the time series data associated with it contains unusual or inconsistent values. To detect anomalies as per this definition, we first cluster instances based on their context variables. We refer to these clusters as *context clusters*. Then we partition the instances belonging to a context cluster based on their time series data, and refer to these clusters as *metric patterns*. We formally define the similarity measures for context variables and time series data in Section 4. We want to point out that our approach of identifying similar context across instances is valid for two different scenarios: when similar contexts appear over time on the same component or when different components provide similar functionality. The latter case happens when a functionality is replicated across multiple components, e.g., when popular web sites are hosted across multiple servers in order to support large demand. Hence, we refer to the similarity in context variables across instances as *peer similarity*, and the similarity in time series data for similar context as *temporal similarity*. We assign a score to each instance based on the distance of its multivariate time series from the closest metric pattern, and flag instances with high score as anomalous (we define this *anomaly score* in Section 5).

Instantiating our framework for Hadoop. Our approach is aimed at distributed systems with a large number of components that often have functionality replicated across components to support heavy workload. It also needs data on system events and errors logged for each component, and time series measurements from each component. One widely used system that meets these two criteria, and is also easy to deploy is Hadoop, the de facto industry standard for building large, parallel data processing infrastructure. Hence, we instantiate our framework for Hadoop, i.e., we use Hadoop system logs to demonstrate how to extract context variables and instances from real world data, and also collect time series measurements on resource usage (CPU, memory, disk, and network utilization) from servers used to deploy Hadoop. Another advantage of using Hadoop is that we can inject anomalies while the system is running.

3 Problem Definition for Hadoop

In this section, we briefly introduce Hadoop and define our problem.

3.1 MapReduce and Hadoop The MapReduce programming model facilitates processing of a massive dataset by partitioning it into smaller chunks that are distributed across individual machines in the cluster. Data processing is done in two phases – Map and Reduce. During the Map phase the same (Map) function is applied to all the chunks belonging to a data set in parallel at individual machines. The output of the Map phase (spread across the cluster) is fed into the Reduce phase. The Reduce phase consists of multiple Reduce tasks that are run in parallel, and produces the final output. Thus, a MapReduce job consists of multiple Map and Reduce tasks.

Hadoop is an open-source implementation of the MapReduce framework. In a typical Hadoop cluster, the master and each slave run on separate servers. The master (slave) node runs the *NameNode* (*DataNode*) and *JobTracker* (*TaskTracker*) to manage the distributed file system and job execution, respectively. Each slave node is configured with a fixed number of Map and Reduce slots and each running task takes up one slot.

Hadoop collects several logs at the master as well as slave nodes [9]. The TaskTracker and DataNode processes record task level events, errors and resource usage statistics on each slave node. These task level information are aggregated as job level events and statistics in the JobTracker and NameNode logs at the master. The master node also records control decisions, e.g., which task is scheduled on which slave node, as well as information on any errors or failures of slave nodes. Together, these logs capture the application level semantics, i.e. the context, of an operational Hadoop cluster. From each server in a Hadoop cluster, we can also collect time series data on OS performance metrics such as CPU and memory utilization.

3.2 Context and Metric Variables for MapReduce Consider a Hadoop cluster with M machines. We parse the Hadoop logs to identify the start and end times of the Map and Reduce tasks run on each machine. This gives us a list of *events* and the time at which they occur for each machine. Based on this information, we define the set of instances for each machine where an instance spans the duration between two consecutive events. A new instance is thus created whenever a Map or Reduce starts or ends on the machine. We discard instances corresponding to time intervals during which no task (Map or Reduce) is running. We compute instances for each machine and then aggregate them into a set represented as $\mathcal{I} = \{I_1, I_2, \ldots, I_L\}$.

Instance Each instance is identified by a machine and its time duration. It consists of two components (1) Context values and (2) Metric time series. Thus, we represent an instance I_r as (C_r, M_r) where C_r represents the vector of context variables and M_r represents the vector of metric variables.

Context Variables Context variables characterize the Hadoop tasks running on a machine corresponding to the time duration of an instance. We capture the number and nature of Hadoop tasks using the #Maps, #Reduces and task counters. Task counters are of 5 main types: (1) Map counters: Map Input/Output Bytes/Records (2) Reduce counters: Reduce Input/Output Records, Reduce Shuffle Bytes (3) Combine counters: Combine Input/Output Records (4) Machine counters: CPU Milliseconds, Physical Memory Bytes, Spilled Records (5) HDFS (Hadoop Distributed File System) counters: HDFS Bytes Read/Written. Note that these counters are all task specific and are available in the job history logs after the job gets finished.

For a particular instance (identified by a machine and a duration), there might be many tasks running concurrently. The context values for that instance are then computed as the average of the values across all these tasks.

Time series Metric Variables For every machine we can record a variety of metrics that can help us understand the performance of the machine. We use system level tools to get the values of the following metrics for each machine every second. (1) CPU Utilization (2) Memory Utilization (3) Disk Read Operations (4) Disk Write Operations (5) eth0 RX/TX Bytes. Other system specific measures like eth0 RX/TX Errors, eth0 RX/TX Drops, Memory Buffers, Mem SwapFree, Used File Descriptors, sda Disk Read/Write Sectors, sda Disk pending IO Operations, Interrupts, OS Context Switch, OS Running Processes, OS Blocked Processes could also be exploited for the purpose. However, we focus on showing the benefits of combining the context variables with a simple set of metric variables to obtain better results compared to using just the metric variables without any context information.

3.3 Context-Aware Time Series Anomaly Detection Problem

Given the instances represented in terms of context and metric values, the anomaly detection problem can be defined as follows.

Input: Instances $I_1 = (C_1, M_1), \ldots, I_L = (C_L, M_L)$.
Output: Top κ instances with highest anomaly scores.

We discuss extraction of patterns in Section 4 and define outlier scores and sum up the entire anomaly detection algorithm in Section 5.

4 Pattern Extraction

In this section, we discuss how to extract patterns from the instances and their context supplied by the job history logs as well as from the OS metrics time series. Since the Hadoop logs capture a global semantic view of the system, we propose to first cluster the instances using the context variables to extract context patterns. For each context cluster, we find metric patterns by clustering the corresponding OS metrics time series.

4.1 Extraction of Context Patterns

Context patterns are modeled as K_1 Gaussians in the multi-dimensional space defined by the context variables and can be discovered using K-Means. However, we need to pre-process the data before applying K-Means clustering. Some context variables can have large values while some are quite small. This can cause the K-Means clusters to be biased towards the attributes with large values. Hence, we normalize all the attributes such that they have zero mean and unit variance. We also find that some context variables are highly correlated with each other and can artificially bias the clustering. E.g., for our workload, the following features were highly correlated: bytes written, file bytes written and HDFS bytes written. We calculate the correlation between each pair of variables and retain only one variable from a group of highly correlated variables.

A context pattern represents the logical state of a machine (running many Map jobs, moving many HDFS blocks, etc.) for the time duration corresponding to the instance. Such states capture the diversity of MapReduce computation including the type and intensity of workload, number of Map and Reduce tasks in a job, number of memory-heavy versus CPU-heavy tasks, etc.

4.2 Extraction of Metric Patterns

Context patterns can cluster the dataset with respect to the logical view of the system. To further characterize the dataset within each context, we cluster the metric variables corresponding to all the instances belonging to the same context cluster. Metric patterns are defined with respect to a particular state (context) of a machine. Essentially, metric patterns denote the usual patterns observed in the OS metrics when the machine is in a logical state described by its context variables. Note that the OS metrics associated with an instance are represented as a multi-variate time series. The number of components of this time series is equal to the number of metrics and the length of the time series corresponds to the time duration of the instance. Thus the problem of clustering metric variables is equivalent to clustering of a multi-variate time series dataset.

Clustering multivariate time series is a challenging task. For the proposed problem, a technique is needed such that it can cluster multi-variate time series that (1) may not be of the same size, i.e., may not be defined over the same length of time period, and (2) may not be defined over the same exact time interval (asynchronous). Clustering of general objects needs a definition of a measure of similarity between a pair of objects. We want a measure for *similarity* between two multivariate time series that stresses both on the similarity of interactions between two (or more) univariate time series from a multivariate time series set as well as on the similarity of interactions (across time) within a univariate time series. To capture such semantics and to avoid the problem of uneven lengths of the multi-variate time series, we first define the similarity between two multivariate time series in the space of their principal components.

Similarity Between Two Multi-Variate Time Series Consider two multi-variate time series corresponding to the metrics part M_a and M_b of the instances I_a and I_b. Let the time series be defined over time intervals of length l_1 and l_2 respectively. Thus the multi-variate time series can be represented using matrices ${M_a}^{l_1 \times c}$ and ${M_b}^{l_2 \times c}$ respectively where c is the number of metric variables. For both the matrices, we can compute the top k principal components (that can capture $\geq$ 95% variance) to obtain smaller subspaces $P^{c \times k}$ and $Q^{c \times k}$ respectively. Similarity between the two multi-variate time series can be expressed in terms of the similarity between their principal components as follows.

$$sim(M_a, M_b) = \frac{trace(P'QQ'P)}{k} \tag{4.1}$$

Geometrically, this means that the similarity between the two multivariate time series is the average sum of squares of the cosines of the angles between each principal component in P and Q. Thus, we can rewrite Eq. 4.1 as follows.

$$sim(M_a, M_b) = \frac{1}{k}\sum_{i=1}^{k}\sum_{j=1}^{k} cos^2\theta_{ij} \tag{4.2}$$

where θ_{ij} is the angle between the i^{th} and the j^{th} principal components in P and Q respectively.

Further, Eq. 4.2 can be modified using a weighted average, with the amount of variance explained by each principal component as weights, as follows.

$$sim(M_a, M_b) = \frac{\sum_{i=1}^{k}\sum_{j=1}^{k} \lambda_{Pi}\lambda_{Qj} cos^2\theta_{ij}}{\sum_{i=1}^{k} \lambda_{Pi}\lambda_{Qi}} \tag{4.3}$$

where λ_{Pi} is the eigen value corresponding to the i^{th} principal component (eigen vector) in P. Note that the similarity value lies between 0 and 1.

Modified K-Means to obtain Metric Patterns Algorithm 1 shows the modified K-Means algorithm for clustering of multi-variate time series datasets using the PCA-based similarity measure given in Eq. 4.2. The metric patterns (clusters) are first initialized randomly (Step 1). Then, the aggregate dataset representative of a cluster is created by concatenating matrices of all constituent datasets that currently belong to the cluster (Step 5). Next, each time series is reassigned to the cluster corresponding to the most similar aggregate dataset (Step 18) where similarity is defined in the PCA space using Eq. 4.3. The modified K-Means algorithm finally returns the metric patterns for the input instances.

5 Anomaly Detection

Given an instance from a machine, the proposed method first assigns it to the nearest context pattern and then assign it to a metric pattern within that context cluster. Thus, the method first estimates the context in which the machine was executing the set of Maps and Reduces, and then discovers the closest matching OS metrics behavior for that context. For an instance, if the right context cluster can be detected and if the instance is far away from any of the existing metric patterns for that cluster, then that instance can be labeled as an anomaly. Thus, an instance is anomalous if its metrics exhibit an unexpected behavior given its context.

The anomaly score of an instance should therefore depend on (1) how confidently the method can detect its context cluster, and (2) how far away the instance is from its nearest metrics cluster. If the distance between an instance and its nearest context cluster centroid is large, then the appropriate context of that instance cannot be detected. Hence, the normal metric patterns associated with this instance cannot be determined and so it cannot be marked as an anomaly.

Algorithm 1 Modified K-Means for Metric Pattern Discovery

```
Input: (1) Multi-variate time series $M_1, M_2, \ldots, M_N$ representing the metrics
       part of $N$ instances $I_1, I_2, \ldots, I_N$ respectively belonging to the context
       cluster $CP$, (2) Number of metrics clusters $K_2$.
Output: Metric clusters $MP_1, MP_2, \ldots, MP_{K_2}$.
 1: Randomly assign each dataset $M_1, M_2, \ldots, M_N$ to one of the
    $MP_1, MP_2, \ldots, MP_{K_2}$ clusters.
 2: while $MP_1, MP_2, \ldots, MP_{K_2}$ change do
 3:   for Each cluster $MP_i$ do
 4:     Let $\overline{M_1}, \overline{M_2}, \ldots, \overline{M_{|MP_i|}}$ be the elements of $MP_i$.
 5:     Aggregate dataset $D_i = [\overline{M_1}', \overline{M_2}', \ldots, \overline{M_{|MP_i|}}']'$.
 6:   end for
 7:   $MP_1 \leftarrow \phi, MP_2 \leftarrow \phi, \ldots, MP_{K_2} \leftarrow \phi$.
 8:   for Each multi-variate time series $M_i$ do
 9:     $max \leftarrow 0$.
10:     $nearestCluster \leftarrow 1$.
11:     for Each aggregate dataset $D_j$ do
12:       $currSim \leftarrow sim(M_i, D_j)$.
13:       if $currSim > max$ then
14:         $max \leftarrow currSim$.
15:         $nearestCluster \leftarrow j$.
16:       end if
17:     end for
18:     Assign $M_i$ to cluster $MP_{nearestCluster}$.
19:   end for
20: end while
```

Anomaly score of an instance $I = (C, M)$ is thus calculated as follows.

$$score(I) = 1 - sim(M, MP_M) \tag{5.4}$$

where MP_M is the nearest metrics cluster for the instance I. Top κ instances with highest anomaly scores can be marked as anomalies.

Anomaly Post-processing As a post-processing step, the instances which cannot be clustered into any context cluster properly, are removed from the set of anomalies. Specifically, if $dist(C, CP_C) \geq median(CP_C)$, the instance is removed from the set of anomalies where CP_C is the nearest context cluster for the instance I and $median(CP_C)$ is the median of the distances of any instance within cluster CP_C from cluster centroid CP_C.

Online Analysis Big companies like Yahoo! and Microsoft process similar jobs on a daily basis. Our system can build the unsupervised model of context patterns and metric patterns based on a training workload and then apply the model for new test instances. New test instances arriving every day can be cross-checked with the patterns learned using logs of past jobs. For the online case, rather than finding top κ anomalies, we can consider an instance as an anomaly if its distance from the nearest cluster centroid is greater than that of any instance in the training set. If the number of such discovered anomalies increases beyond κ for a batch of test instances, due to drift in the model, the model can be retrained on the recent data.

Our system can classify an instance as normal or anomalous as soon as the task corresponding to the instance finishes on the machine. Even for MapReduce jobs running for several hours, the individual map and reduce tasks are quite short (on the order of tens of seconds to a few minutes) and so the analysis is timely enough to be useful [24].

Iterative Computation The set of instances in the input dataset contains anomalies which can affect the metrics pattern discovery. Thus, once the anomalies have been discovered, the clustering of metrics time series can be performed again, ignoring the anomaly instances. This motivates an iterative approach where clustering of metrics time series and anomaly detection are performed within each iteration. The iterations terminate when the set of top κ anomalies detected in the last two consecutive iterations remains the same.

Summary We summarize our approach in Algorithm 2. The algorithm consists of two stages: extracting patterns and performing anomaly detection based on these patterns. Pattern extraction in turn consists of two phases: extraction of context patterns (Steps 1 and 2) followed by extraction of metric patterns (Steps 5 to 7). On termination, the algorithm returns the anomaly score for each instance.

Algorithm 2 Context-Aware Time Series Anomaly Detection

```
Input: Instances $I_1 = (C_1, M_1), I_2 = (C_2, M_2), \ldots, I_L = (C_L, M_L)$.
Output: Anomaly scores for each instance, $score$.
 1: Normalize the attributes in $C_1, C_2, \ldots, C_L$.
 2: $\{CP_1, CP_2, \ldots, CP_{K_1}\} \leftarrow K - Means(C_1, C_2, \ldots, C_L)$.
 3: Anomaly Set $A \leftarrow \phi$
 4: while $A$ does not change do
 5:     for $i = 1$ to $K_1$ do
 6:         Compute metric patterns for instances $\in (CP_i - A)$ using Algorithm 1.
 7:     end for
 8:     for $i = 1$ to $L$ do
 9:         Compute anomaly score for instance $I_i$ (Eq. 5.4).
10:     end for
11:     $A \leftarrow$ Set of instances with top $\kappa$ anomaly scores.
12: end while
```

6 Discussions

In this section, we will discuss various aspects of our algorithm and the general framework.

Time Complexity Let L be #instances, i_1 be #K-Means iterations, i_2 be #iterations for Algorithm 1, i_3 be #iterations of while loop from Step 4 to 12, K_1 be #context clusters, K_2 be #metric clusters, D_1 be #context variables, and k be #principal components used for PCA-based similarity computation. Then the overall time complexity of Algorithm 2 is $O(Li_3(i_1K_1D_1 + i_2K_2k^3))$ which is linear in the number of instances L. We omit details for lack of space.

Selecting Number of Clusters (K_1 and K_2) K-Means needs the number of clusters as an input. We use K_1 and K_2 as the input for clustering of context patterns and metric patterns. We start with #clusters as 1 and keep increasing them by 1. For each iteration, we compute the within cluster sum of squares of distances. We pick K_1 as the value where the percentage change in the within cluster sum of squares is small (knee of the curve).

Richer Context Besides the #Maps, #Reduces, and the task counters, Hadoop configuration settings like input format, output format, types of compression, file descriptor limits, can also be used as context variables. OS logs such as syslogs on Linux or Event Tracing for Windows [4] can also provide context information. We plan to study these as part of our future work.

7 Experiments

Evaluation of anomaly detection algorithms is quite difficult due to lack of ground truth. We generate multiple synthetic datasets by injecting anomalies, and evaluate anomaly detection accuracy of the proposed algorithms on the generated data. We also conduct case studies by applying the method to real data sets.

7.1 Baselines

We compare the proposed algorithm *Context-Aware (CA)* with two baseline methods: *Single Iteration (SI)* and *No Context (NC)*.

Single Iteration (SI) As described in Algorithm 2, *CA* performs metric pattern discovery and anomaly detection iteratively until the set of top κ anomalies do not change. *SI* is a simpler version of *CA*, which performs only one iteration. Thus the pattern discovery phase in *SI* suffers from the presence of anomalies. This baseline will help us evaluate the importance of ignoring the anomalies when computing the metric patterns.

No Context (NC) *CA* performs pattern discovery in a context-sensitive manner. A particular metric pattern may be very popular in a particular context but may never be exhibited in another context. Thus, performing context-aware anomaly detection should be better than a context insensitive approach. We test this claim by comparing *CA* with *NC*. Note that this baseline is similar to [14] which performs fault detection by using K-Means based clustering in the metrics space only.

7.2 Synthetic Datasets

We generate our synthetic datasets as follows. The context part of our dataset comes from real Hadoop runs, while the metrics part is generated synthetically. Our Hadoop cluster consists of 6 nodes – 1 master and 5 slaves. We run the standard Hadoop examples like Pi Estimator, Word Count, Sorter, Grep-Search (tasks run in parallel) and gather N instances. For each of the N instances, we obtain the context values (#Maps, #Reduces, task counters) from the Hadoop logs. We cluster the context part of these instances into 3 clusters. Fig. 3 shows the context clusters with respect to the different context variables. The values of context variables have been normalized as discussed in Section 4. Note that Cluster 1 contains instances with large number of Map tasks and hence display high values for Map counters. Cluster 3 contains instances with large number of Reduce tasks and hence display high values for Reduce counters. Cluster 2 contains instances with a few Map and a few Reduce tasks.

For each context cluster, we synthetically generate the metrics time series as follows. We first fix a number of metric clusters and randomly assign the instances to one of the metric clusters. We also fix the number of metrics. Next, we randomly generate a template matrix for each of the metric clusters. This template matrix serves as the basic multi-variate time series for that metric cluster. For a metric cluster, we generate the multi-variate time series for each instance by adding 10% uniform noise to the each of the entries of the template matrix for the cluster. We randomly select the duration of the time series between 20 and 50. Thus, we have for each instance, context values obtained from real logs and metric time series generated synthetically by adding random perturbations to a template matrix for the metric cluster of the instance.

Anomalies are injected in the metric time series as follows. First we set an anomaly factor Ψ and choose a random set of instances, R with $\frac{N \times \Psi}{2}$ instances. For each instance, we swap the time series with that of another randomly chosen instance. This corresponds to the situation of observing metric values which usually do not appear with the given context but may appear in some other context. For some of the instances rather than adding swap-anomalies, we replace the metric time series with random new matrices. This corresponds to the situation of observing metric values which usually never appear in any of the contexts.

Results on Synthetic Datasets We generate a variety of synthetic datasets capturing different experimental settings. For each setting, we perform 20 experiments and report the average values. We vary the number of instances as 500, 1000, 2000 and 5000. We also study the accuracy with respect to variation in the number of metrics (5, 10, 20) and variation in the number of metric clusters (4, 6, 10). We also vary the percentage of injected anomalies as 2%, 5% and 10%. We terminate the external while loop after a maximum of 5 iterations ($i_3 = 5$). For each algorithm, we show the accuracy with respect to matches in the set of detected anomalies and the set of injected anomalies, in Tables 1 and 2 (best accuracy in bold). Each of the accuracy values is obtained by averaging the accuracy for that experimental setting across 20 runs. Average standard deviations are 3.34% for *CA*, 7.06 % for *SI* and 4.58% for *NC*. As the table shows, the two versions of the proposed algorithm outperform the baseline algorithm (*NC*) for all of the settings by a wide margin. On an average across all experimental settings, *CA* is 28% better than *NC*. For small N, we observe that *CA* performs significantly better than *SI*.

Running Time The experiments were run on a Linux machine with 4 Intel Xeon CPUs with 2.67GHz each. The code was implemented in Java. Fig. 4 shows the average execution time spent in the metric pattern discovery stage of the *CA* algorithm for different number of instances and metrics. Note that the algorithm is linear in the number of instances. As number of machines in the cluster increase, number of instances increases proportionately. Thus, our algorithm scales well with number of machines. These times are averaged across multiple runs of the algorithm across different settings for number of metric clusters and across multiple runs for each setting. We also observed that the time spent in anomaly detection (~188ms on an average) is a small fraction of the time spent in clustering metric time series.

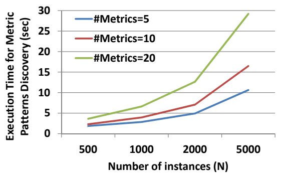

Figure 4: Average Running Time (sec) for Metrics Pattern Discovery Stage of *CA*

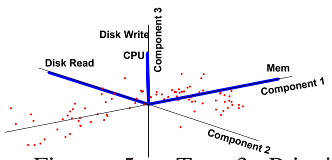

Figure 5: Top 3 Principal Component Representation for Metric Pattern 1 for Context Cluster 3

N	Ψ	$\#Metrics = 5$			$\#Metrics = 10$			$\#Metrics = 20$		
	(%)	CA	SI	NC	CA	SI	NC	CA	SI	NC
500	2	**0.8**	0.55	0.28	**0.741**	0.642	0.281	**0.86**	0.59	0.255
	5	**0.641**	0.641	0.302	**0.79**	0.627	0.332	**0.777**	0.69	0.345
	10	**0.661**	0.617	0.351	**0.742**	0.666	0.35	0.605	**0.633**	0.345
1000	2	**0.716**	0.603	0.255	**0.695**	0.683	0.314	**0.631**	0.608	0.272
	5	**0.706**	0.636	0.368	0.651	**0.654**	0.311	**0.691**	0.641	0.315
	10	0.655	**0.66**	0.359	0.637	**0.679**	0.345	0.659	**0.665**	0.322
2000	2	**0.64**	0.638	0.258	0.603	**0.643**	0.27	0.537	**0.583**	0.278
	5	0.578	**0.655**	0.289	**0.623**	0.623	0.319	0.586	**0.632**	0.321
	10	0.574	**0.6**	0.331	0.659	**0.715**	0.299	**0.626**	0.611	0.311
5000	2	0.664	**0.71**	0.361	0.666	**0.7**	0.362	0.636	**0.661**	0.362
	5	0.604	**0.62**	0.343	0.594	**0.626**	0.423	0.628	**0.669**	0.36
	10	0.661	**0.67**	0.414	0.654	**0.7**	0.41	**0.585**	0.555	0.418

Table 1: Synthetic Dataset Results (*CA*=The Proposed Algorithm, *SI*= Single Iteration, *NC*=No Context) for K_2=4

N	Ψ	$\#Metrics = 5$			$\#Metrics = 10$			$\#Metrics = 20$		
	(%)	CA	SI	NC	CA	SI	NC	CA	SI	NC
500	2	0.462	**0.482**	0.181	**0.71**	0.5	0.265	**0.682**	0.497	0.241
	5	**0.514**	0.505	0.279	**0.488**	0.465	0.3	**0.531**	0.461	0.298
	10	0.466	**0.485**	0.314	**0.527**	0.503	0.334	0.536	**0.549**	0.371
1000	2	0.406	**0.459**	0.223	**0.501**	0.493	0.276	**0.548**	0.548	0.225
	5	0.491	**0.543**	0.264	0.452	**0.516**	0.286	0.53	**0.541**	0.292
	10	0.509	**0.537**	0.327	0.535	**0.556**	0.331	0.476	**0.505**	0.316
2000	2	0.532	**0.564**	0.228	0.511	**0.565**	0.226	**0.576**	0.538	0.26
	5	**0.582**	0.579	0.296	0.503	**0.539**	0.286	0.453	**0.5**	0.237
	10	0.533	**0.55**	0.319	**0.508**	0.495	0.289	**0.499**	0.486	0.288
5000	2	0.506	**0.546**	0.26	0.531	**0.552**	0.352	0.532	**0.545**	0.309
	5	0.544	**0.585**	0.356	**0.543**	0.529	0.362	0.501	**0.522**	0.343
	10	0.557	**0.587**	0.41	0.496	**0.553**	0.398	0.517	**0.534**	0.374

Table 2: Synthetic Dataset Results (*CA*=The Proposed Algorithm, *SI*= Single Iteration, *NC*=No Context) for K_2=10

7.3 Real Datasets We test the effectiveness of our approach for two different real dataset scenarios. These scenarios capture the usual faults that occur in Hadoop instances: CPU Hog and Disk Hog. Such faults have been reported on Hadoop users' mailing list or on the Hadoop bug database. See [20] for details.

Dataset Generation The workload comprises of the two standard Hadoop examples – RandomWriter and Sort. We configure the RandomWriter to write 1GB of random data to HDFS. We run RandomWriter with 16 Maps and each Map generates 64 MB of data. We run sort with 16 Maps and 16 Reduces. Logs and metrics are copied to a single machine

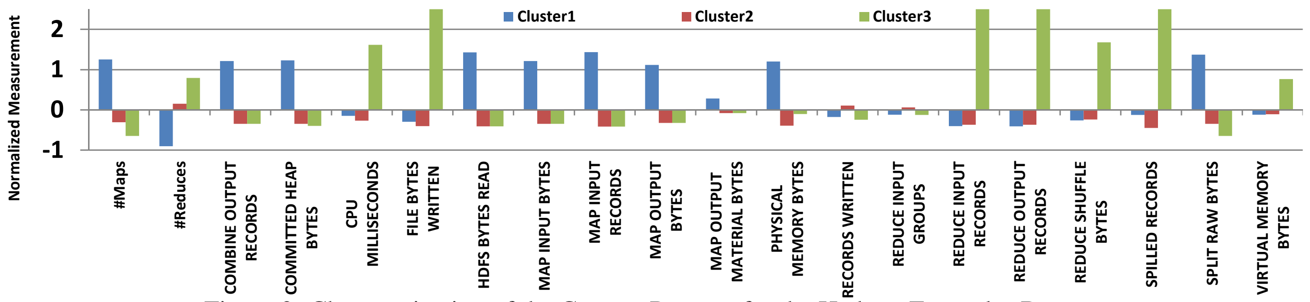

Figure 3: Characterization of the Context Patterns for the Hadoop Examples Dataset

where the processing is performed. Hadoop history logs and metric files are very small and hence it takes negligible time to process/copy them. Also, for our experiments, the cluster had only Hadoop processes running on the machines. However, in general for clusters with multiple applications running on the machines, one needs to extract metrics corresponding to the Hadoop processes only to avoid interference from other applications.

For lack of space, we do not show the characterization of the patterns obtained after clustering in the context space. But we present a few observations. Cluster 1 consists of a mix of Maps and Reduces, Cluster 2 has large number of Maps while Cluster 3 has large number of Reduces. Cluster 1 has a distinctly high number of HDFS bytes being written. Cluster 2 shows a large number of Map Output Records. Cluster 3 demonstrates a large activity in Reduce counters. Further, Fig. 5 shows the various metric variables in the space of the top three principal components for the first metric pattern in the third context cluster. Each metric variable is represented by a vector; direction and length of the vector indicates how each variable contributes to the three principal components in the plot. E.g., Memory and Disk read operations have strong influences on the first and second principal components respectively. The red dots are the instances belonging to the metric pattern as plotted in the top 3 principal component space.

Results on Real Datasets We run the workload 10 times for each of the CPU hog and the Disk hog experiments. For the first run in both the experiments, we inject an anomaly (CPU hog or disk hog) on one of the machines. CPU hog is simulated by running a infinite loop program while disk hog is simulated by writing multiple files to the disk in parallel. Overall, we have 134 and 121 instances for the disk hog and CPU hog datasets. There are 7 and 4 anomalous instances from the machine where the hog was injected for the disk hog and the CPU hog datasets respectively.

Fig. 6 shows a plot of the anomaly scores of various instances. The X axis shows the instance number and the Y axis shows the anomaly score. The steep decrease in the anomaly score both for the CPU Hog and the Disk Hog case clearly shows that our algorithm computes a discriminative score for the anomalies while giving distinctly lower anomaly scores to the normal instances.

As expected we observed that the instances on the machine where the hog is injected has an abnormal metrics behavior. E.g., in case of CPU hog, the CPU utilization of the machine is much higher than that expected for the context. Similarly, in case of disk hog, the number of disk write operations increases by a large amount. Out of the 7 anomalies for the disk hog, 4 of them are present in the top 5 and all 7 get detected in the top 10. In the case of CPU hog, 3 get detected in the top 5 and all 4 in the top 10. Thus, our methodology is quite effective in detecting anomalies, specifically in localizing the anomalies to the machine level and also with respect to the time duration.

For the CPU hog, we did some analysis of the false positives that were reported. One of the false positives appeared to have some reasonable amount of CPU utilization (about 37% which was higher compared to the other members of its metric cluster (average=5.08%, std dev=1.37%)). We believe that this was because of some other process running on the machine at that time. Another false positive showed high number of disk reads (71 compared to an average 28) and low disk write operations (13 compared to an average of 17) for the same metrics cluster.

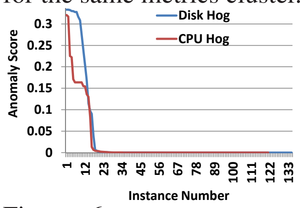

Figure 6: Anomaly Scores for CPU Hog and Disk Hog Datasets (Both the series have been sorted with respect to anomaly scores)

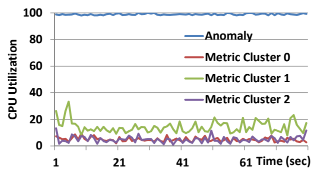

Figure 7: CPU Utilization of an Anomaly Versus Average CPU Utilization of Normal Instances for Metric Patterns within the same Context

We show one of the anomalies for the CPU Hog case in Fig. 7 for a period of 80 seconds. Note that the anomalous instance has a very high CPU Utilization. The other curves show the average CPU Utilization of the instances belonging to the metric patterns within the same context. The graph shows that in the particular context, the expected CPU utilization is quite low and shows a fluctuating trend, while the anomaly shows a very high CPU utilization which is almost flat. Hence, the proposed algorithm marks this instance as an anomaly.

8 Related Work

Our work is related to previous work on anomaly detection for distributed systems using logs only and using performance metrics only.

Anomaly/Fault Detection using System Logs Recently, there has been work that uses system logs for anomaly detection ([11, 12, 16, 17]). Researchers have studied logs using Hidden Markov Models [23], state machines and control and data flow models (SALSA) [19] for fault identification. Besides system logs, message logs across components have been exploited in fault detection systems like Pip [15] and Pinpoint [2]. While log analysis-based systems exploit the event and context information from logs, they fail to exploit the rich information contained in the operating system performance metrics.

Anomaly/Fault Detection using OS Performance Metrics Tiresias performs anomaly detection by exploring individual performance metrics by putting thresholds on each metric separately [22]. Clustering on OS metrics has been used for fault detection (Ganesha [14]) and prediction ([6, 7], ALERT [21]). Time series models and probabilistic correlations among monitored metrics have also been exploited for the task [8, 10]. Recent works [13, 20] discuss about combining Hadoop logs and OS metrics for better fault detection. But our work is different from them in several ways: 1. Though they point out that OS metrics and Hadoop logs are complementary, they do not provide any principled methodology to tightly integrate the two pieces. 2. They provide a supervised approach while our approach is completely unsupervised and requires no prior labeling. 3. Their study is focused on jobs that take exceptionally longer, while we focus on out-of-context metric anomalies. Our method helps in finding time durations on any machine for which the metrics were inconsistent with respect to the context in which they were observed.

9 Conclusions

We motivated the need of combining system log information and OS performance metric observations for effective anomaly detection for MapReduce systems. The anomalies were detected based on the context patterns and metric patterns. The context patterns were derived by clustering in the space of context variables associated with the instances. Metric patterns were discovered for every context cluster using a PCA-based similarity measure for multi-variate time series and a modified K-Means algorithm. Metric pattern discovery and anomaly detection was performed iteratively to mutually improve the quality of results. Using synthetic datasets and real Hadoop runs, we showed the effectiveness of the proposed approach in finding interesting anomalies. The approach could be extended to study anomalies related to *change* in OS performance metrics with *transitions* in the context captured by the system logs.

References

[1] Amazon Outage. http://aws.amazon.com/message/65648/.

[2] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. A. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *ICDSN*, pages 595–604, 2002.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *CACM*, 51(1):107–113, Jan 2008.

[4] Improving Debugging And Performance Tuning with ETW. http://msdn.microsoft.com/en-us/magazine/cc163437.aspx.

[5] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From Dirt to Shovels: Fully Automatic Tool Generation from Ad hoc Data. In *POPL*, pages 421–434, 2008.

[6] X. Gu, S. Papadimitriou, P. S. Yu, and S.-P. Chang. Toward Predictive Failure Management for Distributed Stream Processing Systems. In *ICDCS*, pages 825–832, 2008.

[7] X. Gu and H. Wang. Online Anomaly Prediction for Robust Cluster Systems. In *ICDE*, pages 1000–1011, 2009.

[8] Z. Guo, G. Jiang, H. Chen, and K. Yoshihira. Tracking Probabilistic Correlation of Monitoring Data for Fault Detection in Complex Systems. In *ICDSN*, pages 259–268, 2006.

[9] Hadoop Logs. http://tinyurl.com/hadoop-logs.

[10] G. Jiang, H. Chen, and K. Yoshihira. Modeling and Tracking of Transaction Flow Dynamics for Fault Detection in Complex Systems. *TDSC*, 3:312–326, Oct 2006.

[11] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo. Failure Prediction in IBM BlueGene/L Event Logs. In *ICDM*, pages 583–588, 2007.

[12] J. F. Murray, G. F. Hughes, and K. Kreutz-Delgado. Machine Learning Methods for Predicting Failures in Hard Drives: A Multiple-Instance Application. *JMLR*, 6:783–816, Dec 2005.

[13] X. Pan, J. Tan, S. Kalvulya, R. Gandhi, and P. Narasimhan. Blind Men and the Elephant: Piecing Together Hadoop for Diagnosis. In *ISSRE*, 2009.

[14] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Ganesha: BlackBox Diagnosis of MapReduce Systems. *SIGMETRICS Perform. Eval. Rev.*, 37(3):8–13, Jan 2010.

[15] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI*, pages 9–22, 2006.

[16] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical Event Prediction for Proactive Management in Large-scale Computer Clusters. In *KDD*, pages 426–435, 2003.

[17] B. Schroeder and G. A. Gibson. Disk Failures in Real World: What does MTTF of 1,000,000 Hours mean to you? In *FAST*, 2007.

[18] Sisyphus – A Log Data Mining Toolkit. http://www.cs.sandia.gov/~jrstear/sisyphus/.

[19] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. SALSA: Analyzing Logs as State Machines. In *WASL*, pages 6–13, 2008.

[20] J. Tan, X. Pan, E. Marinelli, S. Kavulya, R. Gandhi, and P. Narasimhan. Kahuna: Problem Diagnosis for Mapreduce-based Cloud Computing Environments. In *NOMS*, pages 112–119, 2010.

[21] Y. Tan, X. Gu, and H. Wang. Adaptive System Anomaly Prediction for Large-Scale Hosting Infrastructures. In *PODC*, pages 173–182, 2010.

[22] A. W. Williams, S. M. Pertet, and P. Narasimhan. Tiresias: Black-box Failure Prediction in Distributed Systems. *IPDPS*, pages 1–8, 2007.

[23] K. Yamanishi and Y. Maruyama. Dynamic Syslog Mining for Network Failure Monitoring. In *KDD*, pages 499–508, 2005.

[24] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys*, pages 265–278, 2010.