2017 IEEE International Conference on Web Services (ICWS)

Log-based Abnormal Task Detection and Root Cause Analysis for Spark

Siyang Lu*, BingBing Rao*, Xiang Wei,*, Byungchul Tak†, Long Wang‡, Liqiang Wang*
*Dept. of Computer Science, University of Central Florida, Orlando, FL, USA
†Dept. of Computer Science and Engineering, Kyungpook National University, Republic of Korea
‡IBM TJ Watson Research Center, Yorktown Heights, NY, USA
Email: {siyang,robin.rao,xiangwei,liqiang.wang}@knights.ucf.edu, bctak@knu.ac.kr, wanglo@us.ibm.com

***Abstract*—Application delays caused by abnormal tasks are common problems in big data computing frameworks. An abnormal task in Spark, which may run slowly without error or warning logs, not only reduces its resident node's performance, but also affects other nodes' efficiency.**

Spark log files report neither root causes of abnormal tasks, nor where and when abnormal scenarios happen. Although Spark provides a "speculation" mechanism to detect straggler tasks, it can only detect tailed stragglers in each stage. Since the root causes of abnormal happening are complicated, there are no effective ways to detect root causes.

This paper proposes an approach to detect abnormality and analyzes root causes using Spark log files. Unlike common online monitoring or analysis tools, our approach is a pure off-line method that can analyze abnormality accurately. Our approach consists of four steps. First, a parser preprocesses raw log files to generate structured log data. Second, in each stage of Spark application, we choose features related to execution time and data locality of each task, as well as memory usage and garbage collection of each node. Third, based on the selected features, we detect where and when abnormalities happen. Finally, we analyze the problems using weighted factors to decide the probability of root causes. In this paper, we consider four potential root causes of abnormalities, which include CPU, memory, network, and disk. The proposed method has been tested on real-world Spark benchmarks. To simulate various scenario of root causes, we conducted interference injections related to CPU, memory, network, and Disk. Our experimental results show that the proposed approach is accurate on detecting abnormal tasks as well as finding the root causes.

***Keywords*-Spark; Log Analysis; Abnormal Task; Root Cause;**

I. Introduction

With rapid growth of data size and diversification of workload types, big data computing platforms increasingly play more important role for solving real-world problems [8, 9]. Several outstanding frameworks are in active use today including Hadoop [1], Spark [2], Storm and Flink. Among them, the Apache Spark has arguably seen the widest adoption. It supports a fast and general programming model for large-scale data processing, in which Resilient Distributed Dataset (RDD) [18] are used to describe the input and intermediate data generated during the computation stages. RDDs are divided into different blocks, called partitions, with almost equal size among different compute nodes. Apache Spark uses pipeline to distribute various operations that work on a single partition of RDD. In order to serialize the execution of tasks, Spark introduces stage. All tasks in the same stage execute the same operation in parallel.

Compute nodes may suffer from a huge of interferences from software (such as operating systems or other processes) or hardware, which leads to abnormal problems. For instance, we name a tasks an abnormal task or straggler when it encounters significant delay in comparison with other tasks in the same stage. In Spark, there is a mechanism named speculation to detect this scenario, in which such slow tasks will be re-submitted to another worker. Spark performs speculative execution of tasks till a specified fraction (defined by `spark.speculation.quantile`, which is 75% by default) of tasks must be complete, then it checks whether or not the running tasks run slower than the median of all successfully completed tasks in a stage. A task is a straggler if its current execution time is slower than the median by a given ratio (which is defined by `speculation.multiplier`, 1.5x by default). In this paper, we propose a different approach compared with Spark Speculation. In our method, we consider whole Spark stages and abnormal tasks happening in any life span could be detected. In addition, Spark's report could be inaccurate because Spark uses only fixed amount of finished task durations to speculate the unfinished tasks.

When abnormal tasks (including stragglers) happen, the performance of Spark applications could be degraded. However, it is very difficult for users to detect and analyze the root causes. First, Spark log files are tedious and difficult to read, and there is no straight-forward way to tell whether abnormal tasks happen or not, even through stragglers can be reported when speculation is enabled. Second, when an abnormal scenario happen, there is few information about the error in log files so that it is difficult for users to see the concreted reasons that lead to the straggler problem. Third, even online tools can monitor the usage and status of system resource such as CPU, memory, disk, and network, these tools do not directly cooperate with Spark, and users still need many efforts to scrutinize root causes based on their reporting. In addition, these monitoring tools usually carry overhead and may slow down Spark's performance. Abnormal tasks could be caused by many reasons, where most of them are resource contentions [5] by CPU, memory, disk, and network. Our motivation is to help users find root causes of abnormal tasks by analyzing only Spark logs.

In this paper, we propose an off-line approach to detect abnormal tasks and analyze the root causes. Our method is

DOI 10.1109/ICWS.2017.135

based on a statistical spatial-temporal analysis for Spark logs, which consists of Spark execution logs and Spark garbage collection logs. There are four steps to detect the root causes. (1) We parse Spark log files according to key words, such as task duration, data location, time stamp, task finish time, and generate a structured log data file. This step will eliminate all irrelevant messages and values. (2) We extract the related feature set directly from structured log file based on our experimental study. (3) We detect abnormal tasks from the log data by analyzing all relevant features. Specifically, we calculate the mean and standard deviation of all tasks in each stage, then determine abnormal tasks for each stage. (4) We generate factor combination criteria for each potential root cause based on analyzing their weighted factor in training datasets. Thus, our approach can effectively determine the proper root causes for given abnormal tasks.

The major contributions of this paper are as follows:

- The approach can accurately locate where and when abnormal tasks happen based on analyzing only Spark logs.
- Our offline approach can detect root causes of abnormal tasks according to Spark logs without any monitoring data, thus it does not have any monitoring overhead.
- It provides an easy way for users to deeply understand Spark logs and tune Spark performance.
- It gives an reasonable probability result for root cause analysis.

II. Spark Architecture

A. Background

Apache Spark is a fast and general engine for large-scale data processing. In order to achieve scalability and fault tolerance, Spark introduces an abstraction called resilient distributed dataset (RDD) , which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. When an application is submitted to Spark, the cluster manager will allocate compute resource according to the requirement of the application, then Spark scheduler distributes tasks to executors, and tasks will be executed in parallel. During this process, Spark driver node will monitor the status of executors and collect the tasks results from the worker nodes. In order to parallelize a job, Spark scheduler divides an application into a series of stages based on data dependence. In each stage, all tasks do not have data dependence and execute the same function.

B. The Framework of Spark Logging

Spark driver and executors record the status of executor and collection of execution information about tasks, stages, and job, which are the source of Spark logs.

Each Spark executor contains two log files, Spark execution log which record by log4j [7] and Spark garbage collection (GC) log, which are the outputs by `stderr` and `stdout`, respectively. Each of worker nodes and master nodes has its own log files. When an application is finished, we collect all Spark log files and aggregate them into an execution log and a GC log.

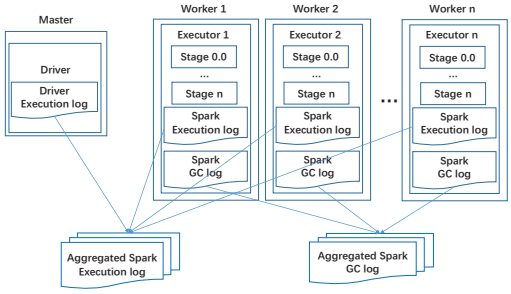

Figure 1. Spark workflow and log files.

III. Abnormal Task Detection and Root Cause Analysis

Spark log does not show abnormal tasks directly, thus users cannot locate abnormal tasks by simply searching keywords. This motivates us to design an automatic approach to help users detect the abnormals and analyze the root causes.

A. Approach Overview

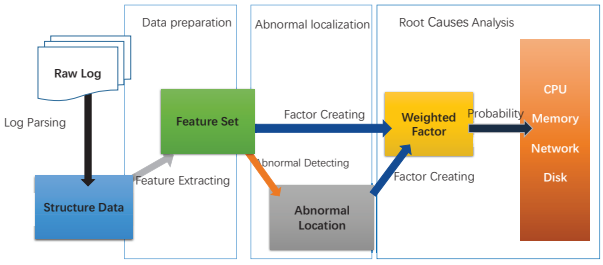

Figure 2. Workflow of abnormal detection and root cause analysis.

The workflow of our approach for abnormal detection and root cause analysis is shown in Figure 2.

1) *Log preprocessing*: We collect all Spark logs, including execution logs and Spark GC logs, from the driver node and all worker nodes. Then, we eliminate noisy data and reformat logs into more structured data.
2) *Feature extraction*: Based on Spark scheduling and potential abnormal task happening conditions, we screen execution-related, memory-related, CPU-related data to generate two matrices: execution log matrix and GC matrix. The details are illustrated in Section III-B.
3) *Abnormal detection*: We implement a statistical analysis approach based on the analysis of four kinds of features, including task duration, timestamps, GC time, and other task-related features, to determine the degree of abnormal tasks and locate their happening. The details are discussed in Section III-C.

4) *Root cause detection*: Instead of qualitatively deciding the exact root causes that lead to the abnormals, we quantitatively measure the degree of abnormals by a weighted combination of certain specific cause-related factors. The details are sh in Section III-E.

B. Feature Execution

According to Spark scheduling strategy, we define and classify all features into three categories, namely, execution-related, memory-related, and CPU-related, which are shown in Table I. For example, the execution-related features can be extracted from Spark execution logs, including task ID, task duration, task finished time, task started time, stage ID, and job's duration. Spark GC log records all JVM memory usage, from which we can extract memory-related features such as heap usage, young space usage, as well as features related system CPU usage such as system time and user time. These feature sets extracted from Spark execution log and GC log are shown in Table I.

C. Abnormal Detection

Adopting Spark speculation may bring false negatives in the process of abnormal detection. Hence, we provide a more robust approach to locate where stragglers happen and how long they take. We will also consider about special scenarios, for example, different stages are executed in sequence or in parallel.

One basic justification of abnormal tasks is that the running time of abnormal tasks is relatively longer than the normal ones. [5] uses "mean" and "median" to decide the threshold. However, in order to seek a more reasonable anomaly detection strategy, we consider not only the mean or median task running time, but also the distribution of the whole data, namely the standard deviation. In this way, we can get a macro-awareness on the task's execution time, and then based on the distribution of data, a more reasonable threshold can be set to differentiate abnormals from the normal ones. The abnormal detection mainly includes the following two issues.

1. *Comparing task running time on different nodes*

We compare task execution time on different nodes in the same stage. Let $T\_task_{i,j,k}$ denote the execution time of task k in stage i on node j. Let avg_stage_i denote the average execution time of all tasks, which belong to different nodes but in the same stage i.

$$avg\_stage_i = \frac{1}{\sum_{j=1}^{J} K_j}(\sum_{j=1}^{J}\sum_{k=1}^{K_j} T\_task_{i,j,k}) \quad (1)$$

Work flow where J and K_j are the total number of nodes and the number of tasks in node j, respectively.

Similarly, the standard deviation of task execution in stage j of all nodes is denoted as std_stage_i. Abnormal tasks are determined by the following conditions:

$$task_k \begin{cases} abnormal & T\_task_k > avg\_stage_i + k * std\_stage_i \\ normal & \text{otherwise} \end{cases} \quad (2)$$

where k is a factor that controls the threshold for abnormal detection. In this paper, we set it to 1.5 by default for fair compare with Spark provided speculator.

Figure 3 (c) shows abnormal detection process in Word-count under CPU interference. Figure 3 (a) and (b) are two stages inside the whole application. Moreover, inside each of the stage, purple-dot line is the abnormal threshold determined by Eq. (1), and the black dot-line indicates the threshold calculated by Spark speculation. For all tasks within a certain stage, the execution time above that threshold are detected as abnormals; otherwise, they are normals. Figure 3 (d) displays memory occupation along the execution of its corresponding working stages.

2. *Locating abnormal happening*

After all tasks are properly classified into "normal" and "abnormal", the whole time line are labeled as a vector with binary number (*i.e.*, 0 or 1, which denote normal and abnormal, respectively). To smooth the outliers (for example, 1 appears in many continuous 0) inside each vector, which could be an abrupt change but not consistent abnormal base, we then empirically set a sliding window with size of 5 to flit this vector. If the sum of numbers inside the window is larger than 2, the number in the center of the window will be set to 1, otherwise 0.

The next step is to locate the start and end time of this abnormal task. Note that, as Spark logs record the task finishing time but not the start time, so we locate the abnormal task's start time as the recorded task finishing time minus its execution time. Moreover, for abnormal detection in each stage, the tasks are classified into two sets. One is for the initial tasks whose start time stamps are the begin of each stage, as these tasks often have more overhead (such as loading code and Java jar packets), and the execution time usually operates much longer than its followings. Another set consists of the rest tasks. Our experiments show that this classification inside each stage can lead to a much accurate abnormal threshold. In this way, our abnormal detection method can not only detect whether abnormals happen, but also locate where and when they happen.

D. Factors Used for Root Causes Analysis

After abnormals are located, we analyze their root causes inside that certain area. For different root causes, we use different features in Spark log matrix and GC matrix to determine criteria to decide the root causes. Specifically, for each root cause analysis, we use the combination of weighted factors to define the degree of probability of each root cause. In all normal cases the factor should equal to 1, and if an abnormal tends to any root cause, the factor will become much bigger than 1. The factors are denoted as a,b,c,d,e,f,g for weight calculating. All of the indexes which are used in our factors' definition are listed here:

Table I
SPARK EXECUTION LOG MATRIX & GC LOG MATRIX

Related	Name	Meaning
Execution-Related	Time stamp	Event happening time
	Task execution time	A task's running duration time
	Stage ID	The ID number of each stage
	Host ID	The Node ID number
	Executor ID	The ID number of each executor running in per-worker
	Task ID	The unique ID number of each task
	Job execution time	A job running duration time (an application may contains many jobs)
	Stage execution time	A stage running duration time
	Application execution time	An application running duration time (after submitted)
	Data require location	The location of task required data
Memory-Related	Heap space	Total Heap memory usage
	Before GC Young space	Young space memory usage before clearing Young space
	After Young GC space	Young space memory usage after clearing Young space
	Before Heap GC space	Total Heap memory usage before GC
	After Heap GC space	Total Heap memory usage after GC
	Full GC time	Full GC execution time
	GC time	Minor GC execution time
	GC category	The time spend on one full GC operation
CPU-Related	user time	CPU time spent outside kernel execution
	sys time	CPU time spent insides kernel execution
	real time	Total elapsed time of the GC operation

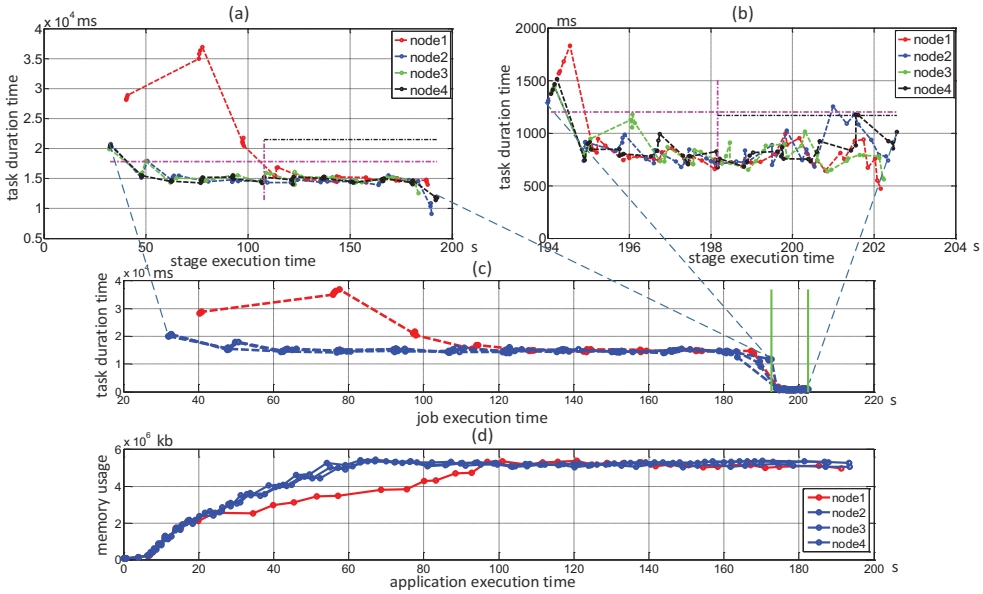

Figure 3. Abnormal detection under CPU interference in the experiment of WordCount: (a) Abnormal detection result in Stage-1. (b) Abnormal detection result in Stage-2. (c) Spark execution log features for abnormal detection in the whole execution. (d) Spark GC log features for abnormal detection in the whole execution.

j,J,i,I,k,K,n,N, inside which, j indicates the jth node, J is set of nodes; i is the index of stage, I is a set of stages; k denotes a task, K is a task set; n stands for a GC record, N is GC records set. All factors used to determine root causes are listed as below.

1. *Degree of Abnormal Ratio (DAR)*

Eq. (3) indicates the degree of abnormal ratio in a certain stage, as defined in Eq. (3).

$$a = \frac{k_{j'}}{\frac{1}{J-1}((\sum_{j=1}^{J} k_j) - k_{j'})} \tag{3}$$

where k_j indicates the number of tasks in node j, and J is the total number of nodes in the cluster. Here, we assume that node j' is abnormal.

2. *Degree of Abnormal Duration (DAD)*

The average task running time should also be considered, as the abnormal nodes often record longer task running time.

$$b = \frac{avg\_node_{j'}}{\frac{1}{J-1}((\sum_{j=1}^{J} avg\_node_j) - avg\_node_{j'})} \quad (4)$$

where avg_node_j is defined as:

$$avg\_node_j = \frac{1}{K_j}(\sum_{k=1}^{K_j} T\_task_{i,j,k}) \quad (5)$$

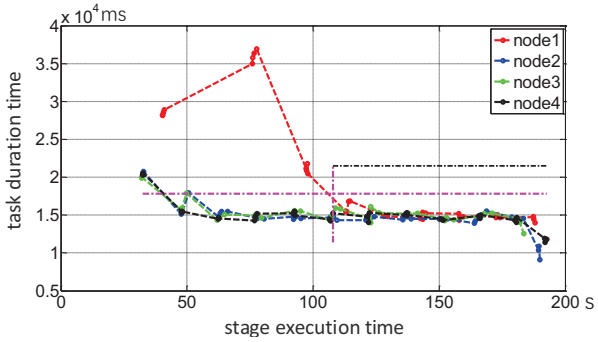

Figure 4. CPU interference injected after 20s application was submitted, and continuously impacts 80s

3. *Degree of CPU Occupation (DCO)*

This factor c shown in Eq. (6) is used for expressing the ratio between the wall-clock time and the real CPU time. In the normal multiple-core environment, "realTime" is often less than "sysTime+UserTime", because GC is usually invoked in multi-threading way. However, if the "realTime" is bigger than "sysTime+UserTime", it may indicate that the system is very busy. We choose a Max value across nodes as the final factor.

$$c = \max_{j \in J}(avg_{j \in J}(\frac{realTime_{i,j}}{sysTime_{i,j} + userTime_{i,j}})) \quad (6)$$

4. *Memory Changing Rate (MCR)*

Eq. (7) indicates the gradient of GC curve. Under CPU, memory, and Disk interference, the interfered node's GC curve will change slower than the normal nodes' GC curve, as shown in Figure 5. k_stable and k_end are the gradients of the connected lines between start position (the corresponding memory usage at abnormal starting time) to the stable memory usage position and the start position to the abnormal memory end position (both the abnormal start and end time are obtained in the previous section) respectively. The reason we conduct this equation is that the interfered node uses less memory than normal nodes under interference. In this way, we use the maximum value of k_stable in the whole cluster (k_stable of normal node) to divide the minimums k_end in the whole cluster (interfered node) to get the value of this factor.

$$d = \frac{\max_{j \in J}(k\_stable_j)}{\min_{j' \in J}(k\_end_{j'})} \quad (7)$$

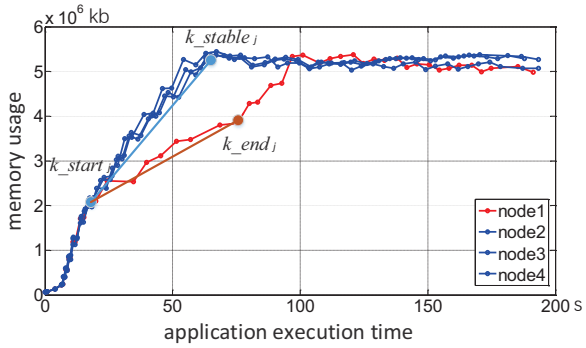

Figure 5. CPU interference is injected after WordCount has run for 30s, and continuously impacts 120s.

5. *Degree of Task Delay (DTD)*

For network interference, the task execution time will be affected when data transmission is delayed. Moreover, a Spark node often accesses data from other nodes, which leads to network interference propagation. Based on these facts, if network interference happens inside the cluster, the whole nodes will be affected, as shown in Figure 6, which is the location of our detected interference. Let a be a factor that describes the degree of interference.

$$e = \exp(J * \prod_{j=1}^{J} abn\_prob_j) \quad (8)$$

Where abn_prob_j indicates the ratio of abnormal that we detect for each node j inside that area. The reason that we use the product of abnormal ratio other than the sum of them is that only when all nodes are with a portion of abnormal should we identify them with a potential of network interference, or if sum is used, we cannot detect this joint probability. Meanwhile, the exponential is to make sure that the final factor e is no less than 1. In this way, the phenomenon of error propagation will be detected and quantified, which can only be shown in the cluster with network interference injection.

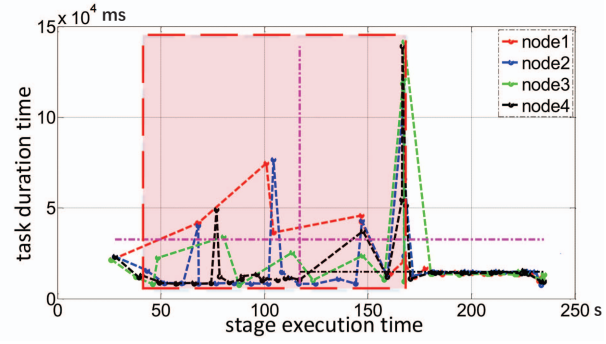

Figure 6. Network interference is injected after WordCount has been executed for 30s, and continuously impacts for 160s.

6. *Degree of Memory Changing (DMC)*

As network bandwidth is limited or the network speed slows down, when one node get affected by that interference, the task will wait for their data transformation from other nodes. Hence, CPU will wait, and data transfer rate becomes low. As shown in Figure 7.

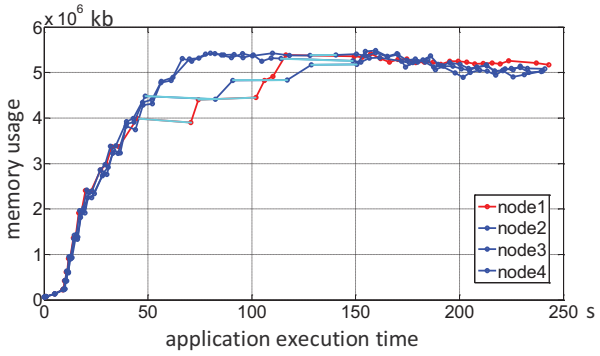

Figure 7. Network interference is injected after WordCount has been executed for 30s, and continuously impacts for 120s.

$$f = \frac{\underset{j\in J}{max}\{\underset{n\in N}{max}[e^{-|m_{j,n}|} * (x_{j,n} - x_{j,n-1})]\}}{\underset{j\in J}{min}\{\underset{n\in N}{max}[e^{-|m_{j,n}|} * (x_{j,n} - x_{j,n-1})]\}} \quad (9)$$
$$\text{where}, m_{j,n} = \frac{y_{j,n} - y_{j,n-1}}{x_{j,n} - x_{j,n-1}}$$

where $m_{j,n}$ indicates the gradient of memory changing in nth task on node j. Eq. (4) is to find the longest horizontal line that presents the conditions under which tasks' progress become tardy (*e.g.*, CPU is relatively idle and memory is kept the same). We first calculate the max value of gradient for each GC point, denoted as m. To identify the longest horizontal line in each node, we make a trade-off between its gradient and the corresponding horizontal length. To determine a relative value that presents the degree of abnormal out of normal, we finally compare the max and min among nodes with their max "horizontal factor" ($e^{-|m_{j,n}|} * (x_{j,n} - x_{j,n})$), where e is to ensure that the whole factor of b not less than 1).

7. *Degree of Loading Delay (DLD)*

Considering that the initial task at the beginning of each stage always have a higher overhead to load data blocks compared to the rest tasks. To only focus on that area, the factor of g is proposed to measure its abnormality. Similar to factor f, instead of taking all the tasks inside the detected stage into consideration, here, the first task of each node is used to replace the "avg_node_j"in Eq. (4). Formally, the equation is modified as Eq. (10) shows.

$$g = \frac{T\_task_{i,j',1}}{\underset{j\in J}{avg}(T\_task_{i,j,1})}, where\ j' \notin J \quad (10)$$

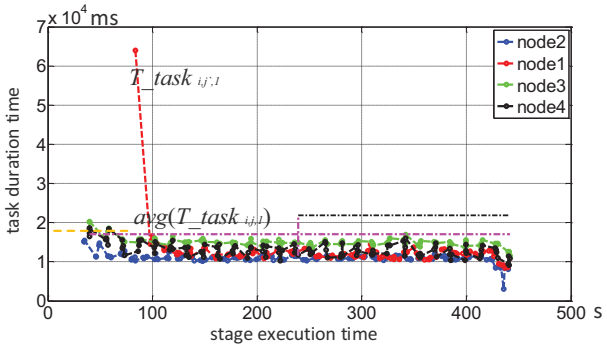

Figure 8. Disk interference is injected after WordCount has run for 20s, and continuously impacts 80s

E. Root Causes Analysis

As shown in Table II, each root cause is determined by a combination of factors with specific weights.

The nodes with CPU interference often have a relatively lower computation capacity, which leads to less tasks allocated and longer execution time for tasks on it. Factors a and b are used to test if the interference is CPU or not, because CPU interference can reduce the number of scheduled tasks and increase the abnormal tasks' execution time. Factor c indicates the degree of CPU occupation, and CPU interference will slow down of the performance compared to normal cases. Factor d is used to measure memory changing rate, because CPU interference may lead memory change to become slowly than other regular nodes.

For the network-related interferences, because of its propagation, the original interfered node will often recover earlier. So our approach is to detect the first recovered node as the initial network-interfered node, and the degree b quantitatively describes the interference. When network interference occurs, tasks are usually waiting for data delivery (factor e), the memory monitored by GC log f is usually unchanged.

For the memory-related interferences, when memory interference is injected into the cluster, we can even detect a relatively lower CPU usage than other normal nodes. Considering this, the task numbers (factor a) and task duration (factor b) are also added to determine such root causes with certain weights. Moreover, the memory interference will impact memory usage, and the factor d should be considered for this root cause detection.

To determine disk interferences, we introduce the factor g to measure the degree of disk interference. The task set scheduled at the beginning of each stage could be affected by disk I/O. Therefore, these initial tasks on disk I/O interfered nodes behave differently from other nodes' initial tasks beginning tasks (factor g), CPU will become busy, and memory usage is different with other nodes'. Therefore, The memory changing rate (factor c) and CPU Occupation (factor d) are also used to determine such root causes.

After deciding the combination of factors for each root cause, we give them weights to determine root causes

Table II
FACTOR FOR EACH ROOT CAUSES

Factor type		CPU	Mem	Network	Disk
a	DAR	√	√		
b	DAD	√	√	√	
c	DCO	√			√
d	MCR	√	√		√
e	DTD			√	
f	DMC			√	
g	DLD				√

accurately. Here, all weights are between 0 and 1, and the sum of them for each root cause is 1. To decide the values of weights, we use classical liner regression on training sets that we obtained from experiments on WordCount, Kmeans, and PageRank, which are discussed in more details in Section IV.

$$\begin{aligned} CPU &= 0.3 * a + 0.3 * b + 0.2 * c + 0.2 * d \\ Memory &= 0.25 * a + 0.25 * b + 0.5 * d \\ Network &= 0.1 * b + 0.4 * e + 0.5 * f \\ Disk &= 0.2 * c + 0.2 * d + 0.6 * g \end{aligned} \tag{11}$$

Then, Eq. (12) is proposed to calculate the final probability that the abnormal belongs to each of the root causes.

$$probability = 1 - \frac{1}{factor} \tag{12}$$

IV. EXPERIMENTS

In this section, we present the experimental results on our abnormal detection and the root cause analysis in three Spark applications, *i.e.*, WordCount, Kmeans, and PageRank which are provided by sparkbench [10].

A. Experimental Setup

To evaluate the performance of our proposed approach, we build an Apache Spark Standalone Cluster with four compute nodes, in which each compute node has a hardware configuration with Intel Xeon CPU E5-2620 v3 @ 2.40GHz, 16GB main memory, 1 Gbps Ethernet, and CentOS 6.6 with kernel 2.6. Apache Spark is v2.0.2.

B. Interference Injection

1) *CPU*: We spawn a bunch of processes to compete with Apache Spark jobs for computing resources, which triggers straggler problems in consequence of limited CPU resource.
2) *Memory*: We run a program that requests a significant amount of memory to compete with Apache Spark jobs. Thus, Garbage Collection will be frequently invoked to reclaim free space.
3) *Disk*: We simulate disk I/O contention using "dd" command to conduct massive disk I/O operations to compete with Apache Spark jobs.
4) *Network*: We simulate a scenario where network latency has a great impact on Spark. Specifically, we use "tc" command to limit bandwidth between two computing nodes.

C. Experimental Result Analysis & Evaluation

We conduct experiments on three benchmarks, WordCount in Spark package, Kmeans and Page Rank in SparkBench [10]. We run each of the benchmarks 20 times with simulated interference injection.

Table III summarizes the probability results of our root cause detection approach. For the first step, totally 320 abnormal cases are created, out of which 38 are detected as normal (accuracy: 88.125%). Among these mis-classified cases, 29 are from memory fault injection and the rest 9 are from disk IO. Meanwhile, additional 60 normal cases are also put into our approach for root cause detection, and no one is reported as abnormal. We also check the normal cases' abnormal factors to demonstrate the effectiveness of our approach. In all three benchmarks, the impact of CPU interference is significant, and tasks under CPU interference can be detected as abnormal with high probability. For memory interference, its probability is not significant because memory interference has less direct effect on Spark tasks, not like root causes. Injecting significant memory interference into one node will cause the whole application crash because the executers of Spark will fail if without enough memory. For network interference, the results show that the proposed approach gives a high probability. Lastly, disk interference shows a high probability in disk root causes. Worth mentioning here, for all different root causes, the detected probability of CPU are always high, because all root causes will eventually affect the efficiency of CPU.

Table III
ROOT CAUSES DIAGNOSIS RESULT

Benchmark	Interference	CPU	Memory	Network	Disk
Wordcount	CPU	**86.5**	35.0	20.0	60.0
	Memory	61.2	**62.6**	20.4	36.0
	Network	51.5	32.5	**85.0**	32.4
	Disk	60.2	40.5	26.2	**82.5**
	Normal	8.5	3.5	5.2	10.3
Kmeans	CPU	**86.0**	53.1	24.5	42.3
	Memory	**60.5**	53.5	35.6	30.5
	Network	43.5	35.2	**87.2**	42.5
	Disk	76.5	53.2	46.2	**82.3**
	Normal	8.6	2.3	3.6	9.6
PageRank	CPU	**83.2**	43.3	24.3	52.5
	Memory	65.4	**67.6**	26.5	45.0
	Network	53.5	46.8	**85.8**	51.0
	Disk	60.3	53.6	25.5	**75.6**
	Normal	9.1	4.5	3.6	10.2

D. Discussion

Our approach is only tested on clusters with injecting interference on a single node. In order to show considerable effect, the interference will last a while. Additionally, as our approach is based on the task analysis inside each stage, it requires the target application with a certain amount of task partitions for each stage. Furthermore, our approach would be less suitable for analyzing user's log with different Garbage Collectors such as G1, CMS, and the new version of Spark log with different Spark schedulers.

V. Related Work

Abnormal tasks could lead to performance degradation in the big data computing frameworks [14, 15] and their root causes are complicated. Ananthanarayanan *et al.* [3] classified root causes into three categories: machine characteristics, which are the main reason, such as CPU usage, memory availability, and disk failure; network characteristics faults like the network bandwidth limitation and package drop; the internals of the execution environment such as data-work partitioning. Garbageman *et al.* [5] proposed that most common cause for abnormal occurrence is server resource utilization and data skew problem only takes 3% of total root causes. Therefore, we consider machine resources include CPU, memory, network, and disk as main root causes to analyze, and ignore data skew.

There are two kinds of approaches in abnormal detection, online and off-line. Some monitoring-related online approaches have been investigated. For example, Ananthanarayanan *et al.* [3] provide a tool called MANTRI that monitors tasks and outliers using cause- and resource-aware techniques. MANTRI uses real-time progress to detect outliers in their lifetime. Spark and Hadoop themselves provide an on-line "speculation", which is a built-in component to detect stragglers. There are many off-line approaches analyze log to locate the error event positions [12, 13, 18]. Moreover, Xu *et al.* [17] use an automatic log parser to parse source code and combine PCA to detect anomaly, it is based on abstract syntax tree (AST) to analyze source code and uses machine learning to train data. Tan *et al.* [16] provides an approach to analyze Hadoop log by extracting state-machine views of a distributed system's execution. Moreover, it combines control-flow and data-flow generated from debug log to catch normal system events and errors. In our approach, we extract features directly from log, but do not change any level of logging. Xu *et al.* [5] provide an experiment-based approach to determine root causes of stragglers using an integrated off-line and online model.

VI. CONCLUSIONS

In this paper, we propose a novel approach for Spark log analysis, and it identifies abnormals by combining both Spark log and GC log, and then analyze the root causes by weighted factors without using additional system monitoring information. Moreover, our approach can also identify the root causes of abnormals with probability.

VII. ACKNOWLEDGEMENT

This work was supported by NSF-CAREER-1622292.

References

[1] Apache Hadoop website. http://hadoop.apache.org/.

[2] Apache Spark website. http://Spark.apache.org/.

[3] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, volume 10, page 24, 2010.

[4] S. Diersen, E.-J. Lee, D. Spears, P. Chen, and L. Wang. Classification of seismic windows using artificial neural networks. *Procedia computer science*, 4:1572–1581, 2011.

[5] P. Garraghan, X. Ouyang, R. Yang, D. McKee, and J. Xu. Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters. *IEEE Transactions on Services Computing*, 2016.

[6] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *NSDI*, 2011.

[7] C. Gülcü. *The complete log4j manual*. QOS. ch, 2003.

[8] H. Huang, L. Wang, E.-J. Lee, and P. Chen. An mpi-cuda implementation and optimization for parallel sparse equations and least squares (lsqr). *Procedia Computer Science*, 9:76–85, 2012.

[9] H. Huang, L. Wang, B. C. Tak, L. Wang, and C. Tang. Cap3: A cloud auto-provisioning framework for parallel processing using on-demand and spot instances. In *IEEE Intl. Conference on Cloud Computing*, 2013.

[10] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In *The 12th ACM International Conference on Computing Frontiers*, page 53. ACM, 2015.

[11] H. Ma, S. R. Diersen, L. Wang, C. Liao, D. Quinlan, and Z. Yang. Symbolic analysis of concurrency errors in openmp programs. In *Intl. Conference on Parallel Processing (ICPP)*. IEEE, 2013.

[12] A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *DSN*. IEEE, 2007.

[13] S. Ryza, U. Laserson, S. Owen, and J. Wills. *Advanced Analytics with Spark: Patterns for Learning from Data at Scale*. O'Reilly Media, 2015.

[14] V. Subramanian, H. Ma, L. Wang, E. Lee, and P. Chen. Rapid 3d seismic source inversion using windows azure and amazon EC2. In *SERVICES*. IEEE, 2011.

[15] V. Subramanian, L. Wang, E. Lee, and P. Chen. Rapid processing of synthetic seismograms using windows azure cloud. In *CloudCom*, 2010.

[16] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. Salsa: Analyzing logs as state machines. *WASL*, 8:6–6, 2008.

[17] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP*. ACM, 2009.

[18] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. USENIX Association, 2012.

[19] H. Zhang, H. Huang, and L. Wang. MRapid: An efficient short job optimizer on Hadoop. In *the 31st IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2017.