Accepted Manuscript

LADRA: Log-based abnormal task detection and root-cause analysis in big data processing with Spark

Siyang Lu, Xiang Wei, Bingbing Rao, Byungchul Tak, Long Wang, Liqiang Wang

PII: S0167-739X(18)30768-4
DOI: https://doi.org/10.1016/j.future.2018.12.002
Reference: FUTURE 4623

To appear in: *Future Generation Computer Systems*

Received date : 1 April 2018
Revised date : 8 November 2018
Accepted date : 6 December 2018

Please cite this article as: S. Lu, X. Wei, B. Rao et al., LADRA: Log-based abnormal task detection and root-cause analysis in big data processing with Spark, *Future Generation Computer Systems* (2019), https://doi.org/10.1016/j.future.2018.12.002

LADRA: Log-Based Abnormal Task Detection and Root-Cause Analysis in Big Data Processing with Spark

Siyang Lu[a,*], Xiang Wei[a,b], Bingbing Rao[a], Byungchul Tak[c], Long Wang[d], Liqiang Wang[a,*]

[a]*Dept. of Computer Science, University of Central Florida, Orlando, FL, USA*
[b]*School of Software Engineering, Beijing Jiaotong University, China*
[c]*Dept. of Computer Science and Engineering, Kyungpook National University, Korea*
[d]*IBM T.J. Watson Research Center, Yorktown Heights, NY, USA*

Abstract

As big data processing is being widely adopted by many domains, massive amount of generated data become more reliant on the parallel computing platforms for analysis, wherein Spark is one of the most widely used frameworks. Spark's abnormal tasks may cause significant performance degradation, and it is extremely challenging to detect and diagnose the root causes. To that end, we propose an innovative tool, named *LADRA*, for **l**og-based **a**bnormal tasks **d**etection and **r**oot-cause **a**nalysis using Spark logs. In LADRA, a log parser first converts raw log files into structured data and extracts features. Then, a detection method is proposed to detect where and when abnormal tasks happen. In order to analyze root causes we further extract pre-defined factors based on these features. Finally, we leverage General Regression Neural Network (GRNN) to identify root causes for abnormal tasks. The likelihood of reported root causes are presented to users according to the weighted factors by GRNN. LADRA is an off-line tool that can accurately analyze abnormality without extra monitoring overhead. Four potential root causes, *i.e.*, CPU, memory, network, and disk I/O, are considered. We have tested LADRA atop of three Spark benchmarks by injecting aforementioned root causes. Experi-

*Corresponding authors. This work was supported in part by NSF-1836881.
Email addresses: `siyang@knights.ucf.edu` (Siyang Lu), `lwang@cs.ucf.edu` (Liqiang Wang)

mental results show that our proposed approach is more accurate in the root cause analysis than other existing methods.

Keywords: Spark, Log Analysis, Abnormal Task, Root Cause

1. Introduction

Parallel computing frameworks that follows the MapReduce [1] paradigm are widely-used in real-world big data applications to handle batch and streaming data. Among these, Spark [2] has recently gained wide-adoption. Different from the Hadoop framework [3], Spark supports a more general programming model, in which an in-memory technique, called Resilient Distributed Dataset (RDD) [4], is used to store the input and intermediate data generated during computation stages.

While Spark is highly successful for data analytics, it could suffer from significant performance degradation under the existence of abnormal tasks. A task is considered *abnormal* if it shows significant delay in comparison with other tasks within the same stage. A few causes of such performance degradation can be due to ineffective coding, resource contention, and data locality problems [5, 6, 7, 8].

To mitigate such performance problems, Spark employs a speculation mechanism [9] to detect stragglers during runtime, in which slow tasks are re-scheduled after marked as stragglers. Spark checks and performs speculative execution of tasks till a specified fraction (defined by `spark.speculation.quantile`, which is 75% by default) of tasks is completed. Spark identifies stragglers by checking whether the running tasks are much slower (*e.g.*, 1.5 times, by default) than the median of all successfully completed tasks in the current stage. However, speculation mechanism cannot detect all stragglers and does not provide the root causes of degraded performance. In addition, monitoring tools are usually heavy-weight and cause significant overhead, which may impact the performance of Spark even for normal executions. Therefore, abnormal task detection and root cause analysis still remain grand challenges.

This paper proposes LADRA, an off-line tool for **l**og-based **a**bnormal tasks

detection and **r**oot-cause **a**nalysis for big data processing with Spark. LADRA detects abnormal tasks by examining features extracted from logs and analyzes them to find root causes via a neural network model. Specifically, our proposed approach adopts a statistical spatial-temporal analysis for Spark logs, which consists of Spark execution logs and JVM garbage collection (GC) logs related to resource usage. LADRA's abnormal task detection method is more effective than Spark speculation, as all Spark stages are considered and abnormal tasks happened in any life span could be detected. Moreover, Spark's report could be inaccurate because Spark uses only fixed amount of finished task duration to speculate the unfinished tasks. Our approach reports the likelihood of each potential root cause, which can be leveraged by users to tune resource allocations and reduce the impact of abnormal tasks. For instance, in one of our experiments, LADRA reports that abnormal tasks are caused 80% by network issues and 20% by CPU issues on victim nodes, users may check the network condition first, then tune CPU usage accordingly. There are four major root causes for task abnormalities: CPU, memory, network, and disk I/O, all of which are considered by this paper.

We make the following contributions in this work.

- An abnormality detection method is proposed that can accurately locate where and when abnormal task executions happen by analyzing Spark logs.
- 22 log features and 7 factors are identified to be critical in exposing the degree of abnormality from the analysis of Spark logs and GC logs.
- A neural network-based analysis method is proposed, which is more accurate and provides the ranked likelihood for true root causes in order to better understand the performance problems and to tune the Spark settings.

The rest of the paper is organized as follows. Section 2 introduces the background knowledge of Spark and surveys the related work. Section 3 gives an overview of our approach. Section 4 illustrates the feature extraction from Spark logs and abnormal task detection based on these features. Section 5 presents factor synthesization for root cause analysis. Section 6 describes the details of root cause analysis using GRNN. Section 7 shows our experimental results by evaluating our approach on several widely used benchmarks. Section 8 summarizes our method and discusses its limitations and future work.

2. Related work and background

In this section, we give brief background of Spark scheduling mechanisms and its log structures. Then, we review related work in the area of the root cause analysis for big data platforms.

This paper significantly extends our previous paper [10], a statistical method for detecting task abnormalities and analyzing root causes. Compared with our prior work, the factor extraction is extended and the weighted statistical method for detection is improved, which are presented in Section 6.1. Our previous approach diagnoses root causes by applying weights to each factor. Such rule-based weight calculation approaches may cause false positives. Moreover, due to the complex relationships between hardware and software and between input and output, we believe that a non-linear model can do a better job. As we stated before, the root cause detection is better to be treated as a regression rather than a classification problem. Hence, in this paper, the most significant extension is that we propose a new General Regression Neural Network (GRNN) as a better choice, which can avoid the ad-hoc factor selection and weight computing.

2.1. Spark architecture and its log structure

Spark architecture: Apache Spark is an in-memory parallel computing framework for large-scale data processing. Moreover, to achieve the scalability and fault tolerance, Spark introduces resilient distributed data set (RDD) [4], which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. As shown in Figure 1, Spark cluster consists of one master node and several slave nodes, named as workers, which may contain one or more executors. When a Spark application is submitted, the master will request computing resource from the resource manager based on the requirement of the application. When the resource is ready, Spark scheduler distributes tasks to all executors to run in parallel. During this process, the master node will monitor the status of executors and collect results from worker nodes.

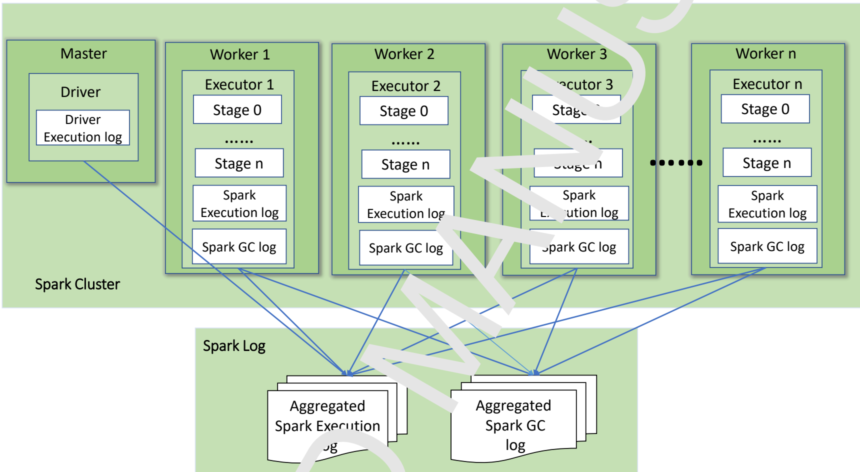

Figure 1: Spark framework and log files

Spark logs include execution logs and JVM GC logs. Spark driver (master node) collects the information of all executors (*i.e.*, driver log), and each executor records the status of tasks, stages, and jobs within the executor (*i.e.*, execution log). Besides these logs, Spark JVM Garbage Collection (GC) logs are also used by our analysis, which are the output from two output channels, `stderr` and `stdout`. When an application is finished, we collect all Spark logs and aggregate them into two different categories: execution logs and GC logs. An example is shown in Figure 2.

Spark uses "log4j", a JAVA logging framework, as its logging framework. Spark users can customize "log4j" by changing configuration parameters, such

17/02/22 21:04:02.259 INFO
TaskSetManager: Starting task 12.0 in stage 1.0 (TID 58, 10.190.128.101, partition 12, ANY, 5900 bytes)
17/02/22 21:04:02.259 INFO
CoarseGrainedSchedulerBackend$DriverEndpoint: Launching task 58 on executor id: 1 hostname: 10.190.128.101.
17/02/22 21:04:02.276 INFO
TaskSetManager: Finished task 1.0 in stage 1.0 (TID 47) in 14975 ms on 10.190.128.101 (1/384)

Figure 2: An example of Spark execution log.

as log level, log pattern, and log direction. In this paper, we use the default configurations in "log4j". As shown in Figure 2, each line of Spark execution log contains four types of information: *timestamp* with ISO format, *logging level* (*e.g.*, INFO, WARNING, or ERROR), *related class* (which class prints out this message) and *message content*. A message content contains two main kinds of information: constant keywords (*e.g.*, `Finished task in stage TID in ms on`), and variables (*e.g.*, `1.0 1.0 47 14975..`).

[*GC (Allocation Failure)*
[*PSYoungGen: 95744K->9080K(111616K)*] *95744K->9088K(367104K), 0.0087250 secs*]
[*Times: user=0.03, sys=0.01, real=0.01 secs*]

Figure 3: An example of Spark garbage collection (GC) log.

During the execution of a spark application, JVM monitors memory usage and outputs its status to GC logs when garbage collection is invoked. GC logs report two kinds of memory usage: heap space and young generation space, where young generation space is a part of heap memory space to store new objects. Figure 3 shows an example of Spark JVM GC log, where "Allocation Failure" invokes this GC operation, and "PSYoungGen" shows the usage of young generation memory space. In "95744K->9080K(111616K)", the first numeric is the young space before this GC happens, the second one is the young space after this GC, and the last one is the total young memory space. Similarly, "95744K->9088K(367104K)) illustrates heap memory instead of young generation space.

2.2. Related work

2.2.1. Root causes

There are several categories of the root causes for the abnormal performances. Ananthanarayanan *et al.* [11] identify three categories of root causes for Map-Reduce outliers: the key role cause is machine characteristics (resource problems), the other two causes are network and data skew problem. Ibidunmoye *et al.* [12] depict that four root causes may cause bottlenecks, which are system resource, workload size, platform problems, and application (buggy codes). Garbageman *et al.* [13] analyze around 20-day cloud center data and summarize that most common root cause in cloud center of abnormal occurrence is server resource utilization, and data skew problems only take 3% of total root causes. According to the above studies on real world experiment, the primary root causes of abnormal tasks are machine resources, which includes CPU, memory, network, and disk I/O. Moreover, the mentioned resource root causes mainly impact the performance of CPS computation layers. Therefore, in our paper, we consider the only the four main root causes, and ignore data skew and ineffective code problems.

2.2.2. Existing approaches

Statistical and machine learning techniques are promising approaches in the root causes analysis, and their combination has been widely used in the parallel computing area to solve performance degradation problem caused by abnormal executions. Abnormality detection and analysis using this approach can be categorized largely into online, offline, and combination of online and offline approaches.

Online detection: The online detection strategy is invoked during the executions of applications. For example, both Spark and Hadoop provide online "speculation"[9], which is a built-in component for detecting stragglers statistically. Although it can detect stragglers during runtime, it does not offer the root causes. In addition, the speculation is often inaccurate, *i.e.*, it may raise too many false alarms [14]. Chen *et al.* [15] propose a tool called Pinpoint that monitors the execution and uses log traces to identify the fault modules in J2EE applications via standard data mining approaches. A stream-based mining algorithm for online anomalies prediction is presented by Gu *et al.* [16]. Ananthanarayanan *et al.* [11] design a task monitoring tool called Mantri, which can cut outliers and restart tasks in real time according to its monitoring strategy.

Offline detection: Nevertheless, monitoring data may not be always accessible from the user side, due to the fact that the monitoring tools are hard to install and tune. Hence, some studies focus on the off-line strategy by analyzing logs instead of monitoring [17, 18]. For example, Tan *et al.* [19] introduce a pure off-line state machine tool called SALSA, which simulates data flows and control flows in big data systems with statistical method, and leverages Hadoop's historical execution logs. Then, Tan *et al.* [20] build up a performance tool to visualized MapReduce which based on SALSA. However, those state machine based statistical approaches can not extract feature by itself. Chen *et al.* [21] propose a self-adaptive tool called SAMR, which adds weights for calculating each task duration according to historical data analysis. Xu *et al.* [22] use an automatic log parser to parse source code and combine PCA to detect anomaly, which is based on the abstract syntax tree (AST) to analyze source code and uses machine learning to train data. Qi *et al.* [23] leverage Classification and Regression Tree (CART) to analyze straggler root causes by using Spark event logs and monitoring data (hardware metrics such as CPU status, disk read/write rate and network send/receive rate) which collected by synchronous sampling tool. However, our approach is a pure off-line method and only leverage Spark log to analyze abnormal tasks. Furthermore, we prefer using probabilistic output to determine the degree and category of abnormality, rather than considering the problem of classifications of positive and negative samples that CART did.

Combination of online and offline detection: In order to achieve higher accuracy, the offline strategy can be combined with the online one. Garraghan *et al.* [13] propose an empirical approach to extract execution paths for straggler detection by leveraging an integrated offline and online model. Some machine learning approaches are also leveraged in predicting system faults using logs and monitoring data, which are similar to the root cause analysis problem. Fulp *et al.* [24] leverage a sliding window to parse system logs and predict failures using SVM. Yadwadkar *et al.* [25] propose an offline approach that works with resource usage data collected from the monitoring tool Ganglia [26]. It leverages Hidden Markov Models (HMM), which is a liner machine learning approach. Moreover, there are some off-line approaches that analyze both log files and monitoring data to identify abnormal events. Aguilera *et al.* [27] propose two statistical methods to discover causal paths in distributed systems by analyzing historical log and monitoring data from the traces of applications. The most closely related work to our approach is BigRoots [28], which detects stragglers by Spark speculation and analyzes the root causes by extracted features. It leverages experience rule to extract features for each task from application log and monitoring data. However, the threshold in Spark speculation is not proper to detect abnormal tasks. In addition, BigRoots considers only the features for each individual task, which can not capture the status change of the cluster, thus such a rule-based method is very limited. In our method, we choose the combination of features to create the factors presenting the status change of the whole cluster, and a GRNN technique is leveraged instead of a rule-based statistical approach to avoid the limits.

3. Overview of IADRA's approach

Although Spark logs are informative, they lack direct information about the root cause of abnormal tasks. Thus, simple keyword-based log search is ineffective for diagnosing the abnormal tasks, which motivates us to design an automatic approach to help users detect abnormal tasks and analyze their root causes. An overview of our tool is depicted in Figure 4, which contains five primary components: log prepossessing, feature extraction, abnormal task detection, factor extraction, and root cause analysis.

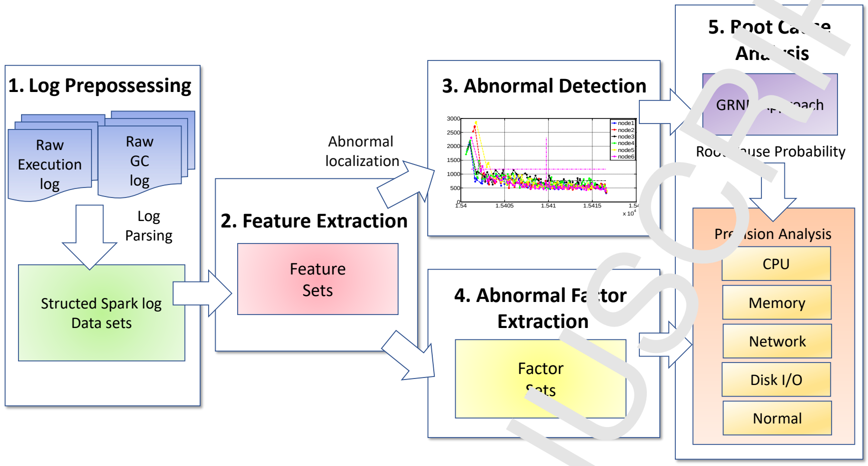

Figure 4: The workflow of LADRA

1. *Log prepossessing*: Spark log contains a large amount of information. In order to extract useful information for analysis, we first collect all Spark logs, including execution logs and JVM GC logs, from the driver node and all worker nodes. Then, we use a parser to eliminate noisy and trivial logs, and convert them into structured data.

2. *Feature extraction*: Based on the Spark scheduling and abnormal task occurring conditions, we quantify the data locality feature with a binary number format. Then, we screen structured logs and select three kinds of feature datasets: execution-related, memory-related, and system-related. Finally, we store them into two numerical matrices: execution log matrix and GC matrix.

3. *Abnormal detection:* We implement a statistical abnormal detection algorithm to detect where and when the abnormal tasks happen based on the analysis of execution-related feature sets. This detection method determines the threshold by calculating the standard deviation of task duration and use it to detect abnormal tasks in each stage from Spark logs, which is introduced in Section 4.

4. *Abnormal factor extraction:* According to our empirical case study, we combine special features to synthesize two kinds of factors, the speed factor and the degree factor, which describe the status of each node in the whole cluster. Section 5 introduced these factors used by our root cause analysis method.

5. *Root cause analysis:* We propose a General Regression Neural Network (GRNN) based approach for our root-cause analysis, in which probability result can be calculated more accurately than our previous statistical work. Our experiments show that the GRNN-based approach has more accurate results than existing approaches, which are introduced in details in Section 6.

4. Log feature extraction and abnormal task detection

4.1. Log feature extraction

When an abnormal task happens, it usually does not cast any warnings or error messages. As Spark does not directly reveal any information about abnormal tasks, it is a very challenging problem to detect these problems. Our approach starts from understanding the Spark scheduling strategy, then extracts features associated with CPU, memory, network, and disk I/O to build a feature matrix, which reflects the whole cluster's status. These features can be classified into three categories: execution-related, memory-related, and system-related, as shown in Table 1.

The execution-related features are extracted from Spark execution logs, including (1) the ID number of each task, stage, executor, job, and host, (2) the duration of each task, stage, and job, (3) the whole application execution time, (4) the timestamp for each event, and (5) data locality. Spark GC logs represent JVM memory usage of each executor in workers, from which we can extract memory-related features such as heap usage, young space usage before GC, young space usage after GC. In addition, system related features can be also extracted from GC logs, such as real time, system time, and user time.

Table 1: Extracted features for abnormal task detection

Feature Category	Feature Name		
Execution related	Task ID	Job ID	Task duration
	Stage ID	Job duration	Data locality
	Host ID	Stage duration	Timestamp
	Executor ID	Application execution time	
Memory related	GC time	After young GC	After Heap GC
	Full GC time	Before young GC	Before Heap GC
	Heap space	GC category	
System related	Real time	CPU time	User time

4.2. Abnormal task detection

Our abnormal task detection is based on the extracted feature sets. In order to eliminate the false negative problem in the Spark speculation's detection mentioned in Section 1, a more robust approach is designed to locate where and when an abnormal case happens, which includes the following two steps.

Step-1: Comparing task duration on inter-node:

One basic rule for abnormal task identification is that the duration of abnormal task is relatively much longer than the duration of normal tasks (long tail). In the existing approaches for abnormal detection, both Hadoop and Spark use speculation, and [13] uses "mean" and "median" to decide the threshold. However, to seek a more reasonable abnormal detection strategy, we consider not only the mean and median of the task running times, but also the distribution of the whole tasks' duration including the standard deviation. In this way, we can get a macro awareness on the task duration, and then based on the distribution of data, a more reasonable threshold can be determined to differentiate the abnormal from the normal ones.

We compare the duration of tasks in the same stage but across different nodes (inter-node). Let $T\_task_{i,j,k}$ denote the execution duration of task k in stage i on node j. And let avg_stage_i denote the average execution time of all tasks, which run on different nodes in the same stage i.

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

where J and K_j are the total node numbers and total task numbers in node j, respectively.

To determine a more appropriate threshold, we leverage the standard deviation of tasks duration in stage j of all nodes, which is denoted by std_stage_i, and λ is a threshold parameter used in Spark speculation, which is 1.5 by default. Thus, abnormal tasks can be determined by the following conditions:

$$task_k = \begin{cases} abnormal & T\_task_k > avg\_stage_i + \lambda * std\_stage_i \\ normal & \text{otherwise.} \end{cases} \tag{2}$$

Step-2: Locating abnormal task happening: After the first step, all tasks are classified into "normal" and "abnormal", the time line is labeled as a vector with binary number (*i.e.*, 0 or 1, which denotes normal and abnormal, respectively). To smooth the outliers (*e.g.*, 1 appears after many continuous 0) inside each vector, which could be an abrupt change but inconsistent abnormal case, we empirically set a sliding window with the size of 5 to scan 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 timestamp and end timestamp of the current abnormal task. Note that, since Spark logs record the task finishing time but not the start time, we locate the real abnormal task's start time as the recorded task finishing time minus its duration. Moreover, to detect abnormal tasks in each stage, we classify tasks into two sets. One set includes the initial tasks whose start timestamp are the beginning of each stage, as these tasks often have more overhead (such as loading code and Java jar packages), and they usually last much longer than the following tasks. The other set consists of the remaining 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 abnormal tasks happen, but also locate where and when they happen.

Figure 5 shows abnormal detection process in our experiment for Spark WordCount under CPU interference. Figure 5 (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 duration longer than the threshold are determined as abnormal tasks; otherwise, they are normal. Figure 5 (c) shows the execution-related feature visualization in the whole execution time. 5 (d) uses memory-related feature to display memory occupation along the execution of its corresponding working stages.

As we mentioned before, the data skew problem is not within our four considered root causes. Therefore, in the real analysis, those abnormal tasks caused by data skew should be eliminated as a noise. Data skew tasks can be easily detected by checking data locality features (*i.e.*, target data is not on the current node) combined with task duration features from execution logs.

5. Factor extraction for root-cause analysis

To look for the root causes of abnormal tasks, we introduce abnormal factors, which are the synthesis of features based on the empirical study on the 22 features in Spark log matrix and GC matrix. Those factors are normalized features that present status change of the whole cluster, not only for assessing individual components, such as task and stage, but also a series of abnormal tasks, which may be generated by continuous interference affecting the cluster.

In normal cases, each factor should be close to 1; otherwise, it implies an abnormal case. In our factors' definition, j denotes the jth node, J presents a set of nodes; i indicates 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 a GC record set. All factors

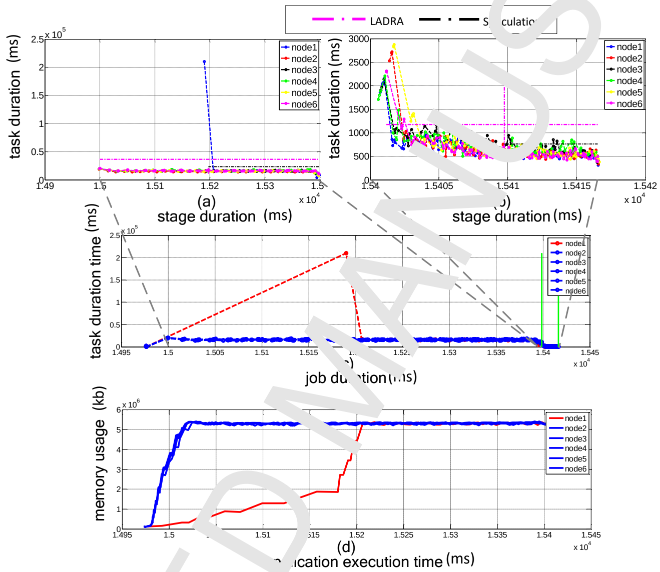

Figure 5: Abnormal detection under CPU interference in the experiment of WordCount: (a) Abnormal detection result in Stage-1. (b) Abnormal detection result in Stage-2. (c) Execution-related feature visualization for abnormal detection in the whole execution. (d) Memory-related feature visualization for abnormal detection in the whole execution.

used to determine root causes are listed as below.

Degree of Abnormal Ratio (DAR) describes the degree of imbalanced scheduling of victim nodes, due to the fact that the victim nodes will be scheduled with fewer tasks than other normal nodes. For example, as shown in Figure 6, CPU interference can cause fewer tasks (red dots) to be scheduled at a victim node (node1) than normal nodes. Eq. (3) illustrates the degree of abnormal ratio in a certain stage. Therefore, the factor DAR implies that the number of tasks in intra-node on a certain stage can be used for abnormal detection.

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

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

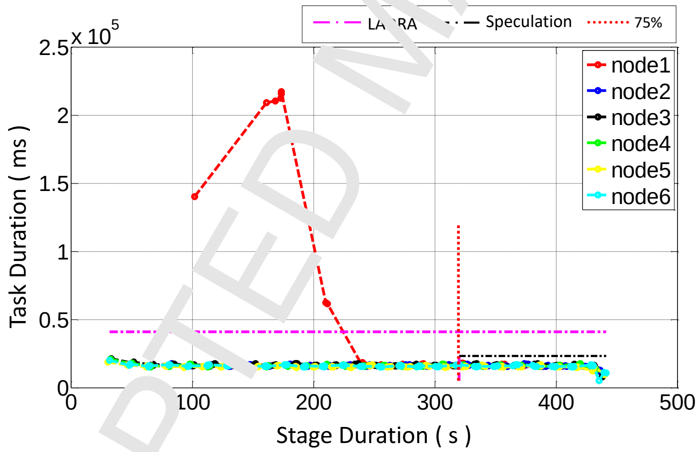

Figure 6: Task duration variation in CPU interference injected after Sorting application has been submitted for 60s, and continuously impacts for 120s.

Degree of Abnormal Duration (DAD) is used to measure the average task duration, as the abnormal nodes often record longer task duration.

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

where avg_node_j is defined as:

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

Degree of CPU Occupation (DCO) describes the degree of CPU occupation by calculating 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 a multi-threading way. However, if the "realTime" is bigger than "sysTime+userTime", it may indicate that the system is quite busy due to CPU or disk I/O contentions. We choose a max value across nodes as the final factor.

$$DCO = max(\underset{j \in J}{avg}(\frac{realTime_{i,j}}{sysTime_{i,j} + userTime_{i,j}})) \tag{6}$$

Memory Change Speed (MCS) indicates the speed of memory usage change according to GC curve. Due to the fact that under CPU, memory, and disk I/O interference, the victim node's GC curve will vary slower than the normal nodes' GC curve, as shown in Figure 7. $start_a$ and $stable_a$ are the points of the start position (the corresponding memory usage at abnormal starting time) and the stable memory usage position, respectively. $start_b$ and $stable_a$ are the start and end positions of abnormal memory, respectively, which are obtained by analyzing logs introduced before. The intuition is that the interfered node gradually uses less memory than normal nodes under interference, as shown in Figure 7. Hence, we use the area under GC curve a in the whole cluster ($start_a$ of normal node) to calculate this factor, as shown in Eq. (7).

$$MCS = \frac{\int_{start}^{stable_a} f(x_a)dx_a}{\int_{start}^{stable_b} f(x_b)dx_b} \tag{7}$$

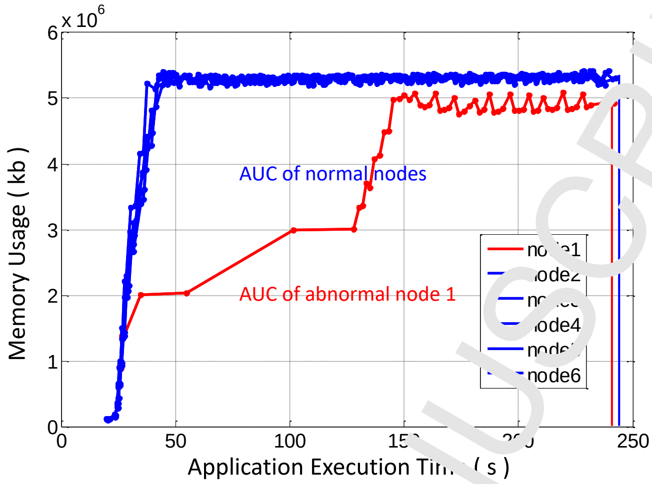

Figure 7: Memory usage variation in CPU interference injected after WordCount application has been submitted for 20s, and continuously impacts for 120s.

Abnormal Recovery Speed (ARS) measures the speed of abnormal task's recovery. Since one Spark node often accesses data from other nodes, it can leads to network interference propagation. It is both inter-node and intra-node problem. We can detect network interference happening inside cluster, as shown in Figure 8, which is the location of our detected interference and shows that task duration will be affected by delayed data transmission. We leverage Eq. (8) to calculate this factor, where abn_prob_j indicates the ratio of the abnormals 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, we identify them with a potential of network interference; if their sum is used, we cannot detect this joint probability. Meanwhile, the exponential is to make sure this factor is no less than 1. Hence, the phenomenon of error propagation will be detected and quantified by calculating this factor.

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

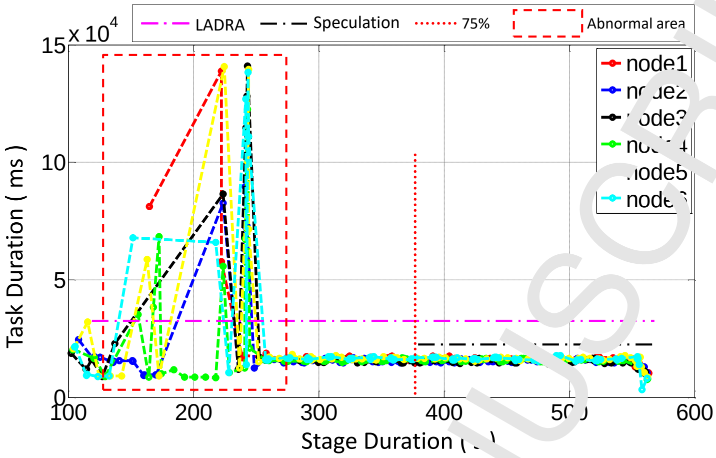

Figure 8: Task duration variation in Network interference injected after WordCount has been executed for 100s, and continuously impacts for 160s.

Degree of Memory Change (DMC) describes how much of memory usage changed during the execution in each node. In fact, when network bandwidth is limited, or the network speed slows down, the victim node gets affected by that interference, and tasks will wait for their data transformation from other nodes. Hence, the tasks will pause or work very slowly, and data transfer rate becomes low, as shown in Figure 9. We leverage Eq. (9) to find the longest horizontal line that presents the conditions under which tasks' progress become tardy (*e.g.*, CPU is relatively idle and memory remains the same). In Eq. (9), $m_{j,n}$ indicates the gradient of memory changing in the nth task on node j. First, the max value of gradient is calculated for each GC point, denoted as m. Second, we make a trade-off between its gradient and the corresponding horizontal length to identify the longest horizontal line in each node. Then, 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" ($e^{-|m_{j,n}|} * (x_{j,n} - x_{j,n-1})$), where e is to ensure that the whole factor of

b not less than 1.

$$DMC = \frac{\max\limits_{j \in J}\{\max\limits_{n \in N}[e^{-|m_{j,n}|} * (x_{j,n} - x_{j,n-1})]\}}{\min\limits_{j \in J}\{\max\limits_{n \in N}[e^{-|m_{j,n}|} * (x_{j,n} - x_{j,n-1})]\}} \tag{9}$$

where $m_{j,n} = \frac{y_{j,n} - y_{j,n-1}}{x_{j,n} - x_{j,n-1}}$.

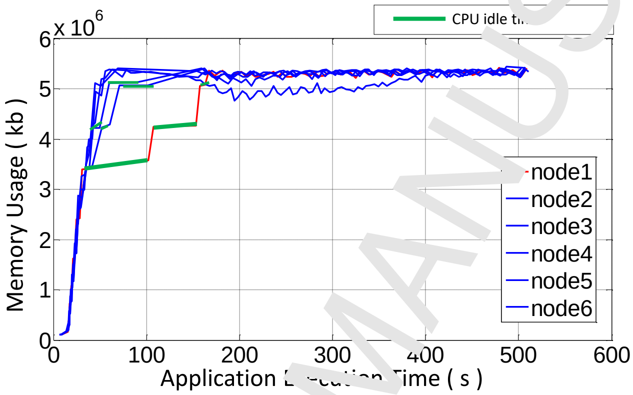

Figure 9: Memory usage variation in Network interference injected after WordCount has been executed for 30s, and continuously impacts for 160s.

Degree of Loading Delay (DLD) measures how much difference of loading duration on cluster nodes. Note that the initial task at the beginning of each stage always has a higher overhead to load data compared with the rest tasks. Similar to the factor DMC, instead of taking all tasks inside the detected stage into consideration, here, the first task of each node is used to replace the "avg_node_j".

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.

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

6. Root cause analysis

6.1. Statistical rule based approach

We propose a statistical rule based approach for root cause analysis extended from [10]. As shown in Table 2, 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 DAR and DAD are used to test whether the interference is CPU or not, because CPU interference can reduce the number of scheduled tasks and increase the abnormal tasks' execution time. Factor DCO indicates the degree of CPU occupation, and CPU interference will slow down of the performance compared to normal cases. Factor MCS is used to measure memory changing rate, because CPU interference may lead memory change, thus the nodes become slow than other regular nodes.

For the network-related interference, because of its propagation, the nodes interfered earlier will often recover earlier, too. So our approach is to detect the first recovered node as the initial network-interfered node, and the degree ARS quantitatively describes the interference. When network interference occurs, tasks are usually waiting for data delivery (factor DMC).

For the memory-related interference, 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 DAR) and task duration (factor DAD) are also added to determine such root causes with certain weights. Moreover, the memory interference will impact memory usage, and the factor MCS should be considered for this root cause detection.

To determine disk-related interference, we introduce the factor DLD 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 DLD), CPU will become busy, and memory usage is different from

Table 2: Related factors for each root cause

Factor	CPU	Mem	Network	Disk
DAR	√	√		
DAD	√	√	√	
DCO	√			√
MCS	√	√		√
ARS			√	
DMC			√	
DLD				√

other nodes'. Therefore, the memory changing rate (factor MCS) and CPU Occupation (factor DCO) 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 accurately as Eq. (11) shows. 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 linear regression on training sets that we obtained from experiments. Eq. (12) is proposed to calculate the final probability that the abnormal belongs to each of the root causes.

$$\begin{gathered} CPU = 0.3 * DAR + 0.3 * DAD + 0.2 * DCO + 0.2 * MCS \\ Memory = 0.25 * DAR + 0.25 * DAD + 0.5 * MCS \\ Network = 0.1 * DAD + 0.4 * ARS + 0.5 * DMC \\ Disk = 0.2 * DCO + 0.2 * MCS + 0.6 * DLD \end{gathered} \tag{11}$$

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

To sum up, the statistical rule based approach offers a reasonable result to explain its root causes probabilities. However it can not give a satisfied result with higher precision for its classifying. Since the relationship between factors is not simply linearly correlated, and we also changed old factor MCR to a new factor MCS with AUC calculation instead of gradients calculation and add it to our factor sets. From this point, a GRNN-based approach is proposed for root cause analysis to consider non-linearly correlated relationship of new factor set, and avoid human ad-hoc choosing and classification.

6.2. GRNN approach

In this paper, we propose a new neural network based model to automatically calculate the probability of each root cause. We use a one-pass training neural network, GRNN, to create a smooth transition and more accurate results.

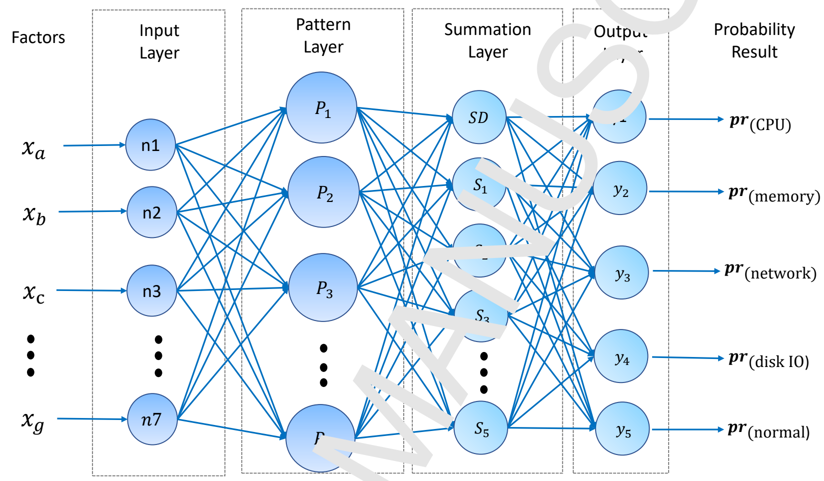

Figure 10: The architecture of our GRNN-based model for root-cause analysis.

GRNN is a simple and efficient network with fast computing speed, because GRNNs transfer function (pattern layer) is a kind of Gaussian function, and it could achieve local approximation with fast speed without any back propagation training operations. As due to the fact that classical neural networks, especially deep neural networks, require much more efforts to tune hyper-parameters, which has been proved to be not proper to fit small datasets, just like our Spark log. Hence, we choose GRNN in our design. Thanks to its flexible structure, which can automatically set the number of nerve cells in the pattern layer. In brief, the BP (Back Propagation) based deep learning algorithms may be vulnerable to the over-fitting problem especially when the dataset is small, which is just the characteristic of our dataset. Traditional data fitting algorithms usually assumes that the data obey a certain distribution in advance, which can drastically affect the final result. As a non-parameter neural network model for data fitting, with its high efficiency and accuracy, GRNN is fully capable of dealing with our current problem. In addition, the experimental results demonstrate the effectiveness of GRNN compared with other attempts we have tried.

As a non-parameter neural network model for data fitting, with its high efficiency and accuracy, GRNN is fully capable of dealing with our current problem. In addition, the experimental results demonstrate the effectiveness of GRNN compared with other attempts we have tried. A representation of the GRNN architecture for our implementation of root cause identification is shown in Figure 10. Our model consists of four layers: input layer, pattern layer, summation layer, and output layer. According to our data structure, the input layer consists of 7 neurons, which indicates the dimension of our extracted input feature vector (x_a,x_b...x_g). The pattern layer is a fully connected layer, which consists of neurons with the same size as input data, and followed by the summation layer. At the end, the output layer of GRNN gives a prediction result on the probability for each root cause. We use softmax function to convert the output into a normalized one for more intuitive comparison.

The transfer function F_i in pattern layer is defined in (13), $\mathbf{X}$ denotes the input data, σ represents as a smooth parameter, which is set to 0.5 according to our experimental attempts. The hyper-parameter of σ is used to control the smoothness of the model. When the value is relatively large, it is equivalent to increasing the variance in the Gaussian density distribution, which makes the transition between different categories smoother. While the problem is that the classification boundary will be blurred. Conversely, when a smaller value is assigned to this hyper-parameter, the ability to fit real data of the model will be stronger but the generalization turns out to be relatively weak. In the following, summation layer is added, which contains two kinds of neurons: S-summation neuron (S) and D-summation neuron (SD), as defined in (14), respectively. SD neurons are used to calculate the arithmetic summation of pattern layer's output. The remaining S neurons weight summation for the output of pattern layer. The i denotes ith number of input data, j denotes the jth dimension of output, and S_j denotes the j^{th} S neuron output. Then, the w denotes weight in hidden layer. The label (output layer) here is a 5-dimensional one-hot vector with one indicating normal log and the rest four are injections. y indicates y_j indicates the j^{th} output item the output as defined in (15). Due to probability representation of root cause, after the output layer of GRNN, we add a softmax layer to convert the sum of 5-dimensional output to be 1.

$$F_i = exp\ (\frac{-(\mathbf{X}-\mathbf{X_i})^{\mathbf{T}}(\mathbf{X}-\mathbf{X_i})}{2\sigma^2}), where\ \mathbf{X} = [\mathbf{x_a}, \mathbf{x_b} ... \mathbf{x_g}]^{\mathbf{T}} \tag{13}$$

$$Summations \begin{cases} SD = \sum_{i=1}^{n}(F_i), & \text{where } i = 1 : n \\ S_j = \sum_{i=1}^{n}(w_{ij}F_i), & \text{where } j = 1, 2, 3, 4, 5 \end{cases} \tag{14}$$

where n is equal to input data set size.

$$y_j = \frac{S_j}{SD}, \text{where } j = 1, 2, 3, 4, 5 \tag{15}$$

To sum up, GRNN can select a dominant weight for each of our factors, and provide the root cause probability results with high accuracy.

7. Experiments

We evaluate LADRA on four widely used benchmarks and focus on the following two questions. (1) Can the abnormal tasks be detected? (2) What accuracy can LADRA's root cause analysis achieve? In the experiment, we conduct a series of interference injections to simulate various scenarios that lead to abnormal tasks.

7.1. Setup

Clusters. We set up an Apache Spark standalone cluster with one master node (labeled by m1) and six slave nodes (labeled by n1,n2,n3,n4,n5,n6) based on Amazon EC2 cloud resource. Each node is configured with type of "r3.xlarge" (2 virtual cores and 30GB of memory) and Ubuntu 16.04.9. We conduct a bunch of experiments atop of Apache Spark 2.2.0 with JDK 1.8.0, Scala-2.11.11, and Hadoop-2.7.4 packages. Given that an AWS instance is configured with EBS by default, it is difficult for us to inject disk I/O interference. Hence, we set up a 90G ephemeral disk for each instance and deploy a HDFS to store data.

Table 3: Benchmark resource intensity

	CPU	Memory	Disk I/O	Network
WordCount	√		√	√
Sorting	√		√	√
K-Means	√			√
PageRank	√	√		

Workload: In fact, some Spark applications may consume resources more intensively. According to previous studies on Spark performance [29], we choose four benchmarks built on Hibench [30] and one real-world CPS application in our experiments: WordCount, Sorting, PageRank, K-means, which cover the domain of statistical batch application, machine learning program, and iterative application. WordCount and Sorting are one-pass programs, K-means and PageRank are iterative programs. We characterize the benchmarks by resource intensive type and program type for underpinning our approach's scalability. The resource intensity of each benchmark is shown in Table 3. The characteristics of four benchmarks are listed as follows.

- WordCount is a one-pass program for counting how many times a word appears. We leverage RandomTextWriter in Hibench to generate 80G datasets as our workload and store it in HDFS. It is CPU-bound and disk-bound during map stage, then network-bound during reduce stage.
- Sorting is also a one-pass program that encounters heavy shuffle. The input data is generated by RandomTextWriter in Hibench. Sorting is disk-bound in sampling stage and CPU-bound in map stage, and its reduce stage is network-bound.
- K-means is an iterative clustering machine learning algorithm.The workload is generated by the k-means generator in Hibench, and is composed

of 80 million points and 12 columns (dimensions). It is CPU bound and network-bound during map stage.

- PageRank is an iterative ranking algorithm for graph computing. In order to analyze root causes of abnormal tasks with PageRank, we use Hibench PageRank as the testing workload, and generate eighty thousand vertices by Hibench's generator as input datasets. It is CPU-bound in each iteration's map stage, and network bound in each reduce stage.

  A CPS K-means is a real-world CPS application in civil engineering that we developed before. The workload data size is 1[illegible] GB and collected by sensors installed at a classroom building. Those sensors measure real time temperature and humidity from each classroom. The collected data set is leveraged for detecting outlier temperature and humidity. To solve this real-world problem with effective approaches, we implemented a K-means algorithm on Spark for pre-clustering and grouping sensor data into sub-clusters and decide the outliers.

7.2. LADRA interference framework

In order to induce abnormal tasks in the real execution for experiment, we design an interference framework that can inject four major resource (CPU, memory, disk I/O, and network) interference to mimic various abnormal scenarios. In order to simplify experiment, we apply all interference injection techniques only on node n1 for all test cases. In addition, for each injection, it will be launched during a time interval of 10 seconds and 60 seconds after the first spark job is initiated, and continue for 120 seconds to 300 seconds. Finally, when a test case is over, we recover all involved computing nodes to normal state by terminating all interference injections. Specifically, the following interference injections are used in our experiments:

- *CPU interference*: CPU Hog is simulated via spawning a bunch of processes at the same time to compete with Apache Spark processes. This injection causes CPU resource contention in consequence of limited CPU resource.

- *Memory interference*: Memory resource scarcity is simulated via running a program that requests a significant amount of memory in a certain time to compete with Apache Spark jobs, then we hold on this certain of memory space for a while. Thus, Garbage Collection will be frequently invoked to reclaim free space.
- *Disk interference*: Disk Hog (contention) is simulated via leveraging "`dd`" command to continuously read data and write them back to the ephemeral disk to compete with Apache Spark jobs. It impacts both write and read speed. After the interference is done, we clear the generated files and system cache space.
- *Network interference*: Network scenario is simulated when network latency has a great impact on Spark. Specifically, we use "`tc`" command to limit bandwidth between two computing nodes with specific duration. In this way, the data transmission rate will be slowed down for a while.

7.3. Abnormal task detection

To evaluate LADRA, we compare LADRA's detection with the Spark speculation. Each benchmark is executed 50 times without any interference injection, and 50 times under the circumstances of abnormal tasks. After that, we calculate the True Positive Rate (`TPR`) and False Positive Rate (`FPR`) results by counting the correct rate of each job classification as shown in Eq. (16) and Eq. (17). The comparison result is shown in Table 4.

As a build-in straggler detector, Spark speculation brings False Positive (`FP`) and True Negative (`TN`) problems in abnormal task detection. We compare LADRA with Spark speculation in details. For instance, Figure 11 shows one stage in a normal K-means execution, x-axis and y-axis present stage duration and task duration, respectively, and no abnormal tasks are detected by LADRA

Table 4: LADRA's abnormal task detection compares with Spark speculation's approach in four intensive benchmarks, where TPR = True Positive Rate, FPR = False Positive Rate.

Abnormal tasks detection	LADRA		Spark speculation	
	TPR	FPR	TPR	FPR
WordCount	0.96	0.06	0.94	0.8
Sorting	0.96	0.16	0.96	0.7
K-Means	0.7	0.1	0.2	0.7
CPS K-Means	0.7	0.1	0.2	0.7
PageRank	0.6	0.517	0.9	0.48

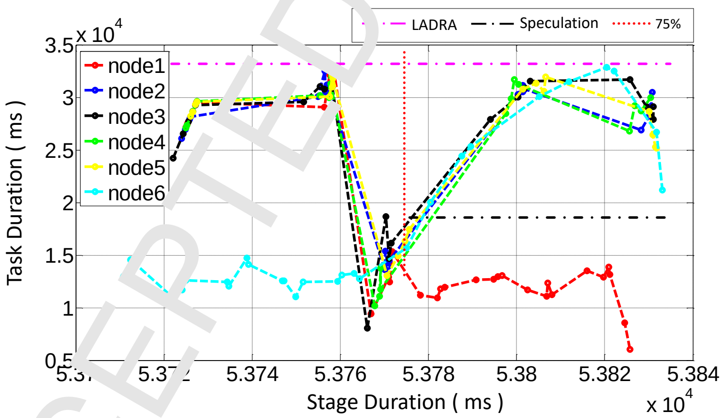

Figure 11: Abnormal task detection for K-means without interference injection.

Table 5: Root cause analysis result of LADRA's GRNN approach, TPR = True Positive Rate, P = Precision

GRNN	WordCount		Sorting		K-Means		CPS K-Means		Pagerank	
	TPR	P	TPR	P	TPR	P	TPR	P	TPR	P
CPU	1.000	1.000	1.000	0.940	0.857	0.835	0.866	0.837	0.951	0.826
Disk I/O	0.450	0.420	0.679	0.894	0.423	0.692	0.53[illegible]	0.666	0.540	0.847
Network	1.000	0.955	1.000	0.853	0.679	0.730	0.700	0.75[illegible]	0.688	0.564
Normal	0.919	0.837	0.965	0.924	0.733	0.686	0.73[illegible]	0.632	0.602	0.640

(purple higher horizontal dash dotted line). However, Spark speculation (black lower horizontal dash dotted line) detects stragglers (area above the speculation line and beside red dotted vertical line) after 75% tasks (red dotted vertical line) finish. In this way, Spark speculation may delay the normal execution, as it will reschedule the stragglers to other executors. Moreover, Spark speculation will cause true negative problems as shown in Figure 6, because it only checks the 25% slowest tasks. As shown in Table 4, LADRA has a better accuracy in abnormal task detection than Spark speculation for all benchmarks. However, LADRA has lower accuracy on K-Means and PageRank than WordCount and Sorting. We find that under normal execution, most tasks in the map stage or sampling stage of K-Means and PageRank have an unexpected longer duration, because these benchmarks have many iteration stages, and tasks in those stages have data skew and cross-rack traffic fetching problems. LADRA cannot detect data skew problem within normal detection results. Too many such kinds of tasks with unexpected duration will cause LADRA to report false positives.

$$TPR = \frac{TP}{(TP + FN)} \tag{16}$$

$$FPR = \frac{FP}{(TP + TN)} \tag{17}$$

7.4. LADRA's root cause analysis result

To test the accuracy of LADRA's GRNN approach for root cause analysis, we use cross validation strategy with 1/3 for test data and 2/3 for train data each time. Data in normal cases is also used in our training for improving the accuracy. In order to demonstrate the effectiveness of our approach, we run the GRNN 100 times and get the final accuracy result. We calculate the Precision (P) and True Positive Rate (TPR) for each detected root cause type by Eq. (18) and Eq. (16).

$$P = \frac{TP}{(TP + FP)} \tag{18}$$

We abandon memory root cause analysis in our experiments for three reasons. First, injecting significant memory interference into one node may cause the whole application to crash, as executors of Spark will fail if without enough memory. For instance, injected memory interference in PageRank benchmark not only causes Out-of-Memory (OOM) failures, but also makes executor keep quitting (executors are continuously restarted and fail). Secondly, memory interference does not work for non memory-intensive benchmarks. For instance, WordCount is not a memory intensive program, and it will not evoke abnormal tasks, even injecting significant memory interference. Thirdly, memory interference could also consume CPU resources, and may mislead GRNN's classifying.

Table 5 summarizes the total P and TPR results of LADRA's root cause analysis for four benchmarks. There are two issues to be noted. (1) LADRA has the highest CPU analysis precision (1.000 in CPU root cause analysis for WordCount) and higher network analysis precision (0.9545 in network root cause analysis for WordCount) results than disk I/O (0.4200 in disk I/O root cause analysis for WordCount) for three reasons. First, all four benchmarks are CPU-intensive, and require large CPU resource for computing (map and sampling stages), and network resource to transfer data (reduce stages). Secondly, abnormal tasks have longer duration after CPU interference is injected, and the impact of network injection is significant (CPU stays idle). Thus, the synthesized factors demonstrate their effectiveness. Thirdly, as disk hog is injected by leveraging a bunch of processes to read and write disk, it consumes not only disk I/O but also a certain of CPU resources. Therefore, disk I/O injections may be wrongly classified into other root causes (*e.g.*, CPU, network, or normal). (2) As shown by Table 5, LADRA is more precise on one-pass benchmarks than iterative benchmarks, such as K-means and PageRank. The TPR of k-means and PageRank's disk I/O is lower than the other two benchmarks. It is because that PageRank and k-means are not disk I/O-intensive benchmarks, if the intermediate data is small enough to be caught in memory, it will not use disk space. Therefore, the disk interference does not impact too much for these benchmarks that have small size intermediate data. Moreover, wrong classification of other root causes in k-means and PageRank also impacts LADRA's normal root cause classification, it causes more FP problems, and less TP. So the normal cases in k-means and PageRank also have lower precision and TPR. To compare with the same approach with different data size in different domains, two K-means experiments are performed on our LADRA. One uses a generated dataset by Hibench [30], and the other uses the dataset produced by a real-world CPS application. We keep all the hyper-parameter setting to be identical. Theoretically, due to the workload data distribution is different, the Spark platform will give a weakly different but similar result since data itself is not a critical role, as shown in our experiment.

To sum up, LADRA can analyze root causes via Spark log with high precision and TPR for one-pass applications. However, there may be a few of limitations for LADRA to analyze root causes by only using Spark logs. Although Spark logs contain full information, but not so rich as monitoring data.

It might be not possible to analyze all kinds of root causes by only leveraging log files. Some root causes such as code failures, resource usages, and network failures, may rely on monitoring tools. LADRA's goal is to mine useful information and leverage limited log information to analyze resource root causes without extra overhead.

8. Conclusions and future work

This paper presents LADRA, an off-line log-based root cause analysis tool to accurately detect abnormal tasks for big data platforms.LADRA can identify abnormal tasks by analyzing extracted features from Spark logs, which is more accurate than Spark's speculation-based straggler detection method. In addition, LADRA is capable of analyzing the root causes precisely using a GRNN-based method without additional monitoring. The experimental results using realistic benchmarks demonstrate that the proposed approach can accurately locate abnormalities and report their root causes. According to our experiment results, we can effectively detect the resource abnormal and analyze root causes in Spark applications.

For the future work, we will consider more complex scenarios, such as multiple interferences happening in parallel, to make our framework more robust for root cause analysis.

Acknowledgement: This work was supported in part by NSF-1836881.

References

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

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

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

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

[5] H. Zhang, H. Huang, L. Wang, Mrapid: An efficient short job optimizer on Hadoop, in: Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International, IEEE, 2017, pp. 459–468.

[6] L. Wang, S. Lu, X. Fei, A. Chebotko, H. V. Bryant, J. L. Ram, Atomicity and provenance support for pipelined scientific workflows, Future Generation Computer Systems 25 (5) (2009) 568–576.

[7] V. Subramanian, L. Wang, E.-J. Lee, P. Chen, Rapid processing of synthetic seismograms using windows azure cloud, in: The 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2010), IEEE, 2010.

[8] V. Subramanian, H. Ma, L. Wang, E.-J. Lee, P. Chen, Rapid 3d seismic source inversion using windows azure and amazon ec2, in: Proceedings of the 2011 IEEE World Congress on Services, IEEE, 2011.

[9] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, I. Stoica, Improving mapreduce performance in heterogeneous environments, in: Osdi, Vol. 8, 2008, p. 7.

[10] S. Lu, B. Rao, X. Wei, B. Tak, L. Wang, L. Wang, Log-based abnormal task detection and root cause analysis for spark, in: Web Services (ICWS), 2017 IEEE International Conference on, IEEE, 2017, pp. 389–396.

[11] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, E. Harris, Reining in the outliers in map-reduce clusters using mantri, in: OSDI, Vol. 10, 2010, p. 24.

[12] O. Ibidunmoye, F. Hernández-Rodriguez, E. Elmroth, Performance anomaly detection and bottleneck identification, ACM Computing Surveys (CSUR) 48 (1) (2015) 4.

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

[14] H. Jayathilaka, C. Krintz, R. Wolski, Performance monitoring and root cause analysis for cloud-hosted web applications, in: Proceedings of the

26th International Conference on World Wide Web, International World Wide Web Conferences Steering Committee, 2017, pp. 469–478.

[15] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, E. Brewer, Pinpoint: Problem determination in large, dynamic internet services, in: Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on, IEEE, 2002, pp. 595–604.

[16] X. Gu, H. Wang, Online anomaly prediction for robust cluster systems, in: Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on, IEEE, 2009, pp. 1000–1011.

[17] A. Oliner, J. Stearley, What supercomputers say: A study of five system logs, in: DSN, IEEE, 2007.

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

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

[20] J. Tan, S. Kavulya, R. Gandhi, P. Narasimhan, Visual, log-based causal tracing for performance debugging of mapreduce systems, in: Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on, IEEE, 2010, pp. 795–806.

[21] Q. Chen, D. Zhang, M. Guo, Q. Deng, S. Guo, Samr: A self-adaptive mapreduce scheduling algorithm in heterogeneous environment, in: Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on, IEEE, 2010, pp. 2736–2743.

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

[23] W. Qi, Y. Li, H. Zhou, W. Li, H. Yang, Data mining based root-cause analysis of performance bottleneck for big data workload, in: High Performance

Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2017 IEEE 19th International Conference on, IEEE, 2017, pp. 254–261.

[24] E. W. Fulp, G. A. Fink, J. N. Haack, Predicting computer system failures using support vector machines, WASL 8 (2008) 5–5.

[25] N. J. Yadwadkar, G. Ananthanarayanan, R. Katz, Wrangler: Predictable and faster jobs using fewer resources, in: Proceedings of the ACM Symposium on Cloud Computing, ACM, 2014, pp. 1–14.

[26] M. L. Massie, B. N. Chun, D. E. Culler, The ganglia distributed monitoring system: design, implementation, and experience, Parallel Computing 30 (7) (2004) 817–840.

[27] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, A. Muthitacharoen, Performance debugging for distributed systems of black boxes, ACM SIGOPS Operating Systems Review 37 (5) (2003) 74–89.

[28] H. Zhou, Y. Li, H. Yang, J. Jia, W. Li, Bigroots: An effective approach for root-cause analysis of stragglers in big data system, arXiv preprint arXiv:1801.03314.

[29] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, F. Özcan, Clash of the titans: Mapreduce vs. spark for large scale data analytics, Proceedings of the VLDB Endowment 8 (13) (2015) 2110–2121.

[30] S. Huang, J. Huang, J. Dai, T. Xie, B. Huang, The hibench benchmark suite: Characterization of the mapreduce-based data analysis, in: Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on, IEEE, 2010, pp. 41–51.

Byungchul Tak is currently an assistant professor at Kyungpook National University, Republic of Korea. He was a research staff member at IBM T.J. Watson Research Center, Yorktown Height, NY prior to joining the university. He received his Ph.D. in computer science in 2012 from Pennsylvania State University. He received his MS degree in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 2003, and his BS from Yonsei University, Korea in 2000. Prior to joining Pennsylvania State University, he worked as a researcher in the Electronics and Telecommunications Research Institute (ETRI), Daejeon, Korea. His research interest includes virtualization, operating systems and cloud computing. He is a member of IEEE.

Siyang Lu is Ph.D. student in the Department of Computer Science at University of Central Florida. He is working in Big Data Computing Lab in area of parallel computing framework optimization. Siyang received his Master of Science in computer science from University of Wyoming in 2015, and Master of Science in software engineering from Tianjin University in 2015. His research interests the design and analysis of parallel systems for big-data computing. For the aspect of design, he is working on optimizing performance. For the aspect of analysis, he is currently working on using log to detect abnormal, analysis root causes and performance defects in large-scale parallel computing systems

Xiang Wei: He received his bachelor's degree in computer science from Tianjin Polytechnic University in 2012, and has been taking successive postgraduate and doctoral programs of study for doctoral degree since Sep. 2012 in Beijing Jiaotong University. He had been rewarded the first-class scholarship for his good performance, and participated in two search grants from National Science Foundation of China (NSFC). His research interest focuses on intelligent information processing.

Long Wang is a Research Staff Member at the IBM T.J. Watson Research Center, Yorktown Heights, NY, where he leads the architecture of Disaster Recovery of IBM Cloud Managed Services to IBM Resiliency Services. His research interests include Fault-Tolerance and Reliability

of Systems and Applications, Dependable and Secure Systems, as well as Measurement and Assessment. He obtained his Ph.D. degree from Department of Electrical & Computer Engineering in University of Illinois at Urbana-Champaign (UIUC) in 2010. Dr. Wang is a member of the IEEE.

Liqiang Wang

Bingbing Rao

HIGHTLIGHTS

- An abnormality detection tool is proposed for log analysis, named LADRA.
- LADRA's detection approach can accurately locate where and when abnormal tasks happen.
- Effective features and abnormal factors are extracted in exposing the degree of abnormality from log analysis.
- Root causes of detected abnormal tasks are analyzed by GRNN based neural network model.
- The results are reasonable and outperform existing methods in precision.