AutoDiagn: An Automated Real-Time Diagnosis Framework for Big Data Systems

Umit Demirbaga, *Member, IEEE*, Zhenyu Wen, *Member, IEEE*, Ayman Noor, Karan Mitra, *Member, IEEE*, Khaled Alwasel, Saurabh Garg, Albert Y. Zomaya, *Fellow, IEEE*, and Rajiv Ranjan, *Senior Member, IEEE*

Abstract—Big data processing systems, such as Hadoop and Spark, usually work in large-scale, highly-concurrent, and multi-tenant environments that can easily cause hardware and software malfunctions or failures, thereby leading to performance degradation. Several systems and methods exist to detect big data processing systems' performance degradation, perform root-cause analysis, and even overcome the issues causing such degradation. However, these solutions focus on specific problems such as stragglers and inefficient resource utilization. There is a lack of a generic and extensible framework to support the real-time diagnosis of big data systems. In this article, we propose, develop and validate AutoDiagn. This generic and flexible framework provides holistic monitoring of a big data system while detecting performance degradation and enabling root-cause analysis. We present an implementation and evaluation of AutoDiagn that interacts with a Hadoop cluster deployed on a public cloud and tested with real-world benchmark applications. Experimental results show that AutoDiagn can offer a high accuracy root-cause analysis framework, at the same time as offering a small resource footprint, high throughput, and low latency.

Index Terms—Root-cause analysis, big data systems, QoS, hadoop, performance

1 Introduction

The rapid surge of data generated through sectors like social media, financial services and industries has led to the emergence of big data systems. Big data systems enable the processing of massive amounts of data in relatively short time frames. For instance, the Netflix big data pipeline processes approximately 500 billion events and 1.3 petabytes (PB) of data per day, further, during peak hours, it processes approximately 11 million events and 24 gigabytes (GB) of data on a per-second basis. Facebook has one of the largest data warehouses in the world, capable of executing more than 30,000 queries over 300 PB data every day. However, the enormousness and complexity of the big data system runs in heterogeneous computing resources, multiple tenant environments, as well as has many concurrent execution of big data processing tasks, which makes it a challenge to utilize the big data systems efficiently and reliably[1]. For example, Fig. 1 shows that the performance degrades at least 10 percent when the resources are not utilized efficiently with Setting 2.

To overcome this, it is imperative to continuously monitor and analyze all available system resources at all times in a systematic, holistic and automated manner. These resources include CPU, memory, network, I/O and the big data processing software components.

Most of the commercial [2], [3], [4] and academic big data monitoring systems mainly focus on visualizing task progress, and the system's resource utilization [5]. However, they do not focus on the interaction between multiple factors and performing root-cause analysis for performance degradation [6], [7]. Moreover, works such as [8], [9] aim to find the best parameters to optimize the performance of big data processing systems, they do not focus on the root-cause analysis that may indicate the viable reasons behind performance degradation and may provide intuitions for parameter tweaking.

Mantri [10] presents a systematic method that categorizes the main reasons causing outliers in a big data system. The authors' work was focused on the MapReduce programming framework in the Hadoop system; they do not discuss how Mantri can be applied to other big processing frameworks (e.g., Apache Spark,[1] and Apache Flink[2]). Garraghan *et al.* [11] proposed an online solution to detect long-tail

- *Umit Demirbaga is with the Newcastle University, NE1 7RU, Newcastle upon Tyne, U.K., and also with the Bartin University, Bartin 74110, Turkey. E-mail: u.demirbaga2@newcastle.ac.uk.*
- *Zhenyu Wen and Rajiv Ranjan are with the Newcastle University, NE1 7RU Newcastle upon Tyne, U.K. E-mail: {zhenyu.wen, raj.ranjan}@newcastle.ac.uk.*
- *Ayman Noor is with the Newcastle University, NE1 7RU Newcastle upon Tyne, U.K., and also with the Taibah University, Medina 42353, Saudi Arabia E-mail: anoor@taibahu.edu.sa.*
- *Karan Mitra is with the Luleå University of Technology, 971 87 Luleå, Sweden. E-mail: karan.mitra@ltu.se.*
- *Khaled Alwasel is with the Newcastle University, NE1 7RU Newcastle upon Tyne, U.K., and also with the Saudi Electronic University, Riyadh 11564, Saudi Arabia E-mail: kalwasel@gmail.com.*
- *Saurabh Garg is with the University of Tasmania, Hobart, TAS 7005, Australia. E-mail: Saurabh.Garg@utas.edu.au.*
- *Albert Y. Zomaya is with the Sydney University, Camperdown, NSW 2006, Australia. E-mail: albert.zomaya@sydney.edu.au.*

Manuscript received 8 Sept. 2020; revised 11 Mar. 2021; accepted 28 Mar. 2021.
Date of publication 2 Apr. 2021; date of current version 7 Apr. 2022.
(Corresponding author: Zhenyu Wen.)
Recommended for acceptance by W. H Robinson.
Digital Object Identifier no. 10.1109/TC.2021.3070639

[1] Online. [Available]: https://spark.apache.org/

[2] Online. [Available]: https://flink.apache.org/

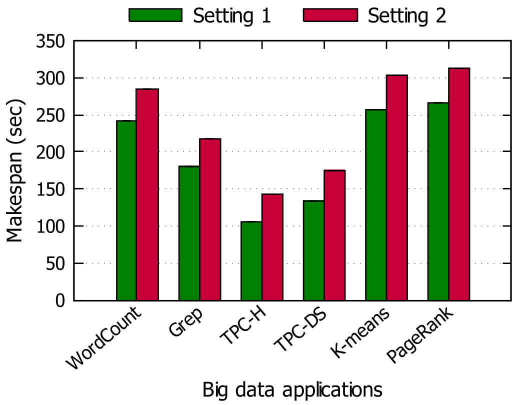


Fig. 1. Six big data applications are executed in a cloud-based Hadoop cluster with two settings: 1) the input data and jobs are allocated in the same node and 2) the input data and jobs are allocated in different nodes. In Setting 2, the execution time of each application is delayed by transmitting data across nodes.

issues in a distributed system. However, these solutions were built for specific scenarios with much scope left for analyzing a variety of problems that can exist in a large scale big data processing system.

To the best of our knowledge, there is a lack of a generic and comprehensive solution for the detection of a wide range of anomalies and performance of root-cause analysis in big data systems. Developing a general and extensible framework for diagnosing a big data system is not trivial. It requires well-defined requirements which could enable the broader adoption of root-cause analysis for the big data systems, flexible APIs to interact with an underlying monitoring system and integration of multiple solutions for detecting performance reduction problems while enabling the automatic root-cause analysis. In this paper, we tackle this research gap, and design and develop AutoDiagn to automatically detect performance degradation and inefficient resource utilization problems, while providing an online detection and semi-online root-cause analysis for a big data system. Further, it is designed as a microservice architecture that offers the flexibility to plug a new *detection and root-cause analysis module* for various types of big data systems.

The contributions of this paper are as follows:

- *An online and generic framework:* We develop a general framework called AutoDiagn which can be adapted for the detection of a wide range of performance degradation problems while pinpointing their root-causes in big data systems.
- *A case study:* We develop a novel real-time stream processing method to detect symptoms regarding outliers in a big data system. After that, we develop a set of query APIs to analyze the reasons that cause the outlier regarding a task.
- *A comprehensive evaluation:* We evaluate the feasibility, scalability and accuracy of AutoDiagn through a set of real-world benchmarks over a real-world cloud cluster.

The paper is organized as follows. The design requirements and idea are outlined in Section 2. In Section 3, we illustrate the high-level system architecture. Section 4 presents a case study that we implemented and the case study is evaluated in Section 5. Section 6 discusses the limitations of this paper and highlights our further work . Before drawing a conclusion in Section 8, we discuss the related work in Section 7.

2 Requirements and Design Idea

In this section, we analyze the key requirements of the real-time big data diagnosis system, extracting the essential features from the literature. Next, we present the key idea of the framework design.

2.1 Fundamental Prerequisite for Diagnosing Big Data Processing Systems

In order to design a generic framework for diagnosing big data processing systems, we classified the fundamental requirements of building a diagnosis system on such systems as follows:

- *Infrastructure monitoring:* Collecting the information about the underlying system, such as network conditions, CPU utilization, memory utilization, and disk I/O status.
- *Task execution monitoring:* Collecting the task information, including execution time, progress, location, location of its input data, input data size, output data size, CPU/memory usage, and process state (running, waiting, succeeded, failed, killed).
- *Abnormal behavior or fault detection:* Detecting abnormal behaviors in big data processing systems, such as slowing tasks, failed tasks, very high/low resource usage, and experiencing very high response time for the requests.
- *Root-cause analysis:* Finding the root cause of performance reduction in big data processing systems, such as the reasons why: tasks are slowing down, resource utilization is low, the response time is high, or when the network latency is high.
- *Visualization:* Visualizing the collected metrics and the results of root-cause analysis of any failures causing performance reduction in the cluster with a user-friendly interface in real-time.

2.2 Key Design Idea

Motivated by the above-mentioned requirements and inspired by medical diagnosis, we highlight the design idea of root-cause analysis for big data processing systems as shown Fig. 2, which aims to provide holistic monitoring and root cause analysis for big data processing systems. First, a set of *Symptom Detectors* is defined and developed in *Symptom Detection* to detect the abnormalities of the big system by processing collected system information stream in real-time. Once a symptom (abnormality) is detected, the *Diagnosis Management* may launch the corresponding *Diagnosers* to troubleshoot the cause of the symptom. One symptom may correspond to root causes. Finally, the decisions are made based on the root-cause analysis results.

2.3 The Generalizability of AutoDiagn

Modern big data processing systems consists of two main types: Big data analytics (e.g., Hadoop, Spark) and Stream

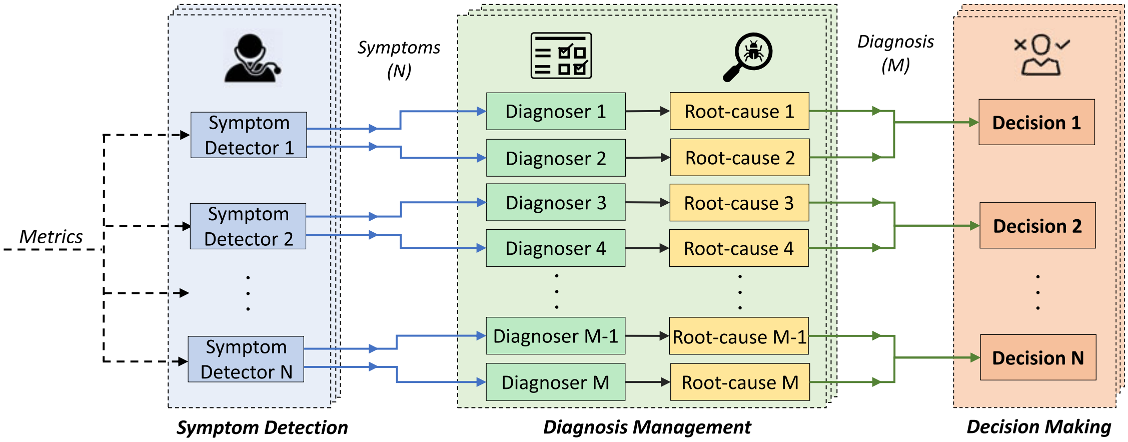


Fig. 2. The key design idea of root-cause analysis for big data processing systems.

processing (e.g., Flink, Spark Stream). Based on our design idea, our AutoDiagn is an independent framework that can be deployed alongside existing big data cluster management systems (e.g., Apache YARN), and ideally it is suitable for root-cause analysis of any big data processing system. However, for the scope of this paper and practical certainty, the implementation of AutoDiagn focuses on debugging root causes of performance degradation (e.g., slow task execution time) in Hadoop due to faults such as data locality, cluster hardware heterogeneity, and network problems (e.g., disconnection). Although we have validated the functionality of AutoDiagn in the context of Hadoop and considering different classes of workload (e.g., WordCount, Grep, TPC-H, TPC-DC, K-means clustering, PageRank), it is generalizable to other big data processing systems executing similar classes of workload.

3 AutoDiagn Architecture

Following the design idea laid out in Section 2, we introduce AutoDiagn, a novel big data diagnosing system. We first illustrate the high-level system architecture and then describe the details of each component. AutoDiagn is implemented in Java and all source code is open-source on GitHub.[3]

3.1 Architecture Overview

AutoDiagn provides a systematic solution that automatically monitors the performance of big data systems while troubleshooting the issues that cause performance reduction. Fig. 3 shows its *two* main components: *AutoDiagn Monitoring* and *AutoDiagn Diagnosing*. *AutoDiagn Monitoring* collects the defined metrics (logs) and feeds *AutoDiagn Diagnosing* with them in real-time. Once the abnormal symptoms are detected by analyzing the collected metrics, a deeper analysis is conducted to troubleshoot the cause of abnormal symptoms.

AutoDiagn Monitoring. AutoDiagn Monitoring is a decentralized real-time stream processing system that collects comprehensive system information from the big data system (e.g., Hadoop Cluster). The *Collected Metrics* is a set of pre-defined monitoring entities (e.g., CPU usage, memory usage, task location, task status) used to detect the abnormal symptoms. Moreover, the system information, required for understanding the cause of detected abnormal symptoms, is collected in this modular.

AutoDiagn Diagnosing. AutoDiagn Diagnosing is an event based diagnosing system. First, the carefully crafted metrics are injected into the *Symptom Detection Engine* which is a real-time stream processing module to detect the abnormal symptoms in a big data system. In this paper, we use the outlier which is a common symptom for performance reduction in a Hadoop cluster as a case study to demonstrate the proposed framework. Section 4.1 illustrates the details of technology that we developed for symptom detection. Moreover, our system follows the principle of modular programming; the new symptom detection method can be easily plugged in. *Diagnoser Plugins* is a component for trouble-shooting the reasons behind the detected symptom. A set of *Diagnosers* is instantiated by the *Diagnoser Manager* when their corresponding symptoms are detected. Then the instantiated *Diagnosers* query a time series database to obtain the required input and their outputs illustrate the cause of the detected symptoms.

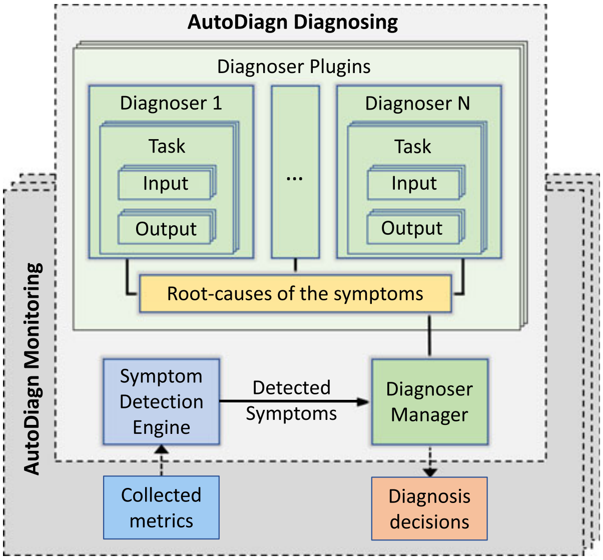


Fig. 3. The high-level architecture of the AutoDiagn system.

[3] Online. [Available]: https://github.com/umitdemirbaga/AutoDiagn

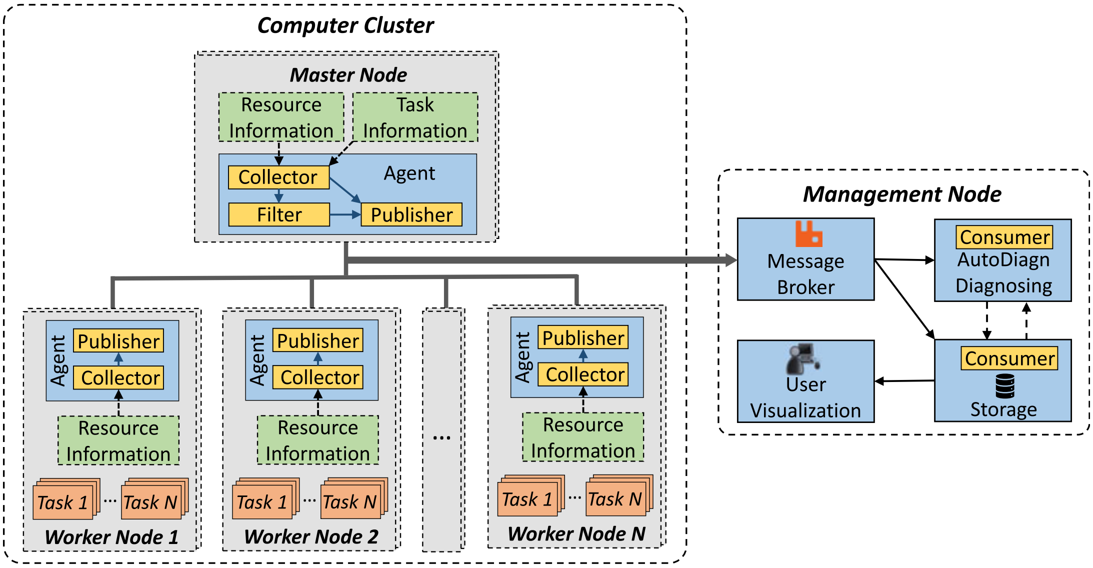


Fig. 4. The high-level architecture of the monitoring framework.

3.2 AutoDiagn Monitoring Framework

AutoDiagn monitoring framework is a holistic solution for continuous information collection in a big data cluster. The framework needs to have a fast, flexible and dynamic pipeline to transfer the collected data as well as a high performance, large scale storage system. We now describe an implementation of the framework for a big data computer cluster, and the high-level system architecture is shown in Fig. 4.

Information Collection. In each compute node, we develop and deploy an *Agent* to collect real-time system information. For the worker node, the *Agent* collects the usage of computing resource via SIGAR APIs,[4] including CPU, memory, network bandwidth, and disk read/write speeds. Moreover, the *Agent* in the master node collects the usage of computing resource as well as the job and tasks information. The *Filter* is developed by using GSon Library[5] to remove the less important information obtained from ResourceManager REST API's,[6] thereby reducing the size of data transmission. The collected information is sent to RabbitMQ[7] cluster which is a lightweight and easy-to-deploy messaging system in each time interval via *Publisher*.

Storage. The acquired information is time series data, we therefore choose InfluxDB[8] for data storage. InfluxDB is a high performance, scalable and open source time series data base which provides a set of flexible open APIs for real-time analytics. The *Consumer* subscribes the related stream topics from RabbitMQ and interacts with InfluxDB APIs to inject the information to the data base.

Interacting With AutoDiagn Diagnosing. The information required for symptom detection is directly forwarded and processed in AutoDiagn diagnosing via a *consumer*. If a symptom is detected, InfluxDB will be queried by AutoDiagn diagnosing for root-cause analysis. Finally, the analysis results are sent back to the database to be stored.

User Visualization. The user visualization allows the users to have a visible way to monitor their big data system. We utilize InfluxDB's client libraries and develop a set of RESTful APIs to allow the users to query various information, including resource utilization, job and task status, as well as root cause of performance reduction.

3.3 AutoDiagn Diagnosing Framework

In this section, we discuss the core components of the AutoDiagn Diagnosing framework (see Fig. 3), as well as the interactions with each other and the AutoDiagn Monitoring framework.

Symptom Detection Engine. The symptom detection engine subscribes a set of metrics from the real-time streaming system. Section 4.1 illustrates the technique that we developed for outlier detection. This component follows microservices architecture to which new symptom detection techniques can be directly attached to our AutoDiagn, interacting with other existing techniques to detect new symptoms.

Diagnoser Manager. The diagnoser manager is the core entity responsible for selecting the right diagnosers to find the reasons that cause the detected symptoms. Additionally, the diagnoser manager is developed as a front-end component, triggered by various detected symptoms (events) via a RESTful API, exposing all diagnosing actions within our framework. The API includes general actions such as starting, stopping or loading a diagnoser dynamically, and specific actions such as retrieving some metrics. Importantly, the diagnoser manager is able to compose a set of diagnosers to complete the diagnosing jobs that may require the cooperation of different diagnosers.

Diagnoser Plugins. The diagnoser plugin contains a set of diagnosers; and a diagnoser is the implementation of the specific logic to perform root-cause analysis of a symptom. Each diagnoser refers to a set of metrics stored in a time series database as the input of its analysis logic. Whenever it is activated by the diagnoser manager, it will perform an analysis, querying the respective metrics, executing the analytic algorithm, and storing the results. Section 4.2 discusses the algorithms to detect the outlier problems, for example,

[4] Online. [Available]: https://github.com/hyperic/sigar
[5] Online. [Available]: https://github.com/google/gson
[6] Online. [Available]: https://hadoop.apache.org/docs/r3.2.1/hadoop-yarn
[7] Online. [Available]: https://www.rabbitmq.com/
[8] Online. [Available]: https://www.influxdata.com/

TABLE 1
AutoDiagn Diagnosing Interface

Symptom Detection (High-level APIs)	**Description**
`QueryOutlier()`	Execute a `Query` that returns the list of `outliers` if any.
`QueryResourceUtil()`	Execute a `Query` that returns the list of the `worker nodes in which the computing resources are not utilized effectively` if any.
Diagnoser (High-level APIs)	**Description**
`QueryNonLocal()`	Execute a `Query` that return the list of `non-local tasks` if any.
`QueryLessResource()`	Execute a `Query` that returns `false` if the cluster is not homogeneous in terms of having resource capacity (CPU/memory).
`QueryNodeHealth()`	Execute a `Query` that returns the list of `disconnected worker nodes` in the cluster if any.
`QueryOversubscribed()`	Execute a `Query` that returns the list of the `oversubscribed tasks` if any.
`QueryDiskIOboundTasks()`	Execute a `Query` that returns the list of the `disk- or IO-bound tasks` if any.
Decision Making (High-level APIs)	**Description**
`RootcauseOutlier()`	Execute a `Query` that illustrate the main reason of the cause of the outlier.
`RootcauseResInef()`	Execute a `Query` that illustrate the main reason of the cause of inefficient resource utilization.
Information Collection (Low-level APIs)	**Description**
`taskExecTime()`	Return the `execution time` since the task started in sec.
`taskProgress()`	Return the `progress` of the running task as a percentage.
`taskInput()`	Return the `input data size` of the running task in mb.
`taskBlock()`	Return the `block id` this task process.
`taskHost()`	Return the `name of the node` this `task ran on`.
`taskCPUusage()`	Return the `CPU usage of the task`.
`taskMemoryUsage()`	Return the `memory usage of the task`.
`taskContainerCPU()`	Return the `allocated CPU` to the container this `task ran on`.
`taskContainerMemory()`	Return the `allocated memory` to the container this `task ran on`.
`blockHost()`	Return the `names of the nodes` that host the `block`.
`pendingTasks()`	Return the `number of the tasks waiting to be run`.
`nodeTotalCoreNum()`	Return the `number of the CPU core number` of the node.
`nodeCPUUsage()`	Return the `CPU utilization` of the node.
`nodeTotalMem()`	Return the `total memory capacity` of the node.
`restartedTasks()`	Return the `name of the restarted tasks` due to nodes that got disconnected from the network.
`nodeMemUsage()`	Return the `memory utilization` of the node.
`nodeDiskReadSpeed()`	Return the `disk read speed` of the node.
`nodeDiskWriteSpeed()`	Return the `disk write speed` of the node.
`nodeUploadSpeed()`	Return the `network upload speed` of the node.
`nodeDownloadSpeed()`	Return the `network download speed` of the node.

See Section 3.4 for definitions and examples.

in a Hadoop cluster. The diagnoser plugin is also designed as a microservice architecture which has two advantages: i) a new diagnoser can be conveniently plugged or unplugged on-the-fly without affecting other components; ii) new root-cause analysis tasks can be composed by a set of diagnosers via RESTful APIs.

3.4 AutoDiagn Diagnosing Interfaces for Hadoop

AutoDiagn exposes a set of simple interfaces for system monitoring, symptom detection and root-cause analysis. Table 1 shows that two types of APIs are defined: high-level APIs and low-level APIs. The high-level APIs consist of *Symptom Detection*, *Diagnoser* and *Decision Making*. The *Symptom Detection APIs* are a set of real-time stream processing functions used to detect the defined symptoms causing the performance reduction in the Hadoop system. Each *Diagnoser* is a query or a set of queries, which aim to find one of the causes of a symptom. For example, QueryNonLocal() tries to find all non-local tasks within a time interval, which is one of the reasons that causes an outlier. Finally, the *Decision Making* APIs are used to analyze the results from each *Diagnoser* and make the conclusion. These high-level APIs have to interact with the low-level APIs (*Information Collection*) to obtain system information including resource usage, and the execution information of the big data system (e.g., ask and job status in a Hadoop system). Based on this flexible design, users can define and develop their own Symptom Detection, Diagnoser and Decision Making APIs and plug them into AutoDiagn.

3.5 Example Applications

We now discuss several examples for big data system root cause applications using AutoDiagn API.

Outliers. Outliers are the tasks that take longer to finish than other similar tasks, which may prevent the subsequent tasks from making progress. To detect these tasks, the real-time stream query `QueryOutlier()` is enabled in the *Symptom Detection Engine*. This function consumes each task's completion rate (i.e., progress) and the executed time to identify the outlier tasks (detailed in Section 4.1). Next, three APIs `QueryNonlocal()`, `QueryLessResource()` and `QueryNodeHealth()`, corresponding to three *Diagnosers* that are used to analyze the reasons causing the detected symptom, are executed. `QueryNonlocal()` queries whether the input data is allocated on the node on which an outlier task is processed. In addition, `QueryLessResource()` investigates whether outlier tasks are running on the nodes that have less available resource. Moreover, `QueryNodeHealth()` examines if an outlier task is the task that is a restarted task due to the disconnected nodes from the network. Finally, `RootcauseOutlier()` is used to process the results from the three Diagnosers and make the conclusion. All the APIs are shown in Table 1 and the technical details are illustrated in Section 4.

Inefficient Resource Utilization. In our case this means that some tasks are pending (or waiting) to be on worker nodes; at the same time, some worker nodes are idle, e.g., low CPU and memory usage. There are many reasons that cause this issue, but here we consider two key causes: *task heterogeneity* and *resource heterogeneity*. The type of tasks in a big data system are various, including CPU intensive tasks, IO intensive tasks and memory intensive tasks. However, the underlying computing resources are typically equally distributed to these tasks, thereby causing inefficient resource utilization. The latter is caused by the heterogeneous underlying computing resources due to the multiple concurrent processing task environments and the queues are built on the saturated nodes.

To detect the *inefficient resource utilization* in a big data system, the real-time stream query `QueryResourceUtil()` is used within a defined time interval. We compute the mean and standard deviation of the usage resources of the whole cluster. If the standard deviation is far from the mean, we will further query whether the tasks are queued on the nodes which have high resource usage rates. If inefficient resource utilization is detected, two *Diagnosers*, `QueryOversubscribed()` and `QueryDiskIOboundTasks()`, which are the root-cause analysis APIs shown in Table 1, are executed to perform root-cause analysis. `QueryOversubscribed()` checks the type of tasks queuing on the saturated nodes. The `QueryDiskIOboundTasks()` checks whether the saturated nodes have less available computing resource, while processing the allocated tasks. The conclusion of the cause of inefficient resource utilization is made in `RootcauseResInef()`.

3.6 Parallel Execution

Following the key design idea, the diagnosers are triggered by the corresponding detected symptom. However, we are able to parallelize the execution of each symptom detector and its diagnosers by partitioning the input data. For example, if one symptom detector needs to process too many data streams, we can use two of the same instances of the symptom detector to process the data streams and aggregate the results from two symptom detectors. The diagnoser can follow the same strategy for parallel execution.

3.7 Reliability Analysis

AutoDiagn follows the centralized design for data collection, which simplifies the implementation of the *Symptom Detection*, *Diagnosis Management* and *Decision Making*. They can easily obtain the required information from one place, instead of interacting with the entire big data system. Moreover, the centralized design does not mean unreliability, due to the high-availability of RabbitMQ. The RabbitMQ cluster can overcome the node fail in the message queuing system while ensuring scalability.

TABLE 2
A Summary of Symbols Used in the Paper

Symbols	Description
J_p	Job progress
$\mathcal{N}$	Name of the task
N_l	List of $\mathcal{N}$
$\mathcal{P}$	Performance of the $\mathcal{N}$
P_l	List of $\mathcal{P}$
$\mathcal{O}$	Progress of the $\mathcal{N}$
O_l	List of $\mathcal{O}$
$\mathcal{T}$	Execution time of the $\mathcal{N}$
T_l	List of $\mathcal{T}$
m_{ed}	The performance of median task
$\mathcal{D}$	Non-local tasks
D_l	List of Non-local task
$\mathcal{R}$	Task running on the node with less resources
R_l	List of $\mathcal{R}$
$\mathcal{W}$	Restarted tasks due to the nodes' network failure
W_l	List of $\mathcal{W}$
S_l	List of outlier task
Sd	Non-local outlier
Sd_l	List of Sd
Sr	Outlier stemming from the resource variation
Sr_l	List of Sr
Sw	Outlier stemming from disconnected nodes
Sw_l	List of Sw
$\mathcal{F}$	Factor value of 1.5 used to find the $\mathcal{S}$

4 Case Study

In the previous section, we have discussed that our framework supports detection of multiple types of symptoms (e.g., outliers, inefficient resource utilization). However, detecting these symptoms is non-trivial; and each symptom can be detected by using different algorithms with different input metrics. In this section, we present a case study that details the technology of detecting outliers and the root-causes analysis for the detected outliers. The notations used in this paper are summarized in Table 2.

4.1 Symptom Detection for Outliers

Ananthanarayanan *et al.* [10] defined the outlier tasks' runtime to be 1.5 times higher than that of the median task execution time; their method is based on the assumption that all tasks are started at the same time and are the same type

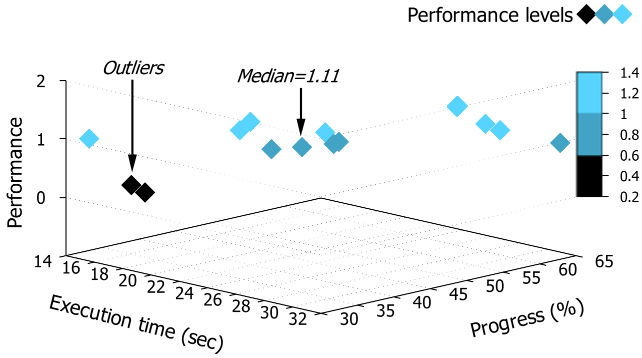


Fig. 5. Performance evaluation of the tasks.

(i.e., the same input data and the same processing code), which is not suitable for real-time symptom detection, because in a time interval the tasks may be submitted at different times; the input data size of the tasks and the code for tasks are not always the same. In this paper, we use *Performance* ($\mathcal{P}$) to measure the outlier as shown in Eq. (1). $\mathcal{O}$ represents the normalized value of the *task progress* in terms of percent work complete, and $\mathcal{T}$ is the normalized value of the task execution time

$$\mathcal{P} = \frac{\mathcal{O}}{\mathcal{T}}. \tag{1}$$

Eq. (2) is used to normalize the $\mathcal{O}$ and $\mathcal{T}$, where x_{min} and x_{max} are the minimal and maximal values of the given metrics (e.g., task progress and execution time) in a time interval. We set $b = 1$ and $a = 0.1$ to restrict the normalized values within the range from 0.1 to 1 [12]

$$x_{norm} = a + \frac{(x - x_{min})(b - a)}{x_{max} - x_{min}}. \tag{2}$$

Moreover, we define the outlier tasks which have 1.5 times less *performance* value than the median *performance* value in each time interval. Fig. 5 shows a snapshot of a time interval (e.g., three seconds), and two mappers are identified as outliers. More evaluations will be discussed in Section 5.

Algorithm 1 demonstrates the proposed *ASD* (automated symptom detection) algorithm in the AutoDiagn system. It is fed by the streaming data provided by the AutoDiagn Monitoring system during job execution. First, the performance of each running task is calculated (see Algorithm 1, Line 11) using Eq. (1). Next, the *median* value of the performance of all tasks is taken to be used to detect outliers (see Algorithm 1, Line 16). Then, the tasks whose performance is 1.5 times less than the performance of the *median* task are selected as outliers (see Algorithm 1, Line 26). As a final step, these tasks detected as outliers are sent to the *Diagnosis Generation* component for root-cause analysis (see Algorithm 1, Line 24).

4.2 Root Cause Analysis for Outliers

When the detected symptoms are passed to the *Diagnoser Manager*, the corresponding *Diagnosers* are executed for trouble-shooting. The following subsection illustrates the technologies that we have developed for analyzing the causes of outliers in a Hadoop cluster.

4.2.1 Root Cause of Outliers

In this paper, we follow the three main reasons that cause outliers, discussed in [10], i.e., Data locality, Resource heterogeneity, and Network failures.

Algorithm 1. Automated Symptom Detection for Outliers

```
Input: $J_p$ - job progress in percentage,
       $\mathcal{F}$ - factor,
       $\mathcal{N}$ - name of the running task,
       $N_l$ - list of $\mathcal{N}$,
       $\mathcal{O}$ - progress of the task,
       $O_l$ - list of $\mathcal{O}$,
       $\mathcal{T}$ - execution time of the task,
       $T_l$ - list of $\mathcal{T}$.
Output: $S_l$ - list of outliers $\mathcal{S}$.
1  // Create a list $S_l$ to store the $\mathcal{S}$
2  $S_l \leftarrow S_l[0]$
3  // Initialize the $m_{ed}$
4  $m_{ed} \leftarrow m_{ed}[0]$
5  while $J_p < 100.0$ do
6    //Clear the $S_l$ and $P_l$
7    $S_l \leftarrow$ Clear($S_l^{new}$, $S_l$)
8    $P_l \leftarrow$ Clear($P_l^{new}$, $P_l$)
9    for each $\mathcal{N}$ in $N_l$ then
10     //Compute $\mathcal{P}$
11     $\mathcal{P} = \frac{\mathcal{O}}{\mathcal{T}}$
12     //Insert the $\mathcal{P}$ into the $P_l$
13     $P_l$.add($\mathcal{P}$)
14   end
15   //Get the $m_{ed}$ from the $P_l$
16   $m_{ed} \leftarrow$ Median value of $P_l$
17   for each value of $P_l$ then
18     if ($\mathcal{P} * \mathcal{F}$) $< m_{ed}$ then
19       //Insert the $\mathcal{N}$ into the $S_l$
20       $S_l$.add($\mathcal{N}$)
21     end
22   end
23   //Update the $S_l$ in Diagnosis Generation component
24   $S_l \leftarrow$ Update($S_l^{new}$, $S_l$)
25   //Update the $N_l$, $O_l$, $T_l$, $J_p$
26   $N_l \leftarrow$ Replace($N_l^{new}$, $N_l$)
27   $O_l \leftarrow$ Replace($O_l^{new}$, $O_l$)
28   $T_l \leftarrow$ Replace($T_l^{new}$, $T_l$)
29   $J_p \leftarrow$ Replace($J_p^{new}$, $J_p$)
30 end
```

Data Locality. Hadoop Distributed File System (HDFS) stores the data in a set of machines. If a task is scheduled to a machine which does not store its input data, moving data over the network may introduce some overheads to cause the outliers issue.

Resource Heterogeneity. The machines in a Hadoop cluster may be homogeneous with the same hardware configuration, but the run-time computing resources are very heterogeneous due to the multiple talents environment, multiple concurrent processing task environment, machine failures, machine overloaded etc. If a task is scheduled to a bad machine (e.g., has less computing resource) it may cause an outlier issue. Moreover, resource management systems for a large-scale cluster like YARN split the tasks over the nodes equally without considering the resource capacities of the nodes in the cluster, but only takes into account sharing the node's resources among the tasks running on the node equally by default [13]. That is more likely to raise an outlier problem in the cluster.

Network Failure. In Hadoop clusters, the network disconnection can cause the running tasks allocated on a disconnected node to be restarted on other nodes, which may lead to the task becoming an outlier and, increase the completion time. The following illustrates the three algorithms that we developed to identify the outliers caused by the three reasons.

4.2.2 Detecting Data Locality Issues

We assume that a *non-local* task ($\mathcal{D}$) (e.g., mapper) is executed on a node where its input data is not stored (In the following, we use *Sd* to represent *non-local outliers*). To detect these tasks, we develop Algorithm 2 to check whether a set of outliers is caused by a data locality issue. The input of our algorithm is a list of detected outliers during the time interval from t to $t+1$ and one of its outputs is a list of outliers which also belongs to the *non-local* tasks. First, we query our time series database to obtain all *non-local* tasks within the given time interval (see Algorithm 2, Line 2).

Here, `QueryNonLocal()`, a root-cause analysis API, is used to find the non-local ones among the running tasks in that period of time. It compares the location where the task is running (host node of the task) with the nodes where the data block is replicated for fault tolerance via information collection APIs shown in Table 1, `taskHost()` and `blockHost()`. If the task is not running on any of these nodes (nodes hosting a copy of the block), this task is marked as a non-local task. In the second step (Algorithm 2, Line 4), we obtain the common elements of list D_l and S_l. These elements symbolize the non-local outliers stemming from a data locality issue.

4.2.3 Detecting Resource Heterogeneity Issues

Algorithm 2 is designed to identify the outliers caused by the resource heterogeneity. The tasks running on the nodes which have less computing resource ($\mathcal{R}$) tend to be outliers [14] (in the following, we use *Sr* to represent *outliers running on the nodes which have less computing resource*). In Algorithm 2, the list of detected outliers during the time interval from t to $t+1$ is used as input and one of the outputs of the algorithm is a list of outliers which also belongs to the tasks running on the node with less computing resource. The time series database is queried to obtain all *the tasks running on the node with less computing resource* within the given time interval (see Algorithm 2, Line 6).

Here, `QueryLessResource()`, a root-cause analysis API, is used to check the heterogeneity of the nodes that host only the running tasks based on the resource specifications of them in that period of time. It detects the nodes with less resource capacity in terms of CPU core numbers and the total amount of memory among the nodes hosting the running tasks. The resource specifications of the nodes (i.e., CPU core numbers, total amount of memory) are obtained from each node via information collection APIs shown in Table 1, `nodeTotalCoreNum()` and `nodeTotalMem()` APIs. As a second step (Algorithm 2, Line 8), we obtain the common elements of list R_l and S_l. These elements symbolize the outliers stemming from a cluster heterogeneity issue.

4.2.4 Detecting Network Failure Issues

Since S_l is obtained from Algorithm 1, a Diagnoser is executed via `QueryNodeHealth()` to find all restarted tasks due to the nodes disconnected by network failure within the given time interval (see Algorithm 2, Line 10). The low-level API `restartedTasks()` is called which distinguishes the restarted tasks due to network failure from the speculation of straggler tasks by analyzing the information of the tasks that is provided by the monitoring agent. Thereafter, we compute the list Sw_l that contains the outlier tasks caused by the network failure (see Algorithm 2, Line 12).

Algorithm 2. Root-Cause Analysis of Outliers

```
   Input: $S_l$ - list of outliers in time interval from $t$ to $t+1$
   Output: $Sd_l$ - list of non-local outliers $Sd$,
           $Sr_l$ - list of outliers stemming from resource variation $Sr$,
           $Sw_l$ - list of outliers stemming from disconnected
                    nodes $Sw$.
 1 // Find all $\mathcal{D}$ within the given time interval
 2 $D_l \leftarrow$ QueryNonLocal(t, t+1)
 3 //Find the common elements in the $D_l$ and $S_l$, and add
   them into the $Sd_l$
 4 $Sd_l \leftarrow$ RetainAll($D_l$, $S_l$)
 5 // Find all $\mathcal{R}$ within the given time interval
 6 $R_l \leftarrow$ QueryLessResource(t, t+1)
 7 //Find the common elements in the $R_l$ and $S_l$, and add
   them into the $Sl_l$
 8 $Sr_l \leftarrow$ RetainAll($R_l$, $S_l$)
 9 // Find all $\mathcal{W}$ within the given time interval
10 $W_l \leftarrow$ QueryNodeHealth(t, t+1)
11 //Find the common elements in the $W_l$ and $S_l$, and add
   them into the $Sw_l$
12 $Sw_l \leftarrow$ RetainAll($W_l$, $S_l$)
```

4.2.5 Decision Making

In this case study, we use a simple decision make method that compares the lists Sd_l, Sr_l and Sw_l and the probability of the reasons causing the outliers by using the number of the elements of a list divided the total number of outlier tasks. For instance, the probability of the performance reduction caused by data locality is $\frac{|Sd_l|}{|S_l|}$.More advanced methods such as deep learning models can be used for processing more complicated decision making tasks in future work.

5 Evaluation

In this section, we present a comprehensive evaluation showing the capacity and the accuracy rate of AutoDiagn, as well as a analysis of its resource consumption and overheads.

5.1 Experimental Setup

Environments. We set up the Hadoop YARN clusters over 31 AWS nodes with 1 master and 30 slaves with the Operating

TABLE 3
The Accuracy of Symptom Detection for Non-Local Outliers in a Homogeneous Cluster

Benchmark	Total tasks	$\mathcal{D}$	Outliers (detected as *Sd*)	Accuracy (%)	Error (σ)
WordCount	234	32	29	90.63	3.9
Grep	236	37	33	89.19	4.8
TPC-H	102	13	12	92.31	6.72
TPC-DS	126	13	12	92.31	6.1
K-means	234	34	29	85.29	1.25
PageRank	235	28	25	89.29	6.2

TABLE 4
The Accuracy of Symptom Detection for the Outliers Stemming From Resource Variation in a Heterogeneous Cluster

Benchmark	Total tasks	$\mathcal{R}$	Outliers (detected as *Sr*)	Accuracy (%)	Error (σ)
WordCount	234	37	33	89.19	2.77
Grep	236	26	24	92.31	4.77
TPC-H	102	9	8	88.89	5.47
TPC-DS	126	13	12	92.31	6.9
K-means	234	36	33	91.67	2.88
PageRank	235	30	28	93.33	5.35

system of each node being Ubuntu Server 18.04 LTS (HVM). The Hadoop version is 3.2.1 and the Hive version is 3.1.1. To meet our experimental requirements, we built two types of cluster. In *Type I* each node has the same configuration (i.e., 4 cores and 16 GB memory). In *Type II*, 25 nodes have 4 cores and 16 GB memory and 6 nodes have 2 cores and 4 GB memory.

Benchmarks and Workload. We used six well-known Hadoop benchmarks in our evaluations namely: WordCount,[9] Grep,[10] TPC-H,[11] TPC-DS,[12] K-means clustering,[13] and PageRank.[14] The input of each benchmark application is 30GB.

Methodology. Our experiments aim to evaluate the effectiveness of AutoDiagn. To this end, we manually inject the above-mentioned three main reasons to cause the outliers, which can be summarized as three types of execution environment. **Env** $\mathcal{A}$: we perform all benchmark experiments in the cluster *Type I*. **Env** $\mathcal{B}$: we perform all benchmark experiments in the cluster *Type I*, but skew the input size stored on different nodes. **Env** $\mathcal{C}$: we perform all benchmark experiments in the cluster *Type II* (a heterogeneous cluster). **Env** $\mathcal{H}$: we perform all benchmark experiments in the cluster *Type I*, and disconnect some nodes' network during execution. Each benchmarking is repeated 5 times and results are reported as the average and standard deviation. In total, there are 90 experiments conducted in our evaluation.

5.2 Diagnosis Detection Evaluation

In this section, we evaluate the accuracy of our symptom detection method. To this end, we execute our benchmarks in **Env** $\mathcal{B}$ to increase number of *Sd* tasks (see Section 4.2.2). Next, to increase the issue of resource heterogeneity (*Sr* referring to Section 4.2.3), we run the benchmarks in **Env** $\mathcal{C}$. Thereafter, we run the benchmarks in **Env** $\mathcal{H}$ to emulate the network failure (*Sw* referring to Section 4.2.4). Finally, we compare the detected Outlier tasks with the ground truths that are the data locality, resource heterogeneity, and network failure issues observed by the AutoDiagn diagnosing system.

TABLE 5
The Accuracy of Symptom Detection for the Outliers Stemming From Network Failures

Benchmark	Total tasks	$\mathcal{W}$	Outliers (detected as *Sw*)	Accuracy (%)	Error (σ)
WordCount	234	11	10	90.91	1.83
Grep	236	13	12	92.31	6.73
TPC-H	102	13	12	92.31	6.54
TPC-DS	126	15	14	93.33	5.43
K-means	234	17	16	94.12	4.33
PageRank	235	19	18	94.74	4.23

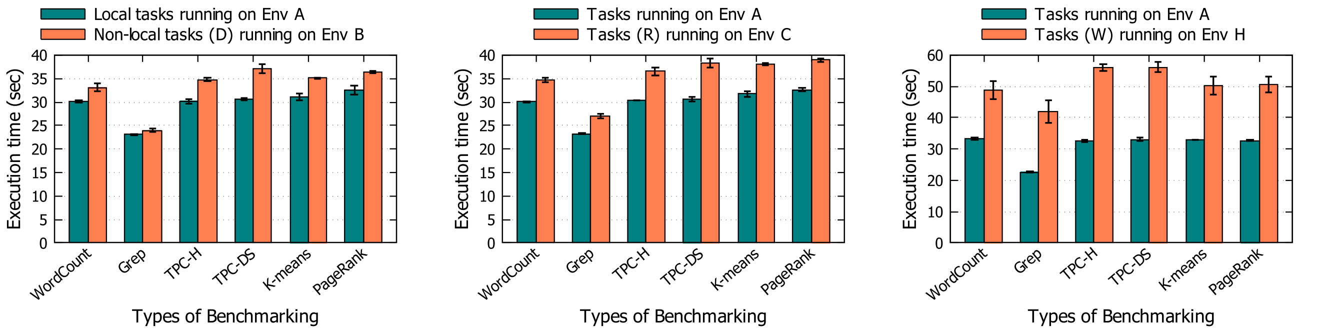
Tables 3, 4, and 5 summarize all the results. All benchmarks achieve high accuracy by using our proposal symptom detection method. The highest accuracy for both *Sd* and *Sr* are 92.3 percent, and for *Sw* is 94.7 percent and the overall accuracy for outlier detection is 91.3 percent, where the *Error* represents the variation of the accuracy depending on the repeated experiments.

We compute the accuracy of our symptom detection method by using the number of detected outlier tasks divided by the *actual* number of the tasks that can cause the outlier issue. Table 3, for example, $\mathcal{D}$ is the total number of non-local tasks and Outliers (*Sd*) is the number of detected outlier tasks that belong to non-local task. Therefore, the accuracy is $\frac{Sd}{\mathcal{D}}$. Tables 4 and 5 follow the same approach to compute the accuracy.

*Outlier Verification.*To further verify the *Sd*, *Sr*, and *Sw* are the main reasons causing the outliers, we conduct the following comparison experiments: 1) comparing the execution time of local tasks and non-local tasks; 2) comparing the execution time of the tasks running in **Env** $\mathcal{A}$ and **Env** $\mathcal{C}$; and 3) comparing the execution time of normal tasks and restarted tasks due to network failure. Fig. 6a proves that non-local tasks consume more time than local tasks due to the overload introduced by data shuffling. Additionally, we compare the throughput of the local tasks and non-local tasks in terms of how much data can be processed in each second. Fig. 7 reveals that the throughput of non-local tasks is only 70 percent that of local tasks.

Moreover, Fig. 6b shows that the execution time of the tasks running on **Env** $\mathcal{A}$ is less than that on **Env** $\mathcal{C}$. This is because the tasks are equally distributed to all computing

[9] Online. [Available]: http://wiki.apache.org/hadoop/WordCount
[10] Online. [Available]: http://wiki.apache.org/hadoop/Grep
[11] Online. [Available]: http://www.tpc.org/tpch/
[12] Online. [Available]: http://www.tpc.org/tpcds/
[13] Online. [Available]: https://en.wikipedia.org/wiki/K-means_clustering
[14] Online. [Available]: https://en.wikipedia.org/wiki/PageRank



(a) Local tasks vs Non-local tasks

(b) Homogeneous cluster vs Heterogeneous cluster

(c) Normal tasks vs Restarted tasks caused by network failure

Fig. 6. Comparison of execution time of the tasks.

nodes and the less powerful nodes are saturated. Furthermore, Fig. 9a shows that the CPU usage of less powerful hosts reaches 100 percent, thereby building a task queue in these hosts, increasing the overall execution time. However, Fig. 9b reveals that the powerful hosts have sufficient computing resources for processing the allocated tasks.

Furthermore, Fig. 6c shows that the execution time of the restarted tasks are longer than the normal tasks. As Fig. 8 illustrates, we compute the execution time of the restarted task by adding the execution time of the task in the disconnected node and that in the rescheduled node.

5.3 Performance and Overheads

Performance Evaluation. We evaluate the performance of AutoDiagn by measuring the end-to-end response time of symptom detection and root-cause analysis. Since they are not affected by the types of benchmark, we report the average of the response time. Fig. 10a shows that the real-time symptom detection can achieve a low response time, which only has 96 milliseconds and 1,059 milliseconds with 100 tasks and 1,000 tasks, respectively. Although the response time increases linearly, the parallel execution method discussed in Section 3.6 can be applied to reduce the latency. The response time for root cause analysis is higher than that of symptom detection. For 100 tasks and 1,000 tasks, their response times are 0.354 seconds and 5.974 seconds, respectively. Unlike the symptom detection which is very sensitive to latency because of the follow-up processes, triggering the further root-cause analysis or alerting the system managers, Root-cause analysis aims to provide a holistic diagnosing of a big system and the analysis results may help to improve the system performance in future. As a result, the real-time root-cause analysis is not compulsory.

System Overheads. To evaluate the system overhead introduced by AutoDiagn, we measure the CPU and memory usage of AutoDiagn Monitoring (agent) and AutoDiagn Diagnosing. Table 6 shows that -*AutoDiagn Monitoring* only consumes approximately 2.52 percent memory and 4.69 percent CPU; while -*AutoDiagn Diagnosis* uses 2.08 percent memory and 3.49 percent CPU.

Fig. 10b shows the network overhead of AutoDiagn. The extra communication cost introduced by our tool is small but it increases when the number of parallel tasks increases. For example, when the number of parallel task is 100, there are about 45 messages per second sent from agents to RabbitMQ cluster and the total size of these messages is 13.5 KB/s. The message rate and network overhead increase to 615 per second and 223 KB/s, respectively, when the number of parallel tasks is 1,000.

Storage Overheads. AutoDiagn needs to dump the system information to a database which may consume extra storage resource. *In our evaluation experiments, it only cost 3.75 MB disk space in total*. Obviously, increasing the types of symptom detection and root cause analysis will also consume more storage resources. We discuss the potential future work in Section 6.

6 Discussion and Future Work

Populating Applications. In this paper, we propose a general and flexible framework to uncover the performance reduction issues in a big data system. In particular, we develop and evaluate big data applications for outliers. New applications (including symptom detection and root-cause analy-

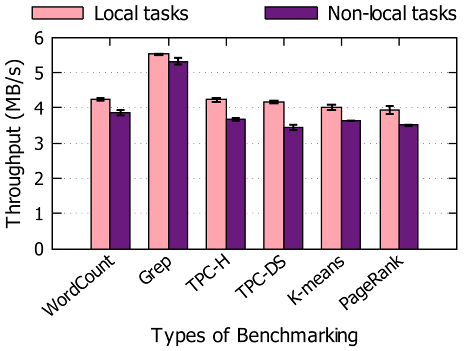


Fig. 7. The throughput of AutoDiagn.

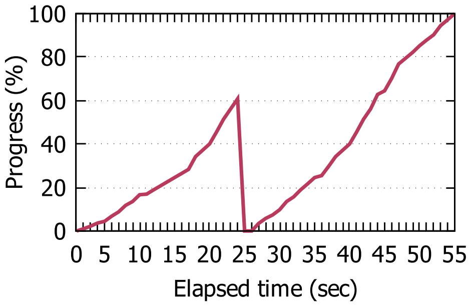
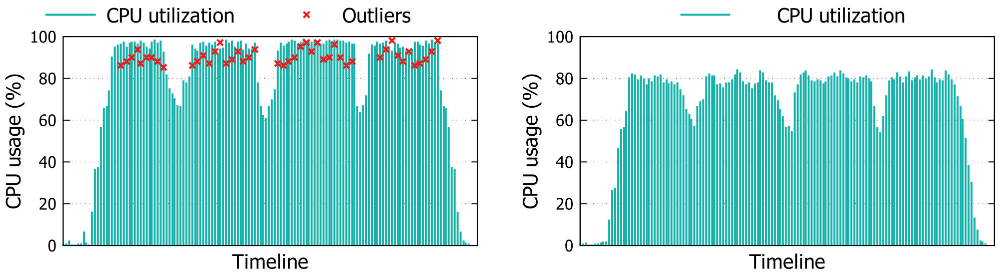
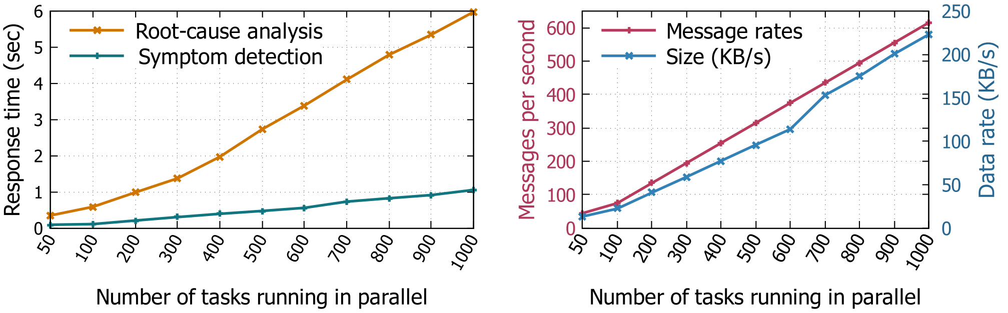


Fig. 8. The life cycle of the restarted task.



(a) CPU utilization of less powerful hosts and outliers (b) CPU utilization of high power hosts

Fig. 9. CPU utilization of two nodes running simultaneously. Outliers are most likely to occur in the nodes which have less computing resource.



(a) The end-to-end response time of AutoDiagn diagnosis system (b) The message rates and network overhead

Fig. 10. Performance evaluation and network overhead of AutoDiagn.

sis) are required to populate our system for future work.

Overhead Cost Reduction. Our system is designed in a loosely-coupled manner, the processing components can be easily scaled. However, the storage overhead increases with the number of applications increasing. [15] proposed a caching method to aggregate the information before sending to destination nodes. We will explore this direction in future work to reduce the storage overhead and network overhead.

Performance Improvement. Mantri [10] utilized the outputs of the root cause analysis to improve the resource allocation in Hadoop clusters. Thus, one open research direction is to build a system which can react to analysis results, thereby improving the performance of the big data system.

7 Related Work

Much recent work in big data systems focuses on improving workflows [16], [17], [18], programming framework [19], [20], [21], task scheduling [22], [23], [24].

Root-Cause Analysis. There is a large volume of published studies describing the role of root-cause analysis. The authors of [10], [25], [26] take the next step of understanding the reasons for performance reduction. Mantri [10] characterizes the prevalence of stragglers in Hadoop systems as well as troubleshooting the cause of stragglers. Dean and Barroso [25] analyze the issues causing tail latency in big data systems. Garraghan *et al.* [11], [27] proposed a new method to identify long tail behavior in big data systems and evaluated in google data trace. The authors in [28] use offline log analysis methods to identify the root cause of outliers in a large-scale cluster consisting of thousands of nodes by tracking the resource utilization. Similarly, Zhou *et al.* [29] use a simple but efficient rule based method to identify the root cause of stragglers.

TABLE 6
Resource Overhead Caused by AutoDiagn Components

Components	Mem (%)	CPU (%)
AutoDiagn Monitoring	2.52	4.69
AutoDiagn Diagnosing	2.08	3.49

Along with these similar works, there are some researchers using statistical and machine learning methods for root-cause analysis. The authors of [30] introduce a Regression Neural Network (RNN) based algorithm to trouble-shoot the causes of stragglers by processing Spark logs. More algorithms such as the associated tree and fuzzy data envelopment analysis [31] and Reinforcement Learning [32] are applied for finding the reasons of stragglers in Hadoop and Spark.

In [33], a Pearson coefficient of correlation is used for root cause analysis to measure linear correlation between system metrics, workload and latency. However, these works lack a systematic solution for root cause analysis for big data processing systems and the proposed methods are not applicable for real-time systems.

Different to other work, the authors of [34] propose a new algorithm that aims to reduce the proportion of straggler tasks in machine learning systems that use gradient-descent-like algorithms. This work offers an idea to develop new Diagnosers for machine learning systems using our framework.

Anomaly Detection and Debugging. The authors in [35] propose a rule-based approach to identify anomalous behaviors in Hadoop ecosystems by analyzing the task logs. This work

TABLE 7
The Features Supported by Existing Tools and AutoDiagn

Feature	DataDog [2]	Sequence IQ [3]	Sematext [4]	TACC [5]	Mantri [10]	DCDB [39]	Nagios [41]	Ganglia [42]	Chukwa [43]	DMon [44]	AutoDiagn
Real-time monitoring	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Near real-time	Yes	Near real-time	Yes
Root-cause analysis	No	No	No	No	Yes	Yes	No	No	No	Yes	Yes
BigData frameworks support	Good	Poor	Good	No	Poor	No	Poor	Poor	Poor	Good and Extensible	Good and Extensible
Underlying resource monitoring	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Real-time monitoring for big data tasks	Yes	Yes	Yes	No	Yes	No	No	No	Yes	Yes	Yes
Auto-scaling	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	Yes
Alerts	Yes	No	Yes	No	No	No	Yes	No	No	No	Yes
Visualization of big data tasks	Yes	No	Yes	No	No	No	No	Yes	No	No	Yes
User customized root-cause analysis	No	No	No	No	No	No	No	No	No	No	Yes

only analyzes the task logs, which fails to capture the performance reduction issues caused by inefficient utilizing the underlying resources. Next, Khoussainova *et al.* [36] build a historical log analysis system to study and track the MapReduce jobs which cause performance reduction based on their relevance, precision and generality principles. However, this cannot be performed for real-time anomaly detection. Du *et al.* [37] train a machine learning model from the normal condition data by using Long Short-Term Memory (LSTM) and this trained model is used for detecting in Hadoop and OpenStack environments. Our AutoDiagn provides infrastructure into which the trained models can be plugged to enrich the applications.

Real-Time Operational Data Analytic System. Agelastos *et al.* [38] propose a monitoring system for HPC systems, which can capture the cases of applications competing for shared resources. However, this system does not consider root-cause analysis of the performance reduction. The authors of [5], [39] do not only provide the feature of real-time monitoring, but are also able to identify the performance issues and trouble-shoot the cause of the issues. In addition to them, [40] uses a type of artificial neural network called autoencoder for anomaly detection. They first monitor the system in real-time and collect the normal data for training the model used to discern between normal and abnormal conditions in an online fashion. However, these systems are developed for HPC clusters and are not suitable for big data systems.

Table 7 presents a brief overview of various monitoring tools for big data frameworks.

8 Conclusion

In this paper, we have presented AutoDiagn, a framework for enabling diagnosing of large-scale distributed systems to ascertain the root cause of outliers, with the core purpose of unravelling the concretization of complicated models for system management. After making a comprehensive literature review and identifying the requirements for real-world problems, we conceived its design. The combination of user-defined functions powered by APIs and the agent-based monitoring system along with the findings obtained from an empirical analysis of the experiments we conducted play a fundamental role in the development of the system. AutoDiagn can be applied to most big data systems along with the monitoring systems. We have also presented the implementation and integration of the AutoDiagn system to the SmartMonit [45], real-time big data monitoring system, combined in our production environment. In our implementation on a large cluster, we find AutoDiagn very effective and efficient.

Outliers are one of the main problems in big data systems that overwhelm the whole system and reduce performance considerably. AutoDiagn embraces this problem to reveal the bottlenecks alongside their root causes.

Acknowledgments

This work was supported in part by the Turkish Ministry of National Education, in part by the following UKRI projects through SUPER under Grant EP/T021985/1, through PACE under Grant EP/R033293/1, and through Centre for Digital Citizens under Grant EP/T022582/1, in part by the National Natural Science Foundation of China under Grant 62072408, and in part by the Zhejiang Provincial Natural Science Foundation of China under Grant LY20F020030.

References

[1] A. Noor *et al.*, “Cyber-physical application monitoring across multiple clouds,” *Comput. Elect. Eng.*, vol. 77, pp. 314–324, Jul. 2019.

[2] Datadog, Accessed: Jul. 13, 2020. [Online]. Available: https://www.datadoghq.com/

[3] Sequenceiq, Accessed: Jul. 14, 2020. [Online]. Available: https://github.com/sequenceiq

[4] Sematext, Accessed: Jul. 13, 2020. [Online]. Available: https://sematext.com/

[5] R. T. Evans, J. C. Browne, and W. L. Barth, “Understanding application and system performance through system-wide monitoring,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2016, pp. 1702–1710.

[6] G. Iuhasz, D. Pop, andI. Dragan, “Architecture of a scalable platform for monitoring multiple big data frameworks,” *Scalable Comput., Pract. Experience*, vol. 17, no. 4, pp. 313–321, 2016.

[7] I. Drăgan, G. Iuhasz, andD. Petcu, “A scalable platform for monitoring data intensive applications,” *J. Grid Comput.*, vol. 17, no. 3, pp. 503–528, 2019.

[8] S. Babu, “Towards automatic optimization of MapReduce programs,” in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 137–142.

[9] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, “Shark: SQL and rich analytics at scale,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 13–24.

[10] G. Ananthanarayanan, *et al.*, “Reining in the outliers in map-reduce clusters using Mantri,” in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 265–278.

[11] P. Garraghan, X. Ouyang, P. Townend, and J. Xu, “Timely long tail identification through agent based monitoring and analytics,” in *Proc. IEEE 18th Int. Symp. Real-Time Distrib. Comput.*, 2015, pp. 19–26.

[12] J. Han, J. Pei, and M. Kamber, *Data Mining: Concepts And Techniques*. Amsterdam, The Netherlands: Elsevier, 2011.

[13] T. Renner, L. Thamsen, and O. Kao, “CoLoc: Distributed data and container colocation for data-intensive applications,” in *Proc. IEEE Int. Conf. Big Data*, 2016, pp. 3008–3015.

[14] A. Rasooli and D. G. Down, “Guidelines for selecting Hadoop schedulers based on system heterogeneity,” *J. Grid Comput.*, vol. 12, no. 3, pp. 499–519, Jul. 2014.

[15] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman, “Aggregation and degradation in Jetstream: Streaming analytics in the wide area,” in *Proc. 11th USENIX Symp. Netw. Syst. Des. Implementation (NSDI)*, 2014, pp. 275–288.

[16] Z. Wen *et al.*, “GA-PAR: Dependable microservice orchestration framework for geo-distributed clouds,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 1, pp. 129–143, Jan. 2020.

[17] Z. Wen, J. Cała, P. Watson, and A. Romanovsky, “Cost effective, reliable and secure workflow deployment over federated clouds,” *IEEE Trans. Serv. Comput.*, vol. 10, no. 6, pp. 929–941, Nov.–Dec. 2017.

[18] Z. Wen, R. Qasha, Z. Li, R. Ranjan, P. Watson, and A. Romanovsky, “Dynamically partitioning workflow over federated clouds for optimising the monetary cost and handling run-time failures,” *IEEE Tran. Cloud Comput.*, vol. 8, no. 4, pp. 1093–1107, Oct.–Dec. 2020.

[19] G. Malewicz, *et al.*, “Pregel: A system for large-scale graph processing,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.

[20] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” in *Proc. 2nd USENIX Conf. Hot Top. Cloud Comput.*, 2010, pp. 1–7.

[21] M. Abadi *et al.*, “Tensorflow: A system for large-scale machine learning,” in *Proc. 12th USENIX Symp Operating Syst. Des. Implementation*, 2016, pp. 265–283.

[22] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, “Quincy: Fair scheduling for distributed computing clusters,” in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Princ.*, 2009, pp. 261–276.

[23] N. J. Yadwadkar and W. Choi, “Proactive straggler avoidance using machine learning,” Univ. Berkeley, Berkeley, CA, USA, White Paper, 2012.

[24] A. Badita, P. Parag, and V. Aggarwal, “Optimal server selection for straggler mitigation,” *IEEE/ACM Trans. Netw.*, vol. 28, no. 2, pp. 709–721, Apr. 2020.

[25] J. DeanandL. A. Barroso, “The tail at scale,” *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[26] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, “Making sense of performance in data analytics frameworks,” in *Proc. 12th USENIX Symp. Netw. Syst. Des. Implementation*, 2015, pp. 293–307.

[27] P. Garraghan, X. Ouyang, R. Yang, D. McKee, and J. Xu, “Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters,” *IEEE Trans. Serv. Comput.*, vol. 12, no. 1, pp. 91–104, Jan.–Feb. 2019.

[28] X. Ouyang, P. Garraghan, R. Yang, P. Townend, and J. Xu, “Reducing late-timing failure at scale: Straggler root-cause analysis in cloud datacenters,” in *Proc. Fast Abstr. 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01316515

[29] H. Zhou, Y. Li, H. Yang, J. Jia, and W. Li, “Bigroots: An effective approach for root-cause analysis of stragglers in big data system,” *IEEE Access*, vol. 6, pp. 41 966–41 977, 2018.

[30] S. Lu, X. Wei, B. Rao, B. Tak, L. Wang, and L. Wang, “LADRA: Log-based abnormal task detection and root-cause analysis in big data processing with spark,” *Future Gener. Comput. Syst.*, vol. 95, pp. 392–403, Jun. 2019.

[31] Z. He, Y. He, F. Liu, and Y. Zhao, “Big data-oriented product infant failure intelligent root cause identification using associated tree and fuzzy DEA,” *IEEE Access*, vol. 7, pp. 34 687–34 698, 2019.

[32] H. Du and S. Zhang, “Hawkeye: Adaptive straggler identification on heterogeneous spark cluster with reinforcement learning,” *IEEE Access*, vol. 8, pp. 57 822–57 832, 2020.

[33] J. P. Magalhães and L. M. Silva, “Root-cause analysis of performance anomalies in web-based applications,” in *Proc. ACM Symp. Appl. Comput.*, 2011, pp. 209–216.

[34] R. Bitar, M. Wootters, and S. El Rouayheb, “Stochastic gradient coding for straggler mitigation in distributed learning,” *IEEE J. Sel. Areas Inf. Theory*, vol. 1, no. 1, pp. 277–291, May 2020.

[35] A. M. Chacko, J. S. Medicherla, and S. M. Kumar, “Anomaly detection in MapReduce using transformation provenance,” in *Proc. Adv. Big Data Cloud Comput.*, 2018, pp. 91–99.

[36] N. Khoussainova, M. Balazinska, and D. Suciu, “PerfXplain: Debugging MapReduce job performance,” 2012, *arXiv:1203.6400*.

[37] M. Du, F. Li, G. Zheng, and V. Srikumar, “Deeplog: Anomaly detection and diagnosis from system logs through deep learning,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 1285–1298.

[38] A. Agelastos *et al.*, “The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications,” in *Proc. Int. Conf. High Perform. Comput., Netwo., Storage Anal.*, 2014, pp. 154–165.

[39] A. Netti, *et al.*, “DCDB wintermute: Enabling online and holistic operational data analytics on HPC systems,” in *Proc. 29th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2020, pp. 101–112.

[40] A. Borghesi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini, “Anomaly detection using autoencoders in high performance computing systems,” in *Proc. AAAI Conf. Artif. Intell.*, 2019, vol. 33, pp. 9428–9433.

[41] Nagios. Accessed: Jul. 15, 2020. [Online]. Available: https://www.nagios.org/

[42] Ganglia. Accessed: Jul. 15, 2020. [Online]. Available: http://ganglia.info/

[43] Apache chukwa. Accessed: Jul. 14, 2020. [Online]. Available: https://chukwa.apache.org/

[44] Dmon. Accessed: Jul. 12, 2020. [Online]. Available: https://github.com/Open-Monitor/dmon

[45] U. Demirbaga, A. Noor, Z. Wen, P. James, K. Mitra, and R. Ranjan, “SmartMonit: Real-time big data monitoring system,” in *Proc. 38th Symp. Reliable Distrib. Syst.*, 2019, pp. 357–359.

Umit Demirbaga (Member, IEEE) received the BSc degree in electronics and computer education from Marmara University, Turkey, in 2011, and the MSc degree in 2017, in computer science from Newcastle University, U.K., where he is currently working toward the PhD degree. His research interests include big data analytics, cloud computing, and distributed systems. He was the recipient of the Outstanding Performance Award with Best Team Project Award in the MSc in 2017.

Zhenyu Wen (Member, IEEE) received the MSc and PhD degrees in computer science from Newcastle University, Newcastle upon Tyne, U.K., in 2011 and 2016, respectively. He is currently a postdoc researcher with the School of Computing, Newcastle University, U.K. His current research interests include IoT, crowd sources, AI system, and cloud computing, which include scalable data management for the Internet of Things. He was the recipient of the IEEE TCSC Award for Excellence in Scalable Computing (Early Career Researchers) in 2020.

Ayman Noor received the bachelor's degree in computer science from the College of Computer Science and Engineering, Taibah University, Madinah, SA, in 2006 and the MSc degree in computer and information science from Gannon University, PA, USA, in 2013. He is currently working toward the PhD degree in computer science with Newcastle University, U.K. His current research interests include cloud computing, monitoring, and machine learning.

Karan Mitra (Member, IEEE) is currently an assistant professor with the Luleå University of Technology, Sweden. He received the Dual-badge PhD from Monash University, Australia, and the Luleå University of Technology in 2013. His research interests include cloud and mobile cloud computing, performance benchmarking of distributed systems, context-aware computing, and QoE. He is a member of the ACM.

Khaled Alwasel received the BS degree in information technology from Indiana University-Purdue University Indianapolis, in 2014 and the MS degree in information technology from Florida International University, USA, in 2015. He is currently working toward the PhD degree with the School of Computing Science, Newcastle University, U.K. His research interests include the areas of software-defined networking (SDN), big data, IoT, edge computing, and cloud computing.

Saurabh Garg is currently a lecturer at the University of Tasmania, Hobart, Tasmania. He has authored or coauthored more than 30 papers in highly cited journals and conferences with H-index 24. He has about three years of experience in industrial research, while working with IBM Research Australia and India. His research interests include distributed computing, cloud computing, HPC, IoT, big data analytics, and education analytics.

Albert Y. Zomaya (Fellow, IEEE) is currently the chair professor of high performance computing and networking with the School of Computer Science, University of Sydney and the director of Centre for Distributed and High Performance Computing. From 2010 to 2014, he was an Australian Research Council professorial fellow, from 2002 to 2007 he was the chair professor of internetworking with the CISCO Systems , and from 2006 to 2007, he was the head of School.

Rajiv Ranjan (Senior Member, IEEE) received the PhD degree from the Department of Computer Science and Software Engineering, the University of Melbourne, in 2009. He is currently a full professor of computing science with Newcastle University, U.K. From 2013 to 2015, he was Julius fellow, a senior research scientist, and project leader with the Digital Productivity and Services Flagship of Commonwealth Scientific and Industrial Research Organization (CSIRO C Australian Government's Premier Research Agency). He was a senior research associate (lecturer level B) with the School of Computer Science and Engineering, University of New South Wales.