SmartMonit: Real-time Big Data Monitoring System

Umit Demirbaga*†, Ayman Noor*‡, Zhenyu Wen*, Philip James*, Karan Mitra§, Rajiv Ranjan*
*Newcastle University, †Bartin University, ‡Taibah University, §Luleå University of Technology

***Abstract*—Modern big data processing systems are becoming very complex in terms of large-scale, high-concurrency and multiple talents. Thus, many failures and performance reductions only happen at run-time and are very difficult to capture. Moreover, some issues may only be triggered when some components are executed. To analyze the root cause of these types of issues, we have to capture the dependencies of each component in real-time.**

In this paper, we propose SmartMonit, a real-time big data monitoring system, which collects infrastructure information such as the process status of each task. At the same time, we develop a real-time stream processing framework to analyze the coordination among the tasks and the infrastructures. This coordination information is essential for troubleshooting the reasons for failures and performance reduction, especially the ones propagated from other causes.

***Index Terms*—Hadoop, MapReduce, Monitoring**

I. Introduction

Big data systems, such as Hadoop and Spark typically run in large-scale computer clusters. For example, Fuxi [1] is an extended implementation of Yarn, which is deployed in a cluster with over 5000 nodes and serves hundreds of millions of customers at Alibaba. For these large-scale systems, there are two key issues that cause performance reduction and inefficient resource utilization. First is the task failures caused by diverse sources of software and hardware faults, and the second is applying unsuitable scheduling policies.

The fault detection in big data systems, however, is very hard due to the considerable scale, the distributed environment and the large number of concurrent jobs. State-of-the-art research is not focused on detecting *emergent failures*. *Emergent failures* happen when the errors exceed the propagation boundaries during the interaction among hardware and software components, and can only be identified at run-time [2]. For example, in the *stragglers* or *tailing behavior* in Hadoop systems, the slower execution of a job may cause the late-time failures for many other tasks, which have strict time constraints related to service-level agreements (SLAs). Moreover, the cluster scheduler uses some heuristics that prioritizes the important jobs and fairly allocates the resources among the jobs [3]. However, these methods ignore the information of the job structure (or dependencies) and schedule the jobs to the available resources without considering the job structure at run-time [4].

In order to detect the *emergent failures* or underlying reasons for the performance reduction, we need to have a comprehensive and consistent monitoring plan to collect the information from each individual process job, while storing, maintaining and analyzing very large volumes of the monitoring data [5]. The monitoring tools such as Google Cloud Monitoring[1], SPARKLINT[2] and Datadog [3] aim to collect various information such as CPU, memory, disk, available bandwidth and execution status of each job. However, they are not able to do a real-time multi-resolution analysis that narrows the scope and increases the resolution, thereby pruning the non-important information.

In this paper, we propose a real-time monitoring system that efficiently collects the run-time system information including computing resource utilization and job execution information and then interacts the collected information with the *Execution Graph* modeled as directed acyclic graphs (DAGs). For example, our system is able to capture the job execution stages and the dependencies of each job in real-time; at the same time, the resource utilization of each job and its underlying host are monitored as well. The main contributions are summarized as follows.

- We develop a big data monitoring system, which can efficiently collect the comprehensive monitoring information from large-scale computer clusters in real-time.
- At the same time, we process these streaming data and interact the processed data with the *Execution Graph* of each task while visualizing the interaction in real-time.

To demonstrate the effectiveness of our system, we applied our system to a small Hadoop cluster that is deployed on AWS. The abovementioned monitoring information is collected in real-time and visualized in a user-friendly interface. The demonstration is available as a screen-cast video on YouTube [6].

II. SmartMonit System Overview

This section explains the architecture and implementation details of SmartMonit.

A. System Architecture

Fig. 1 shows the three main components of our SmartMonit including *Information Collection*, *Computation and Storing* and *Visualization*.

Information Collection. *Information Collection* is used to collect the job and task metrics and the resource utilization of the nodes in a large-scale computer cluster in real time via *SmartAgent* and *Agent*. The *SmartAgent* is deployed on the master node and collects the specific information of tasks (mappers and reducers), application (job details) and the cluster information from the *DataNodes* through the *TaskTrackers*

[1]https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/
[2]https://github.com/groupon/sparklint
[3]https://docs.datadoghq.com/

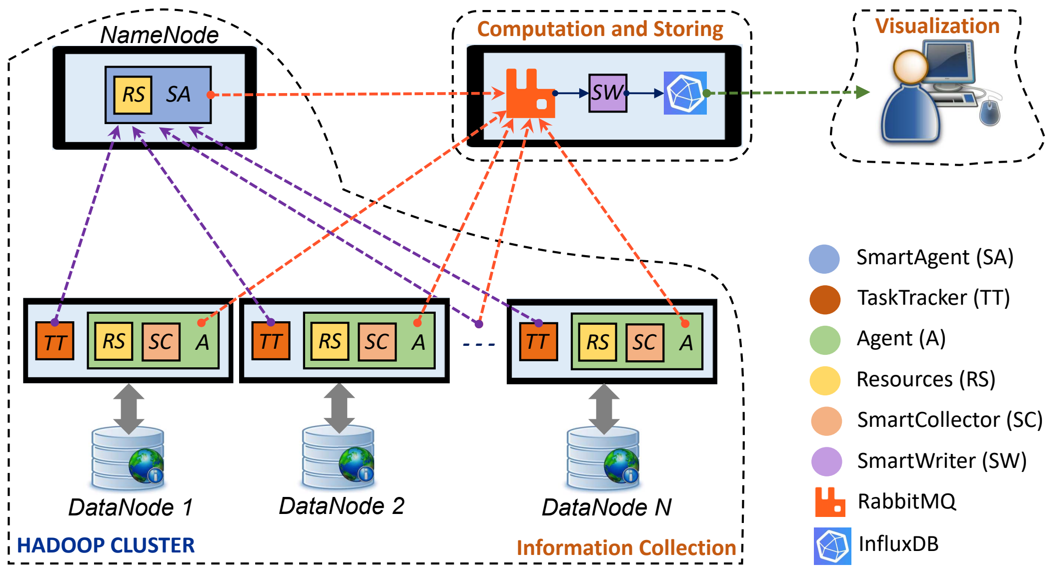


Fig. 1. The framework of monitoring Hadoop cluster in real-time.

using Yarn APIs. Also, the *SIGAR* library is plugged into the *SmartAgent* to monitor the utilization of the resources in the master node, including CPU, memory and network bandwidth. The *SmartCollector* obtains the process information that can be used to build the *Execution Graph* (the details are discussed in §II-B). All monitoring information is streamed to the *Computation and Storing* module. Moreover, in the Slave nodes, the *Agent* collects the process information by using *SmartCollector* and the resource utilization of its host node via the *SIGAR* library. The obtained information is sent directly to *Computation and Storing*.

Computation and Storing. A *RabbitMQ Server* is used to collect the monitoring information sent from the cluster; the *RabbitMQ Server* is an open source message broker system, which provides high throughput, low latency and reliable communication among applications. Then the *SmartWriter* analyzes the collected information pulled from the *RabbitMQ Server* in real-time and writes the processed results into *InfluxDB*, which is an open-source time series database. This database provides high-availability storage and the retrieval of time series data, such as operations monitoring data, sensor data and application metrics.

Visualization. The *Visualization* includes two parts: *query engine* and *user interface*. The *query engine* queries the database in a pre-defined time interval to build the *Execution Graph* (see §II-B). The *Execution Graph* and other collected monitoring information are visualised to enable rapid understanding and diagnostics from human operators.

B. Building Execution Graph

We have developed a real-time stream process module to capture the *Execution Graph* of an application whilst it is running. This module consists of *SmartCollector* and *SmartWriter*. The *SmartCollector* collects the size of each key-value pair generated from each node and sends the collected information to RabbitMQ server. Then, the *SmartWriter* analyzes the streaming data and computes the data transfer size among the mappers and reducers. The following describes the implementation details of this module and Fig 2 illustrates the logic of the algorithm via a WordCount application.

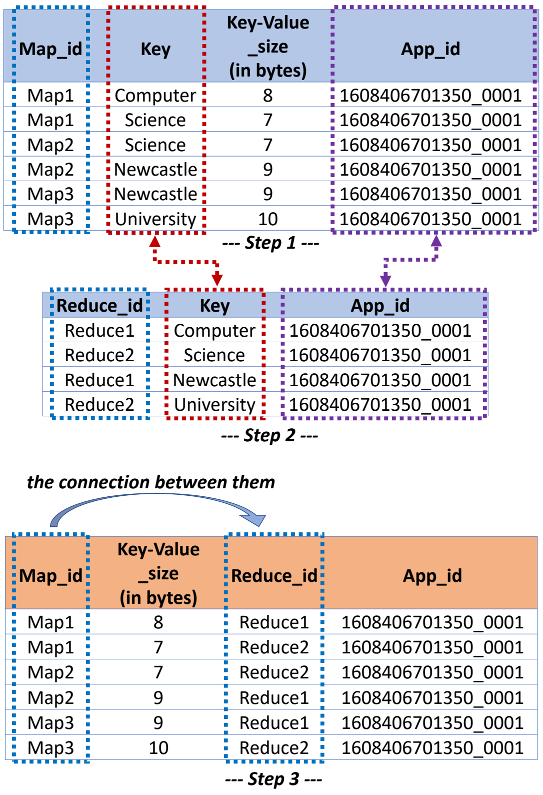
In the map phase, each mapper is assigned more than one key and each key has one value, which is equal to 1. Therefore, we use a *4-tuple* to record the information of each *key-value* pair, i.e., *Map_id, key, key-value size, App_id* as shown in Fig 2, **Step 1** and the recorded tuples are forwarded to RabbitMQ server when they are obtained. In the reduce phase (see **Step 2**), we apply a similar method, but using a *3-tuple* to record the information of each reducer, i.e., *Reduce_id, key, App_id*. Finally, in **Step 3** we use *Key* and *App_id* to match the dependencies among the mappers and the reducers from the same application in *SmartWriter*. For example, The second and third tuple in **Step 1** have the same key ("Science"), and the key ("Science") is shuffled to Reduce2 (see the second tuple in **Step 2**). As a result, we are able to compute the size of the data that is shuffled from mappers to reducers according to the table shown in **Step 3**.

III. Demonstration

This section demonstrates the execution workflow of our SmartMonit by interacting it with a micro-benchmark.

A. Experiment setup

Micro-benchmark. We used Hadoop 3.2.0 and deployed it over 3 AWS virtual machines (VMs). All nodes have 2 CPU cores and 8 GB memory. Moreover, we deployed the *computation and storing* module on a VM with the same instance type outside the Hadoop cluster. The *Agent* and *SmartAgent*

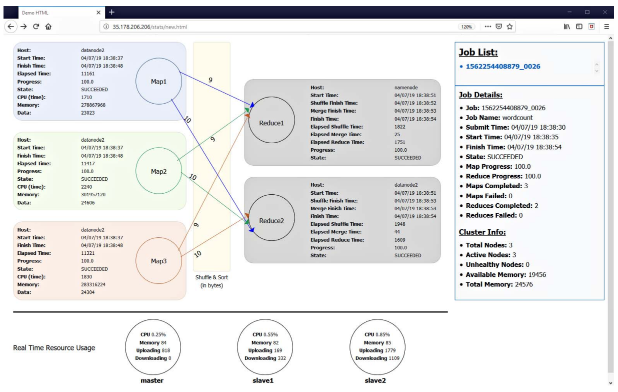

--- Step 1 ---
--- Step 2 ---
the connection between them
--- Step 3 ---

Map_id	Key	Key-Value _size (in bytes)	App_id
Map1	Computer	8	1608406701350_0001
Map1	Science	7	1608406701350_0001
Map2	Science	7	1608406701350_0001
Map2	Newcastle	9	1608406701350_0001
Map3	Newcastle	9	1608406701350_0001
Map3	University	10	1608406701350_0001

Reduce_id	Key	App_id
Reduce1	Computer	1608406701350_0001
Reduce2	Science	1608406701350_0001
Reduce1	Newcastle	1608406701350_0001
Reduce2	University	1608406701350_0001

Map_id	Key-Value _size (in bytes)	Reduce_id	App_id
Map1	8	Reduce1	1608406701350_0001
Map1	7	Reduce2	1608406701350_0001
Map2	7	Reduce2	1608406701350_0001
Map2	9	Reduce1	1608406701350_0001
Map3	9	Reduce1	1608406701350_0001
Map3	10	Reduce2	1608406701350_0001

Fig. 2. The algorithm of SmartWriter.



Host:	datanode2
Start Time:	04/07/19 18:38:37
Finish Time:	04/07/19 18:38:48
Elapsed Time:	11161
Progress:	100.0
State:	SUCCEEDED
CPU (time):	1710
Memory:	278867968
Data:	23023

Host:	datanode2
Start Time:	04/07/19 18:38:37
Finish Time:	04/07/19 18:38:48
Elapsed Time:	11417
Progress:	100.0
State:	SUCCEEDED
CPU (time):	2240
Memory:	301957120
Data:	24606

Host:	datanode2
Start Time:	04/07/19 18:38:37
Finish Time:	04/07/19 18:38:48
Elapsed Time:	11321
Progress:	100.0
State:	SUCCEEDED
CPU (time):	1830
Memory:	283316224
Data:	24304

Host:	namenode
Start Time:	04/07/19 18:38:51
Shuffle Finish Time:	04/07/19 18:38:52
Merge Finish Time:	04/07/19 18:38:53
Finish Time:	04/07/19 18:38:54
Elapsed Shuffle Time:	1822
Elapsed Merge Time:	25
Elapsed Reduce Time:	1751
Progress:	100.0
State:	SUCCEEDED

Host:	datanode2
Start Time:	04/07/19 18:38:51
Shuffle Finish Time:	04/07/19 18:38:53
Merge Finish Time:	04/07/19 18:38:53
Finish Time:	04/07/19 18:38:54
Elapsed Shuffle Time:	1948
Elapsed Merge Time:	44
Elapsed Reduce Time:	1609
Progress:	100.0
State:	SUCCEEDED

Job List:
- 1562254408879_0026

Job Details:
- **Job:** 1562254408879_0026
- **Job Name:** wordcount
- **Submit Time:** 04/07/19 18:38:30
- **Start Time:** 04/07/19 18:38:35
- **Finish Time:** 04/07/19 18:38:54
- **State:** SUCCEEDED
- **Map Progress:** 100.0
- **Reduce Progress:** 100.0
- **Maps Completed:** 3
- **Maps Failed:** 0
- **Reduces Completed:** 2
- **Reduces Failed:** 0

Cluster Info:
- **Total Nodes:** 3
- **Active Nodes:** 3
- **Unhealthy Nodes:** 0
- **Available Memory:** 19456
- **Total Memory:** 24576

	master	slave1	slave2
CPU	0.25%	0.55%	0.85%
Memory	84	82	85
Uploading	818	169	1779
Downloading	0	332	1109

Fig. 3. Execution graph in a real-time monitoring system.

are deployed inside the cluster to collect the monitoring information in real-time. The high-level deployment structure is shown in Fig 1.

B. Real-time monitoring

After the experimental environment was set up, we ran various configurations of the *WordCount* application, in terms of input data size, number of mappers and number of reducers, to evaluate our system.

Fig 3 is a screenshot that shows the real-time execution status and resource utilization of running a configuration of the *WordCount* application. In order to show the full picture of our design, the screenshot was taken when all map jobs and reduce jobs are completed. For more details, please see our screen-cast video on [6].

In the map phase, Figure 3 indicates that all mappers are scheduled to *slave1* and their execution status and the resource usage of the entire cluster are monitored by the *Agents* and *SmartAgent*, displayed in the user interface in real-time (see cycles in the button).

In the shuffle phase, the dependencies between mappers and reducers are obtained by analyzing the collected information through our real-time stream process algorithm discussed on §II-B. Notably, the algorithm also computes the input data size of each reducer (see the numbers above the dependencies) in real-time during the shuffle phase. With this information, we can diagnose the non-salient reasons that cause the performance reduction in the hadoop cluster. For example, if most of the reducers are not running on the nodes containing their input data, the data shuffling will reduce the performance significantly.

The reduce phase is very similar to the map phase, we collect the execution status of reducers and resource usage of their hosts (see Fig 3). The right-hand side summarizes the collected monitoring information of the entire hadoop cluster.

IV. Conclusions

In this paper, we proposed a novel tool that efficiently monitors big data systems. The proposed system collects the run-time system information including computing resource utilization and job execution information. Importantly, it is able to process the collected information to build a dynamic *Execution Graph* for each application while visualizing the graph in real-time.

V. Acknowledgements

This research was partly supported by Engineering and Physical Sciences Research Council, UK under grant EPSRC - EP/R033293/1.

References

[1] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu, "Fuxi: A fault-tolerant resource management and job scheduling system at internet scale," *Proc. VLDB Endow.*, vol. 7, no. 13, pp. 1393–1404, Aug. 2014. [Online]. Available: http://dx.doi.org/10.14778/2733004.2733012

[2] P. Garraghan, R. Yang, Z. Wen, A. Romanovsky, J. Xu, R. Buyya, and R. Ranjan, "Emergent failures: Rethinking cloud reliability at scale," *IEEE Cloud Computing*, vol. 5, no. 5, pp. 12–21, Sep. 2018.

[3] X. Sun, C. Hu, R. Yang, P. Garraghan, T. Wo, J. Xu, J. Zhu, and C. Li, "Rose: Cluster resource scheduling via speculative over-subscription," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, July 2018, pp. 949–960.

[4] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," *arXiv preprint arXiv:1810.01963*, 2018.

[5] A. Noor, K. Mitra, E. Solaiman, A. Souza, D. N. Jha, U. Demirbaga, P. P. Jayaraman, N. Cacho, and R. Ranjan, "Cyber-physical application monitoring across multiple clouds," *Computers & Electrical Engineering*, vol. 77, pp. 314–324, 2019.

[6] Demonstration. [Online]. Available: https://youtu.be/Ok0iJBbC5zA