A Method for Improving the Performance of Spark on Docker Cluster Based on Machine Learning

Chunqi Tian [1, 2, a], Jing Li [1,2], Wei Wang [1,2,3], Liqing Zhang [1,2]

[1]Department of Computer Science and Engineering, Tongji University, Shanghai 200092, China;

[2]The Key Laboratory of Embedded System and Service Computing of Ministry of Education, Tongji University, Shanghai 200092, China;

[3] Hubei Engineering Research Center for Education Information, Wuhan Hubei 430062, China)

[a] tianchunqi@tongji.edu.cn

Abstract

At present, Spark-based applications are very extensive. Reasonable configuration will make Spark jobs have higher execution efficiency. A large number of scholars have conducted in-depth research on the parameter tuning of Spark on virtual machine clusters. In recent years, as an emerging cloud computing infrastructure, containers are more and more widely used in service clusters. Therefore, it is also important to study the parameter tuning of Spark on container clusters. This paper studies the parameter configuration problem of Spark on Docker container cluster, and proposes a new parameter tuning method (ContainerOpt), which uses machine learning method to learn and predict the performance of the job under different parameter combinations, and introduces node automatic scaling mechanism that enable higher-input jobs to achieve better performance. In order to achieve a better balance between job execution time and resource occupation, a performance representation model based on time and resource is proposed to replace the traditional performance representation model based on a single execution time. The experimental results show that compared with the default configuration, the parameter tuning method can improve the execution efficiency by 50%, which proves that it has certain rationality.

Keywords

Spark, Docker, machine learning, parameter tuning.

1. Introduction

In 2008, Google published a paper [1] to propose the MapReduce programming model. In recent years, Hadoop, an open source big data processing framework based on MapReduce, has been widely used in the industry [2]. However, as the data volume of major enterprises grows rapidly, more and more people are dissatisfied with MapReduce's low processing efficiency in iterative and interactive applications. MapReduce model is divided into two processing stages, Map and Reduce, and the intermediate data between them is stored on the disk. A large number of disk IO operations are introduced for this purpose, so the design idea is not suitable for processing iterative and interactive application. To overcome this type of problem, Spark [3] uses a different approach, which caches all intermediate data in memory instead of disk in the form of an elastic dataset RDD. Each RDD remembers how it is built from other datasets (via map, join, Group by, etc.) and refactors itself when needed. In multi-step analysis, Spark's performance is even 100 times higher than MapReduce. Despite this, Spark's performance still has a lot of room for improvement. Without proper parameter tuning, jobs' execution speed will still be slow, which does not reflect the advantages of Spark as a fast big data computing engine.

The parameter configuration methods can be divided into three categories [3-7]: follow the official best practice adjustment guide [8], offline parameters configuration method [6, 9-11] and online parameters configuration method [12, 13]. Among them, the online parameters configuration method searches for a

more optimal configuration by dynamically assigning different configurations and tentatively running use small number of input tasks. The disadvantage of this method is that it requires more time for exploratory multiple operations, which will have a considerable impact on the total execution time of the job, and the results of a small number of inputs often cannot accurately predict the actual input performance.

At present, most of Spark is deployed on virtual machine (hereinafter referred to as VM), and academic research is also based on VM clusters. However, another virtualization method, Docker, has also been widely researched and applied, and has a great impact on various service modes of cloud computing. Reference [14] shows that Spark deployed in a lightweight virtualization framework Docker cluster can achieve better performance in the same configuration than Spark in a VM cluster. However, how to perform Spark on a Docker cluster Research on parameter configuration optimization is still blank.

This paper studies the parameter configuration tuning of Spark in Docker cluster. Based on the characteristics of fast startup and low resource consumption of Docker, a parameter tuning method based on machine learning is proposed.

First, we collect the performance of the job under different configurations and different node numbers, then use the GBDT (Gradient Boosting Decision Tree) algorithm to train the data to obtain the performance prediction model. For small-scale input jobs, they are directly submitted to cluster with the parameters recommended by the performance prediction model a. For large-scale input jobs, they are submitted with more nodes and appropriate parameters.

2. Related work

In Reference [15], the influence of Docker running parameters on Spark job is studied. By comparing the execution time under resource interference and resource completion, it concludeed that Docker configuration parameters can greatly affect Spark performance, especially for different types job, such as WordCount and Sort.

At the same time, there are many references aim to improve MapReduce, such improving the performance of the algorithm, or tuning the parameters of MapReduce jobs. Reference [9] proposed AROMA, a two-stage machine learning and optimization framework. Its first phase is offline work, collecting information about past jobs and using K-Medoid cluster jobs into several categories, each with similar CPU, memory, and disk utilization. Then for each category, we use the support vector machine training data to obtain a performance prediction model for predicting the performance of the job under different combinations of different configuration parameters and input sizes; the second stage is online work. First, we use the default configuration and a small part of the input to run the job to get the resource utilization characteristics, and select the category which has the most similar resource utilization characteristics. Then, using the search optimization technology based on the category corresponding performance prediction model, the approximate optimal parameter configuration that minimizes the resource overhead within the runtime time limit is found.

3. Spark parameter tuning method - ContainerOpt

3.1 Overview of ContainerOpt

The ContainerOpt presented in this paper is dedicated to achieving the highest price/performance ratio in terms of resource usage and job execution time, thereby increasing system throughput and performance of Spark jobs. Figure 1 shows the overall flow of the method, which is based on the second-generation Hadoop-YARN implementation. First, analyze the overall execution flow of the job and combine the research of the predecessors to select the parameters that have a greater impact on the performance of the job. Then, we Collect the performance data of the job under different parameter combinations, use the Gradient Boosting Decision Tree (GBDT) to train the data to obtain the performance prediction model and integrate it into Hadoop. When a new job enters the cluster, ContainerOpt search the parameter space and use the configuration recommended by the performance

prediction model to run new jobs. In this process, if the input data of the job is large (GB level), the node automatic scaling mechanism is started. The number of nodes and the configuration given by the model are used as the new configuration of the job to run the job and record the results in the cluster to decide whether to put the result into the data set. Next, the specific implementation of ContainerOpt will be explained.

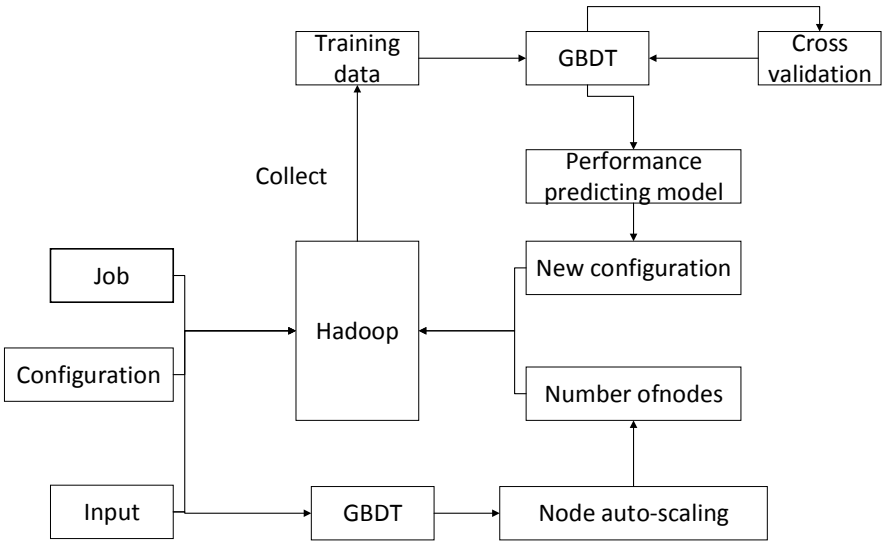

Fig.1 The Overview of ContainerOpt

3.2 Performance representation Model

At present, most studies use job execution time as the only evaluation criterion for Spark performance, but experiments in section 4 of this paper show that in many cases a very small part of the improvement in execution time comes at the cost of a large increase in resources. This is actually unreasonable, because usually there is not only one job in the cluster. If the new job submitted consumes a lot of resources, it will drag down the execution of other jobs and reduce the overall throughput of the system.

On the one hand, in the same cluster, the execution time of jobs with the same allocated resources and input scales is roughly the same, while in different clusters, the execution time of the same resource allocation and input scale may be different; on the other hand, Predicting accurate job execution time is very difficult and completely unnecessary. Therefore, this paper considers the promotion amount of job execution time, that is, the proportion of the reduction of the execution time compared with the time of the job running under the configuration parameter of the minimum required amount of resources (see formula 2), where the minimum required amount of resources refers to the minimum required amount of resources for the successful operation of the job.

$$\mathrm{TI} = \frac{T_{min} - T_{run}}{T_{min}} * 100\% \qquad formula1$$

When the size of job input is small, increasing the CPU and memory allocation, TI will only have a very small improvement. On the contrary, this method will take up more resources and affect other operations. Therefore, this article uses the following model to represent performance:

$$\mathrm{P} = \alpha * \mathrm{TI} + \beta * \frac{1}{\mathrm{R}} \qquad formula2$$

Where R represents the number of resources configured. As can be seen from the formula, the larger the TI, the smaller the R, the better the performance. α and β are used to adjust the weight of execution time and resource consumption in the performance representation model. Experiments show that the range of β is between 0.1 and 0.5.

3.3 Parameter selection

Spark has more than 180 configuration parameters, but only a small amount is critical to the performance of Spark jobs. Combined with previous research, this paper selects six parameters that have a great impact on job performance. The parameters are shown in Table 1. Table 1 also lists other parameters.

Table 1 Spark parameters

parameters	description	Default	range
Spark.num.instances	Number of Executor	2	2~nodes*4
Spark.driver.cores	Number of Driver's CPU	1	1~4
Spark.driver.memory	Driver's memory	1	1~4
Spark.executor.cores	Number of Executor's CPU	1	2~16
Spark.executor.memory	Executor's memory	1G	2`64G
Spark.default.parallelism	Executor's parallelism		Nodes*executor.cores*3
inputsize	Input size		
nodes	Number of nodes		

3.4 Data collection

This article selected 16 types of load for the big data benchmarking framework HiBench, including typical loads (such as WordCount, Sort), iterative jobs (such as PageRank) and machine learning jobs (such as k-means).

The collection of data includes an offline phase and an online phase. In the off-line phase, a sparse sampling method is used to generate an appropriate number of parameter combinations in the parameter space formed by the combination of all configuration parameters: Firstly, the sampling range should be as uniform as possible in the parameter space to avoid excessive deviation between the sampling data and the real data. The search is then performed at smaller intervals within the range of the better-performing parameters, providing optimal results for the parameter search of the new job. During this process, some jobs may fail to run or run several times longer than the default parameter. It is not appropriate to set the run time increment of these jobs to -100%.Using the above approach, you can generate about 400 parameter combinations, and not all of the parameters for each workload are exactly the same. To rule out other factors and randomness, each combination of parameters for each workload was run three times for its average value, resulting in about 6,400 pieces of data.

In the online phase, for a newly submitted job, the appropriate parameter configuration was firstly searched by using the performance prediction model, and the execution time increase predicted by the model at this time was recorded. Then use this parameter configuration to run the job and record the running time. If the time difference is not big, the data of this running will be added to the training data of the model, in order to get more accurate results of the next operation. If large, we consider whether the cluster jitter or the number of jobs cause the difference, and do not add data temporarily, but wait for the result of the next job running. If the deviation persists, it is necessary to consider whether the cluster has been replaced.

3.5 Model training and applying

We use the Gradient Boosting Decision Tree (GBDT) to train the performance data. GBDT, also called MART (Multiple Additive Regression Tree), is an iterative Decision Tree algorithm, which is composed of Multiple Decision trees, and the conclusion of all trees added together to do the final answer.

In order to prevent overfitting, cross validation is used to improve the capability of performance prediction model. After collecting the original performance data, the data is divided into 80% training set and 20% test set. The training set is used to train the model, while the test set is used to test the

accuracy of the model. In order to get more accurate results, we randomly divided the data and carried out three such tests, and the experimental results proved that GBDT has more than 90% accuracy.

Once we have the final model, we can use it to predict the performance of Spark jobs given a combination of parameters. In this paper, random recursive search is used to search parameter space. When the optimal parameter combination given by the model is obtained, if the number of nodes changes, a script is used to quickly start a new Docker to change the Hadoop slaves file and distribute it to each slave node, and then another script is used to set parameters and submit the job for operation.

4. Experiment

The server configuration used in this experiment is Intel i7, 32 core, 128G memory. Initially, four containers were started, one as the master node and three as the slaves node. The version number of Docker is 17.0.6, the version of Spark is 1.6.2, and the version of Hadoop is 2.6.5.

4.1 The influence of the number of nodes on the performance of the job

First, we evaluate the execution of jobs with different number of nodes under the condition that other parameters are exactly the same.

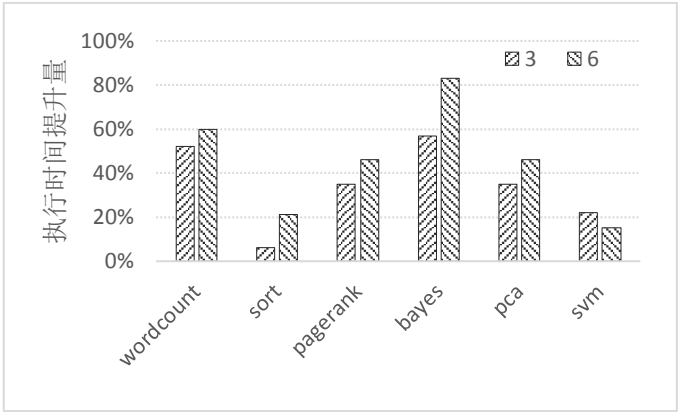

Fig. 2 The impact of the number of nodes on the job performance

The above figure shows the improvement of execution time obtained by 6 jobs when the parameters are exactly the same(the number of executors is 6, executors' CPU is 4,executors' memory is 6GB, paralleliam of job is 24 and input size is 3GB) but the number of nodes is 3 and 6 respectively. It can be seen that, except for SVM, the jobs executed on the six nodes all achieved a significantly larger increase in execution time than the jobs executed on the three nodes. This is because although the number of executors is the same, the resources of each executor are the same, that is, the resources available for the job are the same, but the 6-nodes cluster has a larger HDFS read/write throughput. Therefore, when the input data amount of the job is large and the HDFS is frequently read and written, increase the number of nodes dynamically can achieve better performance. This experiment proves that the node scaling mechanism effectively increases the performance.

4.2 Optimal performance of job under different input

In order to evaluate the best results that ContainerOpt can achieve, the experiment selected three typical types of job types (WordCount, PageRank, LDA) to compare the optimal job execution time improvement achieved with different parameter combinations under different input sizes. The results are shown in figure 3.

The six input sizes for the job are 30KB, 300MB, 3GB, 16GB and 32GB. It can be seen from the experimental results that the execution time of the job running with the ContainerOpt recommended parameter is significantly reduced compared to the default configuration regardless of the input scale (the higher the execution time promotion value, the more the job execution time decreases under the recommendation parameter). And the larger the size of the input data, the more obvious the role of parameter adjustment. This is because the larger the input data size, the higher the resource

requirements. ContainerOpt adjusts the allocated resources by adjusting the parameters, so the job execution speed is faster, which proves that ContainerOpt is indeed effective.

The performance here refers to the performance representation model proposed in this paper. Figure 6 shows how the execution time of Terasort changes as the value of executor.cores change when the value of executor.memory is 4 and input size is 3GB; Figure 7 shows how the execution time of Terasort changes as the value of executor.memory change when the value executor.cores of is 4 and input size is 3GB.

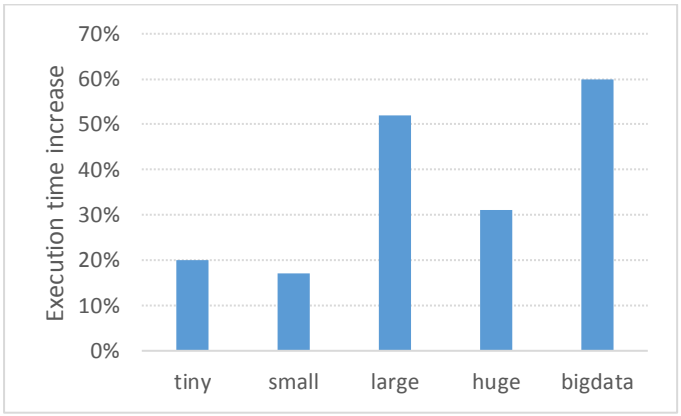

Fig.3 Optimal performance of WordCount with different inputs

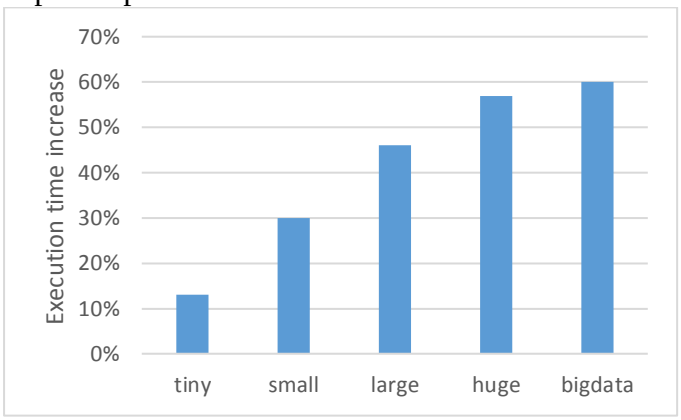

Fig.4 Optimal performance of PageRank with different inputs

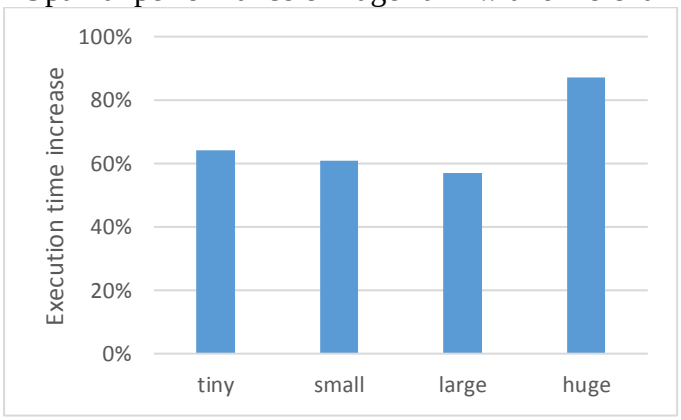

Fig.5 Optimal performance of LDA with different inputs

Fig.6 The amount of execution time increase of TeraSort varies with the number of CPUs.

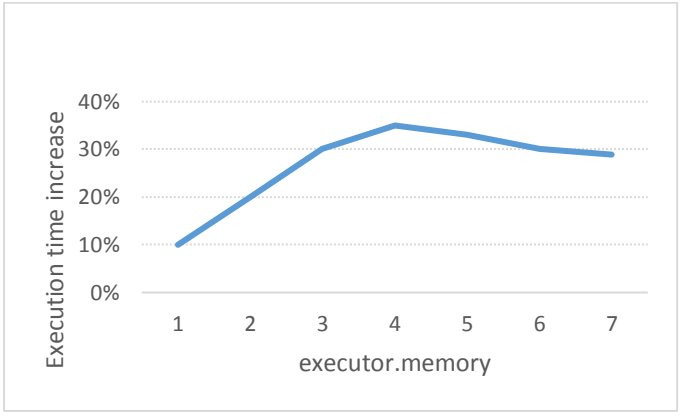

Fig.7 The amount of execution time increase of TeraSort varies with the number of memory

As can be seen from the above figure, the increase of resources is not always positively correlated with the improvement of the execution time of the job, but there is a critical point. When the resources given exceed the critical point, the improvement of the execution time of the job decreases instead. These are mainly because: on the one hand, HDFS has poor concurrency, and when executor's cores toward more, thread switching time increases and job execution time increases. On the other hand, increasing executor memory can cause significant GC latency or even blocking, which can lead to increase of job execution time.

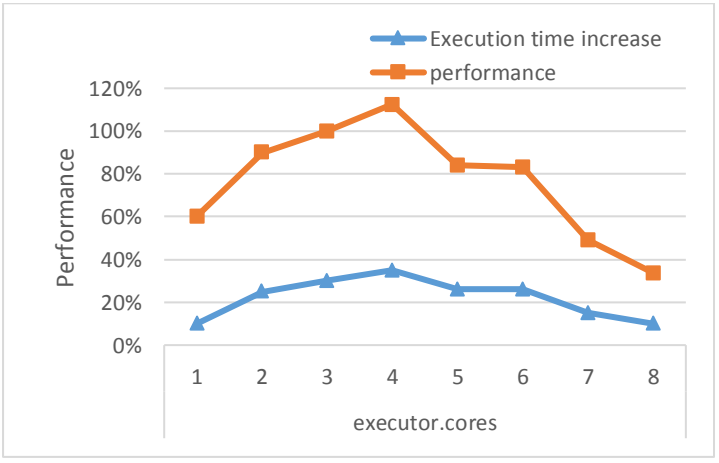

Fig. 8 TeraSort's performance varies with the number of CPUs

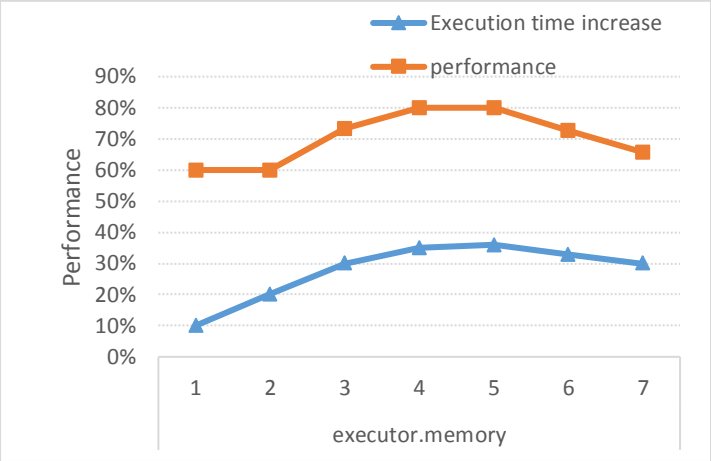

Fig.9 TeraSort's performance varies with the number of memory

It can also be seen from figure 6 and figure 7 that when approaching the optimal point (or the input data size is small), the resource increases by one unit, while the improvement of job's execution time is very limited. This is also the reason why this paper proposes the performance model determined by both execution time and resource.

The following experiment maintained the same conditions above, but the output of the experiment did not take the improvement of the execution time of the job, but the performance of the job, in which the parameter of performance formula 3 was 0.3.

As can be seen from figure 8 and figure 9, the curve trend of execution time improvement and performance is roughly the same, but the size of some points is different.

For example, in the figure 8, when the value of executor's memory is 4GB, the execution time promotion is 35%; when memory is 5GB, the execution time promotion is 36%, which is higher than 35%. However, on the curve of performance, when memory is 4GB, performance is 80%; when memory is 5GB, the performance is also 80%. Although the performance is the same, we will choose the recommendation parameter of 4 instead of 5, so as to achieve the purpose of saving resources.

5. Conclusion

In this paper, Spark on Docker is studied, a performance representation model that is jointly determined by execution time and resource use is proposed, and historical data of a variety of jobs is collected to train for the performance prediction model of jobs with machine learning algorithm. On this basis, ContainerOpt, a parameter adjustment method, is proposed. At the same time, the node auto-scaling mechanism is added to increase the number of nodes when the input data size is large to achieve greater performance improvement, and reduce the number of nodes when the input data size is small to save resources. Experiments show that our method can achieve up to 50% performance improvement.In the future, we will consider more jobs and explore the statistical correlation between jobs, so that when there are new jobs in the cluster, we can judge the suitable configuration according to the previous jobs and increase the scalability of the system.

References

[1] Dean J, Sanjay G. MapReduce: simplified data processing on large clusters , Communications of the ACM, vol.51(2008), 107-113.

[2] Y.B. Chen, B. Liu, Y.T. Shi. Storage and retrieval optimization of large data volume log based on Hadoop architecture ,Information network security, vol.6(2013),p.40-45.(In Chinese)

[3] Apache Spark [EB/OL], http://spark.apache.org/.

[4] Babu S. Towards automatic optimization of MapReduce programs[C]// ACM. Acm Symposium on Cloud Computing, June 10 - 11, 2010, Indianapolis, Indiana, USA. New York: ACM, 2010:137-142.

[5] Herodotou H, Dong F, Babu S. No one (Cluster) size fits all: automatic cluster sizing for data-intensive analytics[C]// ACM. Acm Symposium on Cloud Computing, October 26 - 28, 2011, Cascais, Portugal. New York: ACM, 2011.
[6] Herodotou H, Lim H, Luo G,et al. Starfish: A self-tuning system for big data analytics [J].CIDR, 2011,1(11):161-272.
[7] Ding X, Liu Y, Qian D. JellyFish: Online performance tuning with adaptive configuration and elastic container in Hadoop Yarn[C]// IEEE. IEEE International Conference on Parallel & Distributed Systems, December 14-17, 2015, Melbourne, VIC, Australia. New Jersey: IEEE, 2016:831-836.
[8] Jiang D, Ooi B C, Shi L, et al. The performance of MapReduce: An in-depth study [J]. Proceedings of the VLDB Endowment, 2010, 3(1-2):472-483.
[9] Lama, Palden, and Xiaobo Zhou. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud[C]//ACM. Proceedings of the 9th international conference on Autonomic computing. September 18 - 20, 2012, San Jose, California, USA. New York: ACM, 2012：63-72.
[10] Liao G, Datta K, Willke T L. Gunther: search-based auto-tuning of MapReduce[C]// Euro-Par. Proceedings of the 19th International Conference on Parallel Processing, August 26 - 30, 2013, Aachen, Germany. Berlin: Springer, Heidelberg, 2013:406-419.
[11] Wu D, Gokhale A. A self-tuning system based on application Profiling and Performance Analysis for optimizing Hadoop MapReduce cluster configuration[C]//IEEE. 20th International Conference on High Performance Computing (HiPC), December18-21, 2013, Bangalore, India. New Delhi: IEEE, 2014:89-98.
[12] Li M, Zeng L, Meng S, et al. MRONLINE: MapReduce online performance tuning[C]// ACM. International Symposium on High-performance Parallel & Distributed Computing, June 23 - 27, 2014, Vancouver, BC, Canada. New York: ACM, 2014: 165-176.
[13] Cheng D , Rao J , Guo Y , et al. Improving MapReduce performance in heterogeneous environments with adaptive task tuning[C]//ACM. International Middleware Conference, December 08 - 12, 2014, Bordeaux, France. New York: ACM, 2014: 97-108.
[14] Janki B, Zhengyu Y, Miriam L, et al. Accelerating big data applications using lightweight virtualization framework on enterprise cloud[C]//IEEE. 2017 IEEE High Performance Extreme Computing Conference (HPEC), September 12-14, 2017, Waltham, MA, USA. New York: IEEE, 2017:1-7.
[15] Ye K, Ji Y. Performance tuning and modeling for big data applications in Docker containers[C]//IEEE. International Conference on Networking, August 7-9, 2017, Shenzhen, China, Beijing: IEEE, 2017:1-6.
[16] Xueyuan, Brian, Yuansong. Experimental evaluation of memory configurations of Hadoop in Docker environments[C]//IEEE. 2016 27th Irish Signals and Systems Conference (ISSC), June 21-22, 2016, Londonderry, UK. London: IEEE, 2016: 1-6.
[17] Wang K , Khan M M H , Nguyen N , et al. Modeling interference for Apache Spark jobs[C]//IEEE. IEEE International Conference on Cloud Computing, June 27-July 2, 2016, San Francisco, CA, USA. New York:IEEE,2017:423-431.
[18] Álvaro B. H, María S. Perez , et al. Victor M. Using machine learning to optimize parallelism in big data applications[J]. Future Generation Computer Systems, 2017, 86:1076-1092.
[19] Marco V S, Taylor B, Porter B, et al. Improving Spark application throughput via memory aware task Co-location: A mixture of experts approach[C]//ACM. Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, December 11 - 15, 2017, Las Vegas, Nevada.New York:ACM, 2017:95-108.