Computers and Electrical Engineering 92 (2021) 107092

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

Stargazer: Toward efficient data analytics scheduling via task completion time inference☆

Haizhou Du [a,b,*], Keke Zhang [b], Qiao Xiang [a,c]

[a] Tongji University, China
[b] Shanghai University of Electric Power, China
[c] Yale University, United States of America

ARTICLE INFO

Keywords:
Spark scheduling optimization
Delay scheduling
Computation complexity
Deep learning
Data locality

ABSTRACT

The fundamental challenge of data analytics scheduling is the heterogeneity of both data analytics jobs and resources. Although many scheduling solutions have been developed to improve the efficiency of data analytics frameworks (*e.g.*, Spark), they either (1) focus on the scheduling of a single type of resource, without considering the coordination between different resources; or (2) schedule multiple resources by factoring in limited information about analytics jobs without considering the heterogeneity of resources. This paper presents Stargazer, a novel, efficient system that tackles diversity data analytics jobs on heterogeneous cluster by inferring the completion times of their decomposed tasks. Specifically, Stargazer adopts a deep learning model, which takes into considerations multiple key factors of diversity data analytics jobs and heterogeneous resources, to accurately infer the completion time of different tasks. A prototype of Stargazer is fully implemented in the Spark framework. Extensive experiments show that Stargazer can reduce the average job completion time by 21% and improve average performance by 20%, while incurring little overhead.

1. Introduction

In the recent decade, Big Data analytics has become an indispensable tool in high-energy physics science, engineering, healthcare, finance and ultimately business itself, due to the unprecedented ability to extract new knowledge and automatically find correlations in massive datasets that naturally accumulate in our digital age [1]. In this context, the MapReduce [2] model and its open-source implementation, Spark [3,4], were widely adopted by both industry and academia, thanks to implement an efficient Directed Acyclic Graph(DAG) [5] execution engine and use Resilient Distributed Datasets(RDD) [6] storage mechanism that can efficiently process data streams based on memory, which performs up to 100 times faster than Hadoop [7] based on memory, and 10 times faster than Hadoop based on hard disk. However, Spark still has some issues in producing process needed to be optimized [8].

As the scale and complexity of cluster increase, task scheduling is a necessary prerequisite for performance optimization and resource management on the Spark platform. The process of scheduling the tasks into the cluster resources in a manner that minimizes task completion time and resource utilization is known as task scheduling [9]. Scheduling optimization has a significant influence on system performance of Spark. Although, the data locality mechanism has been successfully achieved on Spark platform, we find that data locality mechanism has remarkable side effects on native scheduling strategy of Spark. For example, when Spark does not find a high-priority task, it will be waiting for a long time until locality level is degraded. Specially, some cross-rack tasks

☆ This paper is for special section VSI-pdcat3. Reviews processed and recommended for publication by Guest Editor Dr. Hui Tian.

* Corresponding author at: Tongji University, China.
E-mail address: duhaizhou@shiep.edu.cn (H. Du).

https://doi.org/10.1016/j.compeleceng.2021.107092
Received 16 March 2020; Received in revised form 24 November 2020; Accepted 9 March 2021
Available online 8 April 2021

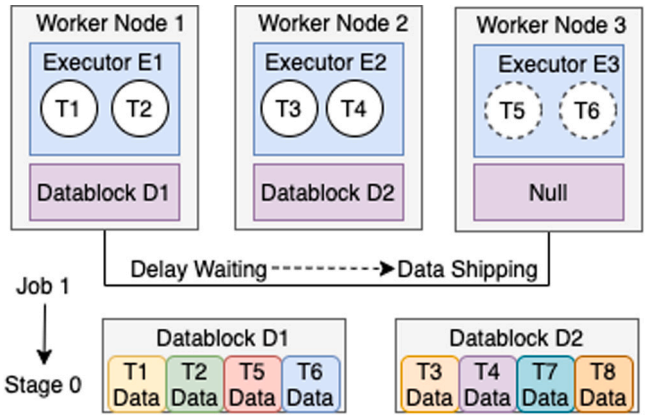


Fig. 1. A motivating example 1: Executor 1 and Executor 2 are already occupied and there are no data stored in Executor 3. Spark starts delay scheduling mechanism, which causes Executor 3 to be idle during the period of time and wastage of resources.

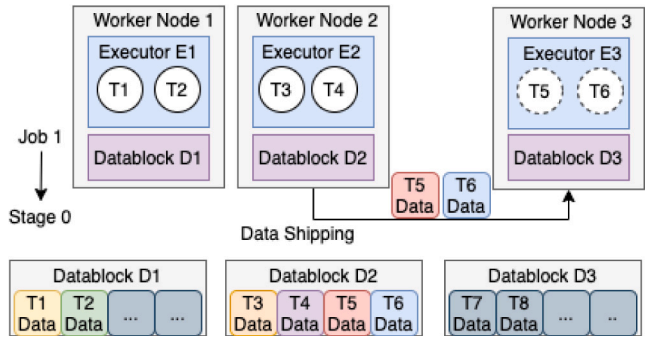


Fig. 2. A motivating example 2: Executor E1 and Executor E2 are already allocated and data of the T5 and T6 are stored in worker node 2. After over waiting latency, the resource of worker node 2 is still occupied, Spark will perform data transfer and pull data remotely to worker node 3. However, if you wait a few more seconds, maybe T5 and T6 will be executed on E2 instead of wasting the remote transfer time.

will generate much network transfer in geographically distributed clusters of real world, which also have a great effect on overall cluster performance. Moreover, efficiently scheduling data processing jobs on distributed Spark clusters requires complex algorithms. Unfortunately, current Spark framework still uses simple heuristics method. It obviously cannot leverage them.

Recent studies on task scheduling algorithms have made substantial progress, in particular on reducing task completion time and allocating the resources properly [10]. Such as [11] combined with machine learning techniques, and considering the heterogeneity of the nodes and their configuration within the grid, and the scheduling of tasks in the dependent DAG, formed the scheduling strategy. [12] considered fair allocation of multiple resource types, which improved system performance from the perspective of resource balance, and eventually gained the good result. [13] coordinates computation-bound and network-bound tasks in a large cluster, and utilized them in a more balanced fashion. However, they all ignore side effects of data locality mechanism on native scheduling strategy of Spark.

Therefore, in this paper we show that deep learning techniques can help to predict task completion time in advance by automatically learning a highly efficient training set to tackle the aforementioned scheduling problem. We design and implement a task scheduling of TCT inference to reduce the task completion time and improves the cluster performance in some degree. At the meanwhile, Stargazer can leverage the network transfer time and delay scheduling. Accordingly, we make the following major contributions.

- To the best of our knowledge, this is the first time making a comprehensive and meticulous analysis on impacting factor of task completion time(TCT). And we strip the four key factors for each task including *node performance, task complexity, data volume and network transfer delay*, which affect the task completion time on a great degree. To the best of our knowledge, this is the first time using deep learning models to optimize task scheduling based on TCT inference on Spark framework.
- We investigate and implement task classification and comparison of the tasks locality based on the TCT inference in the period of task scheduling on Spark platform. We reduce the network transfer time and efficient avoid the waiting time of delay scheduling before task execution. We achieves good efficient as much as possible during computing time.

The rest of this paper is organized as follows. In Section 2, we discuss the motivation of the overall system. In Section 3, we introduce the design of Stargazer. Section 4 demonstrate the evaluation of our system and experiment results. Finally, we review related work in Section 5 and we conclude the paper in Section 6.

2. Motivation

Native Spark performs up to 100 times faster than Hadoop based on memory [14]. Spark relies on data locality, aka data placement or proximity to data source, that makes Spark jobs sensitive to where the data is located.

In order to optimize processing tasks, Spark tries to place the execution code as close as possible to the processed data. This is called data locality. In general manner, Spark will firstly try to move serialized code to the data because the code is usually smaller in size and moving it is much cheaper than transferring the data over the network. However, it is not always possible and sometimes the data must be moved to the executor — for instance when given node is responsible only for storage. The data locality is controlled by the configuration entries beginning with spark.locality.wait. Its value is defined in time units (e.g. s for seconds) and indicates how long Spark can wait to acquire the data on given locality level before giving up. Sometimes the data is not available immediately and the processing task must wait before getting it. However, when the time defined in spark.locality.wait expires, Spark will try less local level, that is: local -> node -> rack -> any.

The problem of imbalance of network resources and computing resources is that network resource is bottleneck in the geo-distributed data center and computing resources is enough. That is one of imbalance phenomenon in spark cluster.

However, new problems have arisen due to the introducing of data locality and delayed scheduling, which is mainly divided into the following two points:

- First, the native data locality of Spark platform has significant side effects. Before allocating resources for a task set, the task set manager will firstly calculate tasks of the level of data locality and Spark prioritizes high-priority tasks with a high level of data locality. The local task is completed very quickly (lower than Spark's delay waiting time parameter Spark.spark.locality.wait). Due to delay scheduling mechanism of Spark, tasks at the next level of data locality are never started. To take an extreme example as shown in Fig. 1, Running a job, and if most data is stored on one node, then tasks are executed on this node on account of data locality, which causes that only the machine where the node is located computation for a long term and the others in the cluster will be always in a waiting state (waiting for local level degradation) and not performing the task, resulting in a huge waste of resources. Moreover, it is incorrect that we select a fixed delay waiting time parameter for all nodes in a heterogeneous cluster.
- Second, the native Spark platform generates an imbalance of the network resource and computing resources on a heterogeneous cluster or geo-distributed clusters. For example, as shown in Fig. 2, the data locality of the current task executing on one node is non-local type, the data block for the current task is not on the node. The data block will be shipped to the node on which a task is executed through network transfer service, thus the network transfer time increases the task execution time. At the same time, it reduces the overall performance of the cluster. An extreme case is when a node completed the current local task and subsequent tasks are all the non-local type. Due to the existing delay mechanism, the delay scheduling will not only result in a large amount of waiting time for task execution but also increase the time of network transmission. It will be important to research whether the next task type can be predicted and transmitted simultaneously in advance when executing the local task on this node.

Thus, our core idea in this paper is to propose an efficient task scheduling of Spark platform that considers task completion time inference. There are still three main challenges lie at the foundation of the scheduling design on Spark platform:

- **Prediction Precision Challenge.** Due to the dynamics and heterogeneity of the cluster, the task completion time is impacted by many factors. For example, the performance of each node will decrease with the job execution, and the network available bandwidth will also change with network traffic. These will eventually affect the task completion time and further affect the accuracy of the prediction time.
- **Estimation Accuracy Challenge.** One of key challenges of TCT-inference scheduling is how to determine whether the next task will execute the delay scheduling. Before we schedule a task in advance based on the predicted completion time of the current task, resources cannot be allocated to the next task at the same time. How to accurately estimate the relationship of locality level between the next task and the current task is very difficult, which is greater than or equal to the following sequence of the task set.
- **Resource Overhead Challenge.** To predict task completion time in advance, we need to monitor the data size across a path and transfer prediction time parameter, which can bring resource overhead. Although we can start other new threads to support our approach, how to reduce overhead is still one of the key challenges.

To tackle the aforementioned challenges, we propose Stargazer, the first task completion inference framework, driven by the design and implementation of a heterogeneous Spark cluster. The basic idea of Stargazer is to systematically analyze and predict task completion time by deep learning technology. In the next section, we will introduce the design of Stargazer.

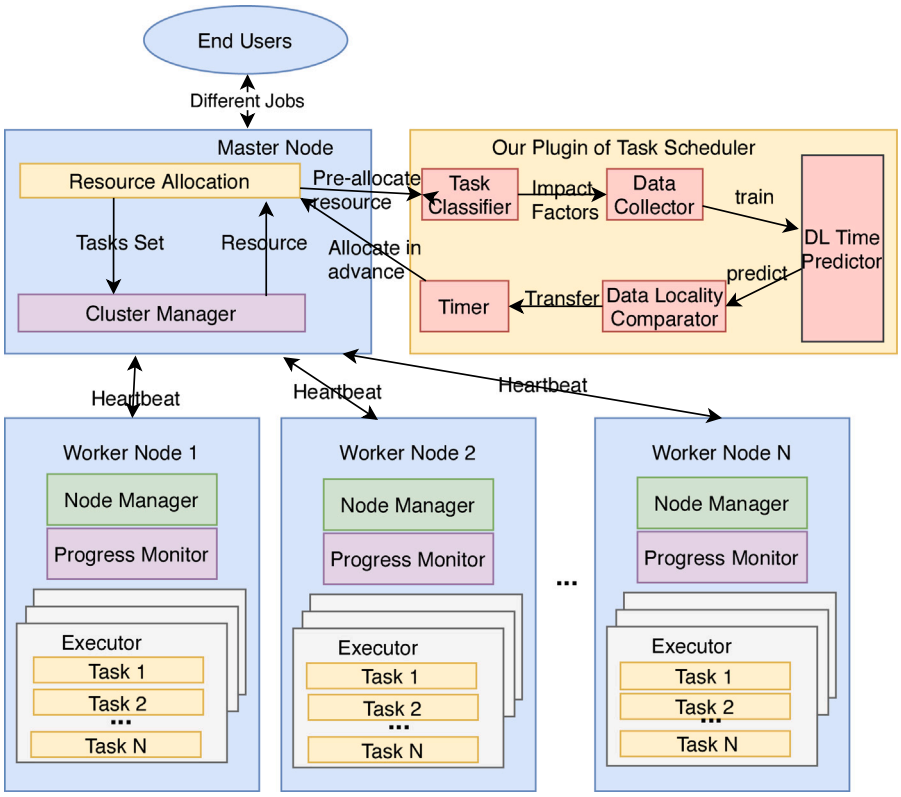


Fig. 3. The overview of Stargazer architecture.

3. Stargazer design

From the above ideas, we have designed the Stargazer, which infers the completion times of decomposed tasks of heterogeneous data analytics jobs and uses such inferred results to schedule the execution of these tasks in the data center. The design and implementation of Stargazer consist of multiple components. In this paper, we focus on two key components: the time prediction component and the task scheduling component.

3.1. Design of system overview

Aiming at the difficulties and challenges presented in Section 2, we propose Stargazer a scheduling optimization approach based on task completion time awareness of the Spark platform. The Stargazer architecture is as shown in Fig. 3. Stargazer mainly consists of time inference component and task scheduling component. The time inference component is based on the long short term memory(LSTM) method for task completion time inference. The task scheduling component implements the function of task classification through task locality mechanism. Moreover, it realizes the comparison of the locality of multiple tasks based on the time inference component.

3.2. Time inference component

Stargazer's design is based on the inference of task completion time. We adopt a mainly deep learning approach to implement the time inference component of the proposed system. The time inference component is divided into four parts: data collection module, training module, data transmission module, and prediction module. The overview of the time inference component is as shown in Fig. 4. The core of the time inference component uses a deep learning model.

We eventually choose long short term memory(LSTM) model as a prediction approach. The LSTM realizes the protection and control of information through the input gate, the forgetting gate, and the output gate. These gates are connected to a multiplication unit to control the state of the cell unit. Eventually, we choose LSTM model to predict task completion time, LSTM adapted to the time prediction problem, and experiment results show good feedback as well.

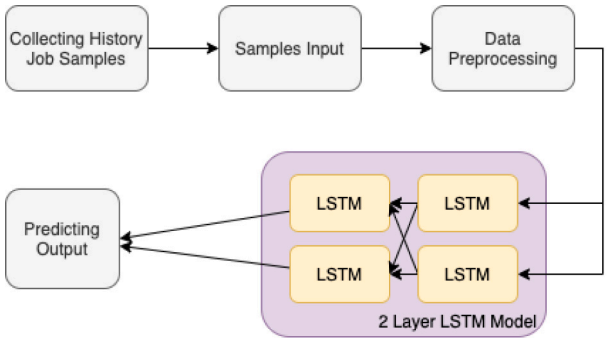


Fig. 4. The overview of time prediction component.

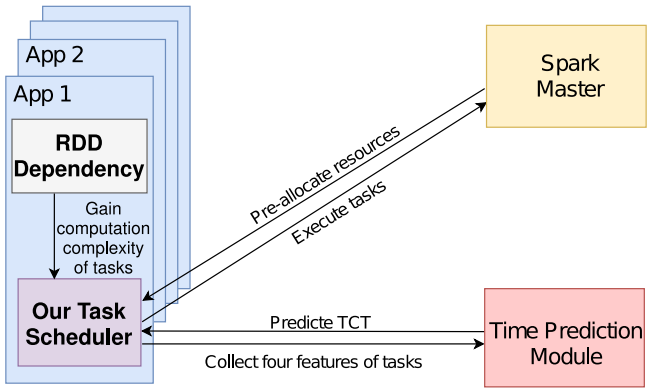


Fig. 5. The overview of time prediction component.

Data collection module

We analyze that task completion time is affected by four factors including task data volume, complexity of task, node performance and network transfer delay [15].

- **Task Data Volume.** when tasks are processed based on the same function logic, the data volume of different task executed on different nodes is different, which caused different task processing time. In our model, We gain the task data volume by reading the byte start-stop index of HDFS data block.
- **Computational Complexity of Task.** The complexity of the task itself will lead to a bottleneck of a single point, resulting in the localized distribution of tasks with high complexity, which will result in restrictions of overall the efficiency of the cluster by single point blocking, making smooth completion of hydration operations unfeasible [8]. We defined transformation times each RDD as measure of complexity of task and implement it through reversing traceability of RDD dependencies as shown in Fig. 3.
- **Node Performance.** Computing nodes have different computing capabilities based on heterogeneity clusters. That is to say, the computing nodes' capabilities is limited by the cluster platform. For example, nodes of high computing performance present an advantage of data processing. We leverage the task average completion time equivalent to the performance of each node.
- **Network Transmission Delay.** When a task is assigned to execute on the node, however, data block required for the task to execute is not stored on the node. Under the circumstances, Spark will activate the network transport service. The network transport service will transfer the required data block from another node to the node. Besides, The network bandwidth and the degree of network congestion between different nodes are different. Therefore, we define the network transfer time as the ratio of the total amount of data on the transfer path to the network bandwidth.

Training module

The purpose of the training module is to learn from the diversity job samples in history, to train a more efficient inference module. We use the two-layer LSTM framework to train the module on the Spark platform. And this training process will be time-consuming and ongoing. We will describe the parameters setting in Section 4. Furthermore, Training different types of jobs LSTM model is based on the same Spark configuration.

Data transfer module

The purpose of Data transfer module is to handle the cost of data transmission in Stargazer. The cost of data transmission is also a key issue. We train the historical data offline, which greatly reduced the time cost. We employ the Scala interface to implement the data collected from Spark cluster and complete the data transfer from the Spark platform to the LSTM model.

Prediction module

The prediction module is mainly used to infer the task completion time of the current task to be executed. The prediction module receives the real-time task features from Spark platform. Then we predict the task completion time and transfer task completion time by the LSTM method. We use two LSTM layers to infer TCT for each task.

3.3. Task scheduling component

We design our task scheduling component without breaking native task localization of Spark. The overview of the ask scheduling component is as shown in Fig. 5. We reduced overall job completion time and improved cluster efficiency by balancing delay scheduling time and network transfer time.

As is shown in Fig. 5, our task scheduling component is divided into four parts including a data collection module, task classifier, task comparator, and task timer.

Data collection module

In this module, We analyze that the task completion time is affected by four factors. The four factors is including task data volume, complexity of task, node performance and network transfer delay. At the same time, these data will transfer into the time prediction module as input data.

Task classifier module

Spark platform defines five localization levels including PROCESS_LOCAL(the data is in the same JVM as the current executor), NODE_LOCAL(the data is on the same server as the current executor), NO_PREF(the data has no locality preference which means data is not on the same node as the current executor and it is also not on other nodes that have executor), RACK_LOCAL(the data is stored on the same rack as the current executor) and ANY(the data is neither stored on the same node nor on the same rack as the current executor). We redefine PROCESS_LOCAL tasks and NODE_LOCAL tasks as the local tasks, and other locality levels as the non-local task [16]. We implement the task classifier and divide different tasks into two categories as local tasks and non-local tasks on the Spark platform. For non-local tasks, we predict job completion time and execution task scheduling based on TCT inference.

Task comparator module

As the above mentioned, How to judge whether the next task will perform delay scheduling or not is a key challenge for Stargazer. We assign tasks in advance based on the current task completion time. Due to the next pending task has not been scheduled to the current node, we cannot gain the data locality level for the next task. Therefore, we cannot judge whether the data locality level of the next task is greater than or equal to the current task. This is the main reason that we added the prediction of the time of the local task into a time prediction component. The purpose of TCT inference of the local task is to compare the completion time of the performed task on each node. We recorded the locality level of the current task. If the locality level of the current task is local, we judge the locality level of the first completed task in the tasks being performed. If the locality level of the first completed task is local, the probability that the locality level of the next scheduled tasks is local is 50% on that node. Therefore, we will assume the locality level of the next scheduled tasks is local under this condition. If a non-local task is executing on the node, the next task on this node will still schedule a non-local task. Hence, we schedule the next non-local task within a delay scheduling time and network transmission time in advance. If the non-local task is completed firstly, the current task set does not need to wait for degradation, and the non-local task is directly scheduled. At the same time, we just save the network transfer time. We can obtain degradation times and waiting times by reading the waiting time from the configuration file on Spark.

Task timer module

Task timer is used to receive the output data of the time inference component It based on our rules for counting down the task ahead of time. We implement an array of *ArrayBuffer* type that records the task prediction completion time from the time inference component. Such as the task locality level, task ID, executor ID and task status. Once the task time is reduced to 0, the execution of the task will be triggered.

In this section, we describe the overview design of the system and present the overview of two main components in detail. We will introduce the implementation of Stargazer in the following section.

4. Evaluation and analysis

We conduct experiments from the simulation in theory and real cluster condition to evaluate the algorithm. In this section, we describe our experimental settings, parameter settings and the analysis of experimental results. We evaluate Stargazer through a series of experiments on a heterogeneous cluster on Spark platform. We begin with four different types of workloads that evaluate how Stargazer behaves under different settings and go on to analyze the cluster overhead by Stargazer.

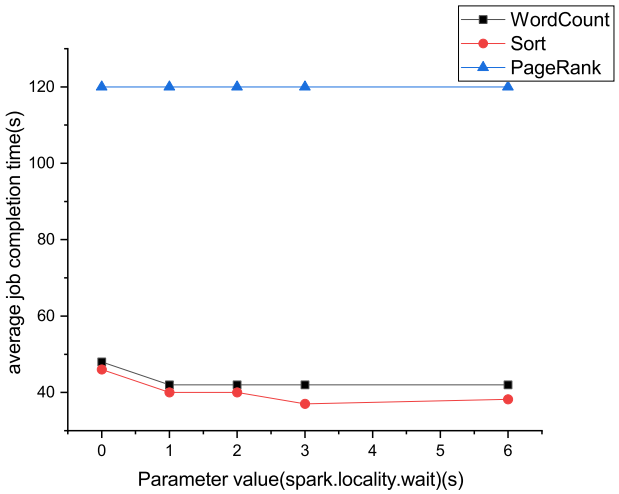


Fig. 6. The average job completion time on WordCount, Sort, PageRank based on various Spark's delay scheduling parameter values on Stargazer.

4.1. Cluster configuration

In our experiments, we have used a total of 20 heterogenous nodes as Spark workers in a 2 racks cluster. The master of our cluster runs on an instance with 40 CPU cores and 80 GB of memory. The master node in our cluster has been used as the Spark master node and the Hadoop Distributed File System(HDFS) Name Node. Ten worker nodes in our cluster is equipped with 4 cores, 16GB of memory. The left worker nodes in our cluster is equipped with 2 cores, 4GB of memory. In each worker node, we run Ubuntu Server 14.04 LTS, with Java 1.8 and Scala 2.12.8 installed. Hadoop 2.7.7 is installed to provide HDFS support for Spark. The bandwidth limit for uplink is 1Gbps whereas downlink bandwidth is as large as 10Gbps. We implement the task assignment algorithm with Spark 1.4.0.

4.2. Competing methods

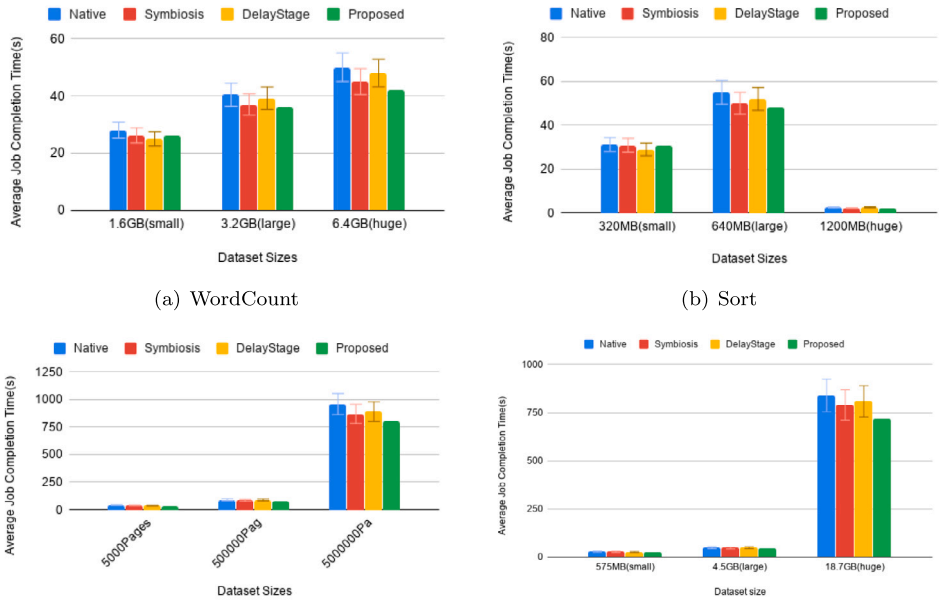
Because of the RDD and localization technology of the Spark platform, there are fewer existing works to predict Spark task completion time than Hadoop platform. They adopt different underlying execution mechanisms. or fairness, we compared Stargazer with three methods, including native Spark scheduling mechanism, Symbiosis [17] and DelayStage [18]. Symbiosis is an online scheduler that predicted resource imbalance and correct imbalance between computation-bound and network-bound tasks in the same executor process. DelayStage is a stage delay scheduling strategy to interleave the cluster resources across the parallel stages, it can increase the cluster resource utilization and speed up the job performance. These methods are chosen as baseline because they are the state-of-the-art in the relevant areas. All methods are implemented on Spark2.2.0 platform and Stargazer are implemented using tensorflow2.04 [19], Symbiosis is taken from our own restore implementation and DelayStage is taken from the authors' github(https://github.com/icloud-ecnu/delaystage).

4.3. Parameter tuning settings

The LSTM model related parameters settings are described as follow. We use the activation function Relu of Python's Keras library and linear fully-connected artificial neural network with default activation function that receives the LSTM output. In order to prevent the occurrence of overfitting, we set the value of dropout probability to 0.2 on the non-recurrent connections of network nodes. The mean square error is used to determine the error calculation method and the RMSprop [20] algorithm is used to determine the iterative update mode of the weight parameter.

We set the learning rate to 0.01 at the begin of training, and then observe the trend of the training cost. If the cost is decreasing, we can gradually increase the learning rate after each epoch. And vice versa. We run four types of popular analytics workloads with different dataset to evaluate the performance of Stargazer. Therefore, we manually adjust the different parameter settings for different workload types in our all experiments. Meanwhile, the appropriate value of the learning rate is stable after several trials. For different workload types, we collected about 8000 pieces of data separately, ten times cross-validation method, and the batch size of our model was 100. The output result of the last neural network layer is the task completion time.

Finally, we consider that the Spark's delay scheduling parameter(spark.locality.wait) and verify that the parameter have little effect on our scheduler system by experiment. The experiment result is shown in Fig. 6. We adopt variously parameter values and gain average job completion time on different workload including WordCount, Sort, PageRank. It help us to select the reasonable parameter value.



(a) WordCount (b) Sort

(c) PageRank (d) K-Means

Fig. 7. We run four types of popular analytics workloads respectively, WordCount, Sort, PageRank, K-Means, and each workload runs with three different data sizes. Comparison of the execution time of the Native, Symbiosis, DelayStage and proposed approach on Spark platform.

4.4. Benchmark

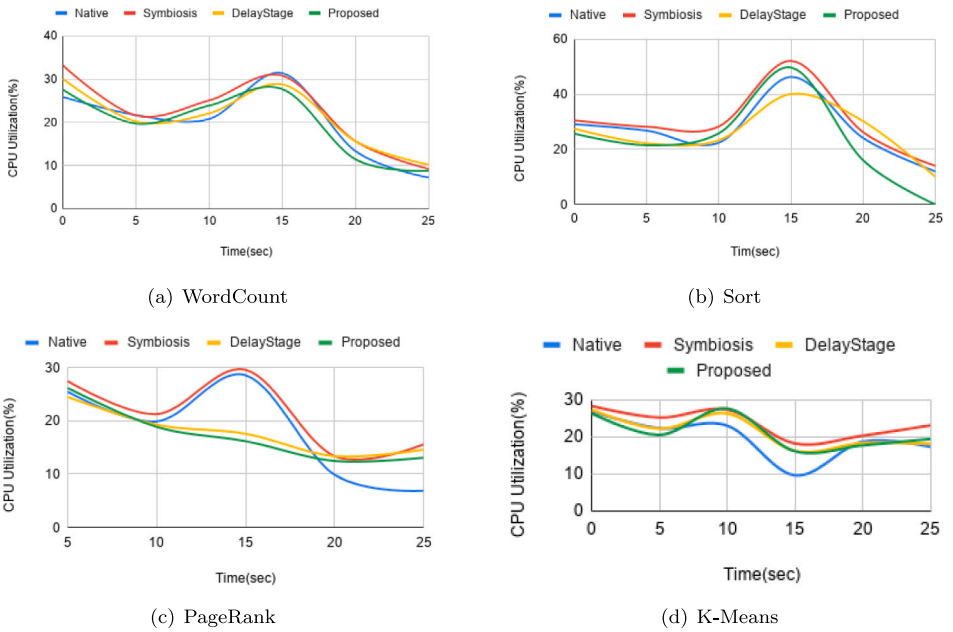
To comprehensively evaluate the performance of Stargazer, we run respectively four types of big analytics workloads, WordCount, Sort, PageRank, K-means, which come from standard benchmark suite of big data analytics platform, HiBench. The WordCount workload counts the occurrence of each word in the input data. The WordCount represents the type of CPU-intensive workload in our experiments. Sort workload sorts its text input data, which is generated using RandomTextWriter. The Sort represents the type of I/O-intensive workload in our experiments. The PageRank is an iterative graph processing algorithm that is designed for Google's web search engine. The data source is generated from Web data whose hyperlinks follow the Zipfian distribution. In the PageRank workload usually incurs a large amount of network traffic to transfer intermediate results among multiple stages of a job. The PageRank workload is thus represents as hybrid workload in our experiments. The K-Means workload is used to test the K-means (a well-known clustering algorithm for knowledge discovery and data mining) clustering in spark.mllib. The input data set is generated by GenKMeansDataset based on Uniform Distribution and Guassian Distribution. There is also an optimized K-means implementation based on DAL (Intel Data Analytics Library), which is available in the DAL module of sparkbench. The K-Means workload represents the type of popular machine learning workloads in our experiments. We generate about 8000 pieces of historical data from Spark with the above 4 types of jobs respectively with Hibench for training model. And we place some datasets in different 2 racks for generating some non-local tasks.

4.5. Effectiveness study

This section examines the efficiency of the our methods, which is a critical factor as it is very difficult to obtain TCT in most big data analytics applications. We aim to answer the following two key questions:

- How efficient are the Stargazer and other methods?
- How much improvement can the Stargazer gain from the TCT inference compared to the native scheduling and others scheduling?

Metrics. We examine the efficiency with respect to data size by generating four synthetic different data sets with different data sizes by HiBench. Similarly, the efficiency test with respect to average job completion time uses a fixed data size and diversity workloads. For example, The large volume of Wordcount datasize is 3.2 GB. The large volume of Pagerank datasize is 500,000 pages. The large volume of Kmeans datasize is 100,000,000 samples. Specifically, We capture the job completion time for each type of workloads. And we observe the completion time variation with the number of compiling servers and network scale. Therefore, we run all types of workloads 20 times, calculate average job completion time and label standard deviation on each bar.



(a) WordCount (b) Sort

(c) PageRank (d) K-Means

Fig. 8. Comparison of the CPU utilization of the native Spark scheduling, Symbiosis, DelayStage scheduling and the proposed approach on four types of popular analytics workloads, WordCount, Sort, PageRank, K-Means on Spark platform.

Findings. In each experiment, the average job completion times achieved with our optimal task assignment algorithm is compared with the default scheduling, Symbiosis and DelayStage scheduling on Spark. The results of our experiments are presented in Fig. 7, showing the average job completion times based on the different size of data set respectively. Obviously, we can see from the figure that Stargazer always performs better than others. The performance improvement is up to 21% from native Spark scheduling as shown in Fig. 7(b). For the Sort benchmark, the 1.2GB dataset is a huge type of dataset on the Hibench suite. We can see that the performance improvement of WordCount and K-Means jobs, which represent computation-intensive type of workloads, obviously higher than the other two I/O-intensive workloads with the increase of the amount of data. As the amount of data increases, the number of non-local tasks generated by computation-intensive workloads increases in Fig. 7(a) and (d). This causes Spark to frequently perform delayed scheduling algorithms and data shipping. However, mainly the time consumption of network-intensive workloads in Fig. 7(b) and (c) is in the shuffle stage. And Stargazer is significantly better than native scheduling. Because the Symbiosis scheduling mainly considers the computation bound and network-bound tasks and DelayStage considers the makespan of the parallel stage. Therefore, under the circumstances where the network is not dominant, Symbiosis performance is not as good as our proposed method. This may be due to the fact that we reduce the network transfer time and avoid the waiting time of delay scheduling during scheduling that task classification and comparison of the locality of multiple tasks. As a result, Stargazer, which uses the LSTM neural network to predict each tasks completion time in the early stage of task scheduling. So Stargazer obtains a shorter average job completion time than all competing methods. Thus, we can conclude that Stargazer not only handle the diversity jobs but enables the ability to leverage the multiple tasks locality to reduce job completion time.

4.6. CPU utilization study

This section investigates the resource overhead of Stargazer with respect to different type jobs with the same data size. We aim to examine the following two key questions:

- How CPU Utilization are the deep learning model for prediction?
- Can the Stargazer still substantially beat the native scheduling and others scheduling with diversity workloads?

Metrics. Although not fully ideal, we still examine the resource overhead with respect to CPU utilization on master node. All fixed datasize of datasets(i.e., 1.2 GB)are generated by the HiBench for the four different frequently-used workloads. We can observe the CPU utilization vibration in a whole job processing from map stage to reduce stage. We had also considered using memory utilization as a metric of resource overhead, but memory fluctuates greatly with many interference factors. Therefore, we just used the CPU utilization as the metric of resource overhead.

Findings. In Fig. 8 we compare the proposed approach with Native, Symbiosis and DelayStage scheduling according to CPU utilization. During the execution of different jobs, the CPU utilization of Stargazer is very high at the beginning stage. We can obviously see that CPU utilization is same between our approach and baselines at most period of a job progress. Specially, at the end of job execution the resource overhead are declined in some CPU-intensive workloads, i.e., Wordcount. Meanwhile, the CPU utilization of our approach is not obvious difference, sometime more higher than other baselines in some I/O-intensive and hybrid workloads. This may be due to the fact that more tasks are completed ahead of scheduling, which reduces the whole job completion time. Moreover, CPU utilization of Stargazer is hardly lower than the native scheduling, and sometimes is higher than the native scheduling. At the same time Symbiosis need more CPU resource to balance different type tasks, Symbiosis significantly costs more CPU resource in all four experiments. DelayState also costs CPU resource to calculate the pipeline. Thus, we can conclude that Stargazer does increase little resource overhead as taking for granted. On the contrary it decreases some degree resource overhead in network-heavy type of jobs.

4.7. Scalability study

We examine the scalability of Stargazer with respect to data size by generating four different type workloads with the 10, 15, 20 nodes of cluster sizes respectively. Similarly, the scaleup test with respect to dimension uses a fixed data size (i.e., 3.2 GB) and varying data volume. We aim to examine the following two key questions:

- Can the Stargazer still substantially beat the native scheduling and others scheduling on scalability with different cluster size?
- How much improvement can the Stargazer gains for different types of workload with different cluster size?

Metrics. We measure scalability on diversity workloads, which include the computation-intensive WordCount workload, the I/O-intensive Sort workload, the network-heavy PageRank workload and the machine learning K-Means workload, by average job completion time with 10, 15, 20 different Spark cluster sizes. Each methods is tested by running in a data set of the same size 20 times.

Findings. The scaleup test results are presented in Fig. 9. These results show that the performance of Stargazer gains in terms of average job completion times vary in different cluster size. Specifically, in the Sort workloads Stargazer gains the average performance about 20% better than the native scheduling, 10% better than Symbiosis and DelayStage scheduling. The effect of Stargazer performance optimization in Fig. 9 is not obvious. This is because although the size of the cluster keeps increasing, however the datasize of workloads does not change significantly. Therefore the cluster system resources are obviously more sufficient to handle these workloads. The cluster with 10 nodes in the figure runs the same WordCount workload as other cluster, which obviously takes a lot of time. This is because the WordCount is the type of CPU-intensive workload, so it takes a lot of time. The four types jobs are all time-consuming on cluster with 20 nodes, which is due to the amount of data running on the cluster is small, and it is not necessary to run so many nodes. Stargazer is significantly better than Symbiosis and DelayStage in all four experiments. The increase in the number of nodes has greatly increased the other cost such as startup and registration on Spark, leading to the most costly job completion time here.

In summary, we can analysis the results from the such above achieved performance improvement as follows. This is because that Stargazer classifies all tasks from the early map stage. In order not to break the task locality mechanism on Spark platform, we implement a efficient task scheduling algorithm for non-local tasks. Stargazer balances the computation resources with network resources and scheduling latency. Essentially, Stargazer reduce delay waiting time and decrease network transfer. Otherwise, due to the Spark delay scheduling mechanism, there will be no tasks to be assigned and wait too long to be provided with the executor that stores its data. The improvement of work performance mainly comes from the shortening of the completion time of the input task in the map stage. The side effect of delay scheduling and network transfer of non-local are not taken into consideration by the Native scheduling in Spark, which explains the performance improvement of our strategy over baseline. As a result, Stargazer requires less time and more accurate to predict each task completion time and obtains a better scheduling efficiency than other methods.

5. Related work

In this paper, we summarize the related work into two categories: properly resource allocation for reducing task completion time and optimizing workload balancing to improve system performance.

5.1. Resource allocation

Properly resource allocation across various nodes has an important effect on big data platforms' systems. Efficient resource allocation can help improve the resources utilization and guarantee a fair distribution of resource, bring better performance for their schedules. Symbiosis [17], an online scheduler that predicted resource imbalance before launching tasks, and correct imbalance between computation-bound and network-bound tasks in the same executor process. [21] introduced a powerful and flexible new framework for scheduling concurrent distributed jobs based on the resource sharing model. To optimize a multiple objects function, [22] focused on assigning tasks belonging to multiple jobs across data centers, with the specific objective of achieving max–min fairness across sharing these data centers, in terms of their job completion times. dSpark [23] proposed a

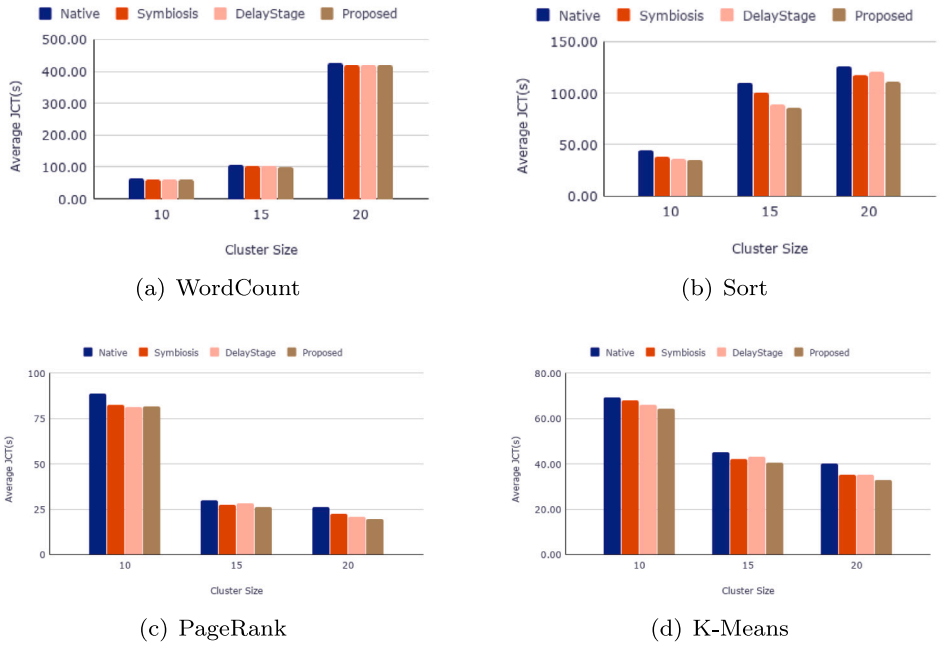


Fig. 9. Comparison of the average job completion times of the Native, Symbiosis, DelayStage scheduling and proposed approach on four types of popular analytics workloads, WordCount, Sort, PageRank, K-Means based on different clusters with 10, 15, 20 nodes respectively on Spark platform.

resource allocation model, which modeled the application completion time with respect to the number of executors and application input/iteration. DelayStage [18] introduced a simple yet effective stage delay scheduling strategy to interleave the cluster resources across the parallel stages, so as to increase the cluster resource utilization and speed up the job performance. Dagon [24] employed the sensitivity aware delay scheduling that prevents executors from long waiting for tasks insensitive to locality, and priority-aware caching that makes the cache eviction and prefetching decisions based on the stage priority determined by DAG-aware task assignment.

These approaches only considered the resource allocation properly overlooking heterogeneity of the nodes and the influence of the network.

5.2. Workload balancing

Distributing the received loads across computing nodes represented a crucial problem in big data platforms' systems [16]. An efficient workload balancing can help improve the resources utilization and guarantee a fair distribution of tasks to be processed, gaining a better performance for their schedules. [25] optimized and improves the current load balancing strategy of Spark based on the computing performance of each node in the Spark cluster, and proposes a task execution node assignment algorithm based on genetic algorithm and particle swarm optimization. [26] proposed a splitting and combination algorithm for skew intermediate data blocks, which can improve the load balancing for various reduce tasks. SASM [27] presented Spark Adaptive Skew Mitigation for automatic skew mitigation that is transparent to Spark users and existing Spark applications, which improved workload unbalance further. LIBRA [28] presented a lightweight strategy to address the data skew problem among the reducers of MapReduce [29] applications.

All of the above optimization approaches consider workload balancing, ignoring the heterogeneity of the nodes and the influence of network.

6. Conclusion

In this paper, we have designed, analyzed and evaluated Stargazer, a novel, efficient system that schedules diversity data analytics applications on heterogeneous resources by inferring the completion times of their decomposed tasks. Stargazer designed to improve job completion performance without breaking the current task locality mechanism. The upshot of Stargazer lies in its ability to predict each task completion time by collecting key factors and using deep learning method for training at runtime. Stargazer balance network transfer time and computation time for non-local tasks delay time scheduling, which decreases the overall completion time of a job and improves the utilization of cluster resources effectively. Finally, Our experiments on different sizes clusters demonstrate that Stargazer effectively reduces job completion times and without increasing extra scheduling delay. Stargazer can reduce the average job completion time by 21% and improve average performance by 20%, while incurring little cluster overhead. Furthermore,

we will open source code of our implementation and expect that the community will add support to more existing and future frameworks.

CRediT authorship contribution statement

Haizhou Du: Conceptualization, Methodology, Software. **Keke Zhang:** Data curation, Writing - original draft, Software, Validation. **Qiao Xiang:** Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Tolle KM, Tansley DSW, Hey AJ. The fourth paradigm: Data-intensive scientific discovery [point of view]. Proc IEEE 2011;99(8):1334–7.
[2] Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. Commun ACM 2008;51(1):107–13.
[3] He X, Shenoy P. Firebird: Network-aware task scheduling for spark using SDNs. In: International conference on computer communication and networks; 2016. p. 1–0.
[4] Jianchao T, Shuqiang Y, Chaoqiang H, Zhou Y. Design and implementation of scheduling pool scheduling algorithm based on reuse of jobs in spark. In: 2016 IEEE first international conference on data science in cyberspace (DSC). 2016, p. 290–5. http://dx.doi.org/10.1109/DSC.2016.81.
[5] Wang J, Zhou H, Hu Y, Laat Cd, Zhao Z. Deadline-aware coflow scheduling in a DAG. In: 2017 IEEE international conference on cloud computing technology and science (CloudCom). 2017, p. 341–6. http://dx.doi.org/10.1109/CloudCom.2017.55.
[6] Zaharia M, Chowdhury M, Das T, Dave A, Ma J, Mccauley M, Franklin MJ, Shenker S, Stoica I. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Usenix conference on networked systems design and implementation; 2012. p. 2–2.
[7] Merla P, Liang Y. Data analysis using hadoop mapreduce environment. In: 2017 IEEE international conference on big data (big data). 2017, p. 4783–5. http://dx.doi.org/10.1109/BigData.2017.8258541.
[8] Zhang X, Li Z, Liu G, Xu J, Xie T, Nees J. A spark scheduling strategy for haeterogeneous cluster. 55, 2018, p. 405–17. http://dx.doi.org/10.3970/cmc.2018.02527,
[9] Kannan G, Kamburugamuve S, Wickramasinghe P, Abeykoon V, Fox G. Task scheduling in big data - Review, research challenges, and prospects. 2017, p. 165–73. http://dx.doi.org/10.1109/ICoAC.2017.8441494.
[10] Khalil WA, Torkey H, Attiya G. Survey of Apache spark optimized job scheduling in big data. Int J Ind Sustain Dev 2020;1(1):39–48.
[11] Orhean AI, Pop F, Raicu I. New scheduling approach using reinforcement learning for heterogeneous distributed systems. J Parallel Distrib Comput 2018;117:292–302. http://dx.doi.org/10.1016/j.jpdc.2017.05.001.
[12] Ghodsi A, Zaharia M, Hindman B, Konwinski A, Shenker S, Stoica I. Dominant resource fairness: Fair allocation of multiple resource types. In: Nsdi, vol. 11. 2011, p. 24.
[13] Jiang J, Ma S, Li B, Li B. Symbiosis: Network-aware task scheduling in data-parallel frameworks. In: IEEE INFOCOM 2016-the 35th annual IEEE international conference on computer communications. IEEE; 2016, p. 1–9.
[14] Pati S, Mehta MA. Job aware scheduling in hadoop for heterogeneous cluster. In: 2015 IEEE international advance computing conference (IACC). 2015, p. 778–83. http://dx.doi.org/10.1109/IADCC.2015.7154813.
[15] Du H, Zhang K, Huang S. Octopusking: A TCT-aware task scheduling on spark platform. In: 25th IEEE international conference on parallel and distributed systems. IEEE; 2019, p. 159–62. http://dx.doi.org/10.1109/ICPADS47876.2019.00031.
[16] Soualhia M, Khomh F, Tahar S. Task scheduling in big data platforms: A systematic literature review. J Syst Softw 2017;134:170–89. http://dx.doi.org/10.1016/j.jss.2017.09.001.
[17] Das A, Lumezanu C, Zhang Y, Singh V, Jiang G, Yu C. Transparent and flexible network management for big data processing in the cloud. In: Presented as part of the 5th USENIX workshop on hot topics in cloud computing. San Jose, CA: USENIX; 2013, URL https://www.usenix.org/conference/hotcloud13/workshop-program/presentations/Das.
[18] Shao W, Xu F, Chen L, Zheng H, Liu F. Stage delay scheduling: Speeding up DAG-style data analytics jobs with resource interleaving. In: Proceedings of the 48th international conference on parallel processing; 2019. p. 1–11.
[19] Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, et al. Tensorflow: A system for large-scale machine learning. In: 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16); 2016. p. 265–83.
[20] Kumar Reddy R, Srinivasa Rao B, Raju KP. Handwritten hindi digits recognition using convolutional neural network with RMSprop optimization. In: 2018 Second international conference on intelligent computing and control systems (ICICCS). 2018, p. 45–51. http://dx.doi.org/10.1109/ICCONS.2018.8662969.
[21] Isard M, Prabhakaran V, Currey J, Wieder U, Talwar K, Goldberg A. Quincy:fair scheduling for distributed computing clusters. In: IEEE International conference on recent trends in information systems; 2009. p. 261–76.
[22] Chen L, Liu S, Li B, Li B. Scheduling jobs across geo-distributed datacenters with max-min fairness. In: INFOCOM 2017 - IEEE conference on computer communications. IEEE; 2017, p. 1–9.
[23] Islam MT, Karunasekera S, Buyya R. dSpark: Deadline-based resource allocation for big data applications in apache spark. In: 2017 IEEE 13th international conference on E-science (E-Science), At Auckland, New Zealand. 2017, p. 89–98. http://dx.doi.org/10.1109/eScience.2017.21.
[24] Xu Y, Liu L, Ding Z. DAG-aware joint task scheduling and cache management in spark clusters. In: 2020 IEEE international parallel and distributed processing symposium (IPDPS). IEEE; 2020, p. 378–87.
[25] Wang S, Zhang L, Zhang Y, Cao N. Spark load balancing strategy optimization based on internet of things. In: 2018 international conference on cyber-enabled distributed computing and knowledge discovery (CyberC); 2018. p. 76–3.
[26] Tang Z, Zhang X, Li K, Li K. An intermediate data placement algorithm for load balancing in Spark computing environment. Future Gener Comput Syst 2018;78:287–301. http://dx.doi.org/10.1016/j.future.2016.06.027.
[27] Yu J, Chen H, Hu F. SASM: Improving spark performance with adaptive skew mitigation. In: 2015 IEEE international conference on progress in informatics and computing (PIC). 2015, p. 102–7. http://dx.doi.org/10.1109/PIC.2015.7489818.
[28] Chen Q, Yao J, Xiao Z. LIBRA: Lightweight data skew mitigation in MapReduce. IEEE Trans Parallel Distrib Syst 2015;26:2520–33. http://dx.doi.org/10.1109/TPDS.2014.2350972.
[29] Merla P, Liang Y. Data analysis using hadoop mapreduce environment. In: 2017 IEEE international conference on big data (big data). 2017, p. 4783–5. http://dx.doi.org/10.1109/BigData.2017.8258541.