A Classification of MapReduce Schedulers in Heterogeneous Environments

Nenavath Srinivas Naik[1] and M. Badrinarayana[2]
[1]Asst. Professor, CVR College of Engineering/CSE Department, Hyderabad, India.
Email: nenavathsrinu@cvr.ac.in
[2]Professor, CVR College of Engineering/CSE Department, Hyderabad, India.
Email: badri@cvr.ac.in

***Abstract*—MapReduce is an essential framework for distributed storage and parallel processing for large-scale data-intensive jobs proposed in recent times. Intelligent scheduling decisions can potentially help in significantly minimizing the overall runtime of jobs. Hadoop default scheduler assumes a homogeneous environment. This assumption of homogeneity does not work at all times in practice and limits the MapReduce performance. In heterogeneous environments, the job completion times do not synchronize. Data locality is essentially moving computation closer (faster access) to the input data. Fundamentally, MapReduce does not always look into the heterogeneity from a data locality perspective. Improving data locality for MapReduce framework is an important issue to improve the performance of large-scale Hadoop clusters.**

This paper primarily provides an overview of the evaluation of Hadoop and introduces the MapReduce framework in detail. This paper also describes some relevant literature work on some recent developments in MapReduce scheduling algorithms in heterogeneous environments.

***Index Terms* -Hadoop, MapReduce, Job Scheduler, JobTracker, TaskTracker and Heterogeneous Environments.**

I. Introduction

The era of distributed and parallel computing had begun in the mid 1960s. During that period, to increase the computational speed, parallel processors were introduced. Later, Ethernet came into the picture that would transform the way data was distributed in a network with nodes working as processors of the parallel machine [1]. As the sequential architectures could not enhance the performance, there was a need for parallel and distributed architectures. In addition to this, the cost of hardware was also to be considered. Usually, it is better to introduce much cheaper parallel-working processors/multiple single-processor connected systems than making a single processor faster.

Nowadays, distributed systems replaced supercomputers as they are cheaper and became better alternatives for faster processing. A distributed system is a collection of multiple, single or multi-core, computers connected to a network that shares data and processing power to solve a given task effectively and efficiently. The motto of the distributed system is that the completion of the task becomes faster if it is shared among multiple processors than a single processor. It is important to observe that a distributed system can complete a task by executing it in the parallel fashion which is not possible with a single processor system [2].

MapReduce is now one of the most popular computational frameworks for large-scale data processing and analysis for parallel and distributed computing systems. MapReduce schedulers perform task assignment to available resources in the cluster. The common goal of MapReduce scheduling is to minimize the overall completion time of a job by appropriately assigning tasks to the available nodes [3].

The rest of this paper is structured as follows. Overview of Hadoop framework is presented in Section II. Section III presents a classification of MapReduce schedulers in heterogeneous environments. Different MapReduce scheduling problems are presented in Section IV. Finally, conclude the paper in Section V.

II. Overview of Hadoop Framework

Big Data depends very much on the capabilities of the systems for storage and processing. Traditional systems like Relational Database are not designed to handle smooth processing of Big Data [4]. In particular, the challenges of big data are those of processing and capturing of unstructured data. It is said many issues exist related to problems associated with sharing, transferring, analyzing, and visualizing of big data [5]. In traditional database systems, data is structured and is stored in tables with fixed number of columns. Each column has the particular data type. However, Big Data have a variety of data formats like audio, video, and text that does not fit in the table sizes.

Data-intensive applications are the ones that can process big data to get useful data [5]. To conduct the activities of data-intensive applications in parallel, requests devoting their time in the movement and manipulation of data are to be made. Movement of data involves the flow of data between two different nodes of a network for further processing.

However, as there is a lot of unstructured data growth, there is much demand for new processing frameworks to deal with it. For this purpose, several solutions emerge, including MapReduce. MapReduce is a framework for large-scale data processing developed by Jeffrey Dean and Sanjay Ghemawat [6] at Google. Now, it is one of the most popular frameworks for large-scale data processing and analysis.

Google trends show an increasing trend for the search term Hadoop worldwide [7]. The growing trend of Hadoop is caused by companies adopting this technology into their technology stack. Companies are now migrating towards Hadoop technology (eco-system) as it provides an easy and straightforward approach to access the data and to get the "Value" from Big Data.

In particular, the MapReduce framework [8] made a significant impact by demonstrating a simple, flexible and generic way of storing and processing large distributed datasets. MapReduce programs are written in a particular functional style and may be executed within a framework that automatically enables distributed and highly parallel execution. MapReduce was quickly embraced as a new paradigm for data-intensive computing and widely adopted by other companies working with web-scale data sets. For example, Hadoop is currently used by major companies such as Amazon, eBay, Facebook, Twitter, Yahoo! and many others [10].

A. Hadoop

The importance of Hadoop is due to its large scale adoption in clusters with commodity systems. Use of Hadoop has spread from large corporations with expensive server farms to small business and academia for research and other data processing tasks. It also indicates a shift from homogeneous to heterogeneous computing environments that are small or medium scale and thrifty.

Figure 1 depicts the general Hadoop cluster. It is composed of nodes, racks, and switches. All nodes in a rack are connected to a rack switch, and all rack-switches are then connected via bandwidth links to the core switch. There are two branches of Hadoop releases, i.e. Hadoop1 and Hadoop2. Hadoop1 is the most famous for batch processing and shows the potential value of Big Data distributed processing. Hadoop2 or YARN (Yet Another Resource Negotiator) [11] provides a unified resource management framework for different data processing platforms. Similar to the original Hadoop framework, the YARN framework also has a centralized manager node running the Resource Manager (RM) daemon and multiple distributed working nodes running the Node Manager (NM) daemons.

However, there are two main differences between the design of YARN and original Hadoop. First, the Resource Manager (RM) in YARN no longer monitors and coordinates job execution as the *JobTracker* of traditional Hadoop do. An Application Master (AM) is generated for each application in YARN, which generates resource requests, negotiates resources from the scheduler of RM and works with the Node Managers to execute and to monitor the similar application tasks.

The Hadoop1 framework contains two components [12]: 1. HDFS (Hadoop Distributed File System), which stores the input data 2. MapReduce engine, which processes the data blocks stored in different nodes of a cluster. HDFS contains a *NameNode* and *DataNodes* in a cluster. *NameNode* is a master node which contains the meta-data information of the data block locations in a cluster. *DataNodes* are slave nodes, which store the data blocks within a cluster. MapReduce contains a *JobTracker* and multiple *TaskTrackers*. *JobTracker* deals with job scheduling and assigns tasks to *TaskTrackers* within a cluster depending on the slot availability. *TaskTrackers* process the *map* and *reduce* tasks on the corresponding nodes in the cluster.

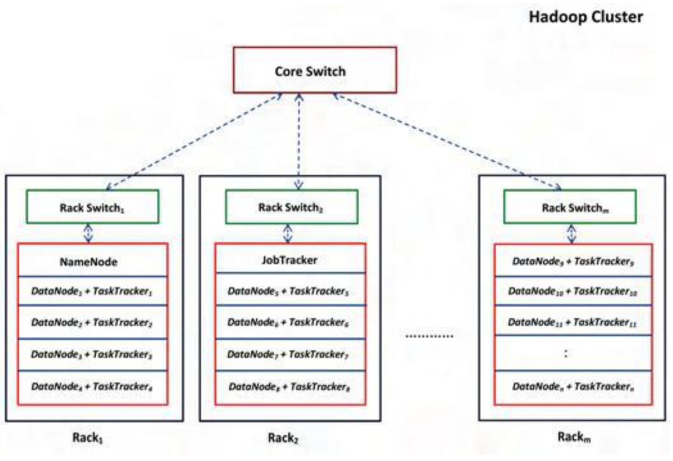
Hadoop Cluster
Core Switch
Rack Switch$_1$
NameNode
DataNode$_1$ + TaskTracker$_1$
DataNode$_2$ + TaskTracker$_2$
DataNode$_3$ + TaskTracker$_3$
DataNode$_4$ + TaskTracker$_4$
Rack$_1$
Rack Switch$_2$
JobTracker
DataNode$_5$ + TaskTracker$_5$
DataNode$_6$ + TaskTracker$_6$
DataNode$_7$ + TaskTracker$_7$
DataNode$_8$ + TaskTracker$_8$
Rack$_2$
Rack Switch$_m$
DataNode$_9$ + TaskTracker$_9$
DataNode$_{10}$ + TaskTracker$_{10}$
DataNode$_{11}$ + TaskTracker$_{11}$
DataNode$_n$ + TaskTracker$_n$
Rack$_m$

Figure 1. A simplified schematic of a general Hadoop cluster [9]

B. MapReduce

MapReduce framework was inspired from the functional programming languages [14]. The input and output data have a particular format of (*key*, *value*) pairs. The Users define an application using two functions: the *Map* function and the *Reduce* function. The *Map* function repeats over a set of the input (*key*, *value*) pairs and produces intermediate output (*key*, *value*) pairs. The MapReduce library groups all *intermediate values* by *key* and gives them to the *Reduce* function. The *Reduce* function iterates over the *intermediate values* associated with the same *key*. Then it produces zero or more output (*key*, *value*) pairs.

MapReduce framework is most widely used across the industry and academia. It has been in practice in many domains of data-intensive applications such as web data mining, machine learning, health care data analysis, and scientific simulation. MapReduce is highly scalable and enables thousands of commodity computers to be used as an efficient computing platform. The framework detects and handles node failures automatically without allowing the overall execution process to be affected.

MapReduce framework follows a simple master-slave architecture, where *JobTracker* is the master node and *TaskTrackers* are the slave nodes. The *JobTracker* handles scheduling decisions for the MapReduce jobs. The *JobTracker* in Hadoop is designed in such a way that the schedulers can be pluggable in and out. The *JobTracker* and *TaskTrackers* communicate with each other through heartbeat messages. Through these messages, the *TaskTracker* indicates to the *JobTracker* that it is alive. As a part of the heartbeat message, the *TaskTracker*

DOI:10.32377/cvrjst1216

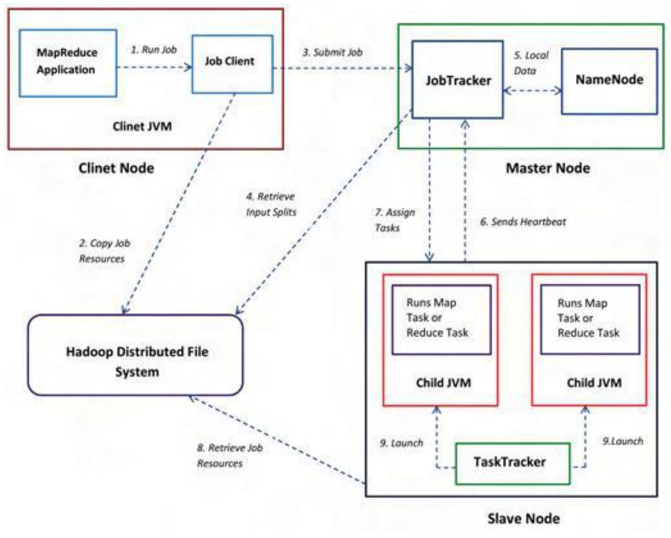

Figure 2. Hadoop MapReduce workflow [13]

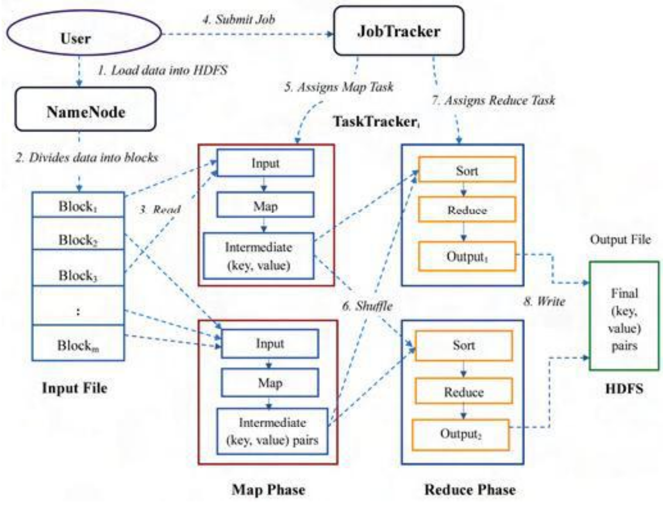

Figure 3. MapReduce Jobflow [15]

also indicate whether it is ready to accept a new task and other related information. The client application submits jobs to the *JobTracker*. The *JobTracker* communicates to *NameNode* to find the data location. The *JobTracker* then creates a *map* task for each data block. These tasks are queued up in the *JobTracker* as per the scheduling algorithm. Whenever the *TaskTracker* requests for a task, the *JobTracker* submits a task to it as a return value. The *TaskTracker* launches each task in a new Java Virtual Machine. A *TaskTracker* can run multiple tasks at an instant which can be configured through the configuration parameters. Once all the job tasks are finished execution, then the output is stored back in HDFS.

We broadly describe MapReduce workflow as shown in Figure 2 with the following sequence of steps.

1) The client submits the MapReduce job to the *JobClient*.
2) The *JobClient* copies the information regarding the job resources to HDFS.
3) The *JobClinet* internally submits the job to the *JobTracker*.
4) The *JobTracker* retrieves corresponding input data blocks of the job.
5) The *JobTracker* interacts with the *NameNode* for metadata information of the input data.
6) The *TaskTracker* periodically sends the heartbeat information regarding the availability of slots and task progress to the *JobTracker*.
7) The *JobTracker* assigns tasks to the *TaskTrackers*.
8) The *TaskTracker* retrieves job resources from the HDFS.
9) Finally, *TaskTracker* launches the child JVM, and it executes the *map* or *reduce* task.

MapReduce contains the following stages [15] when scheduling a job from the master node to the slave nodes as shown in Figure 3.

1) User submits input data to the *NameNode*
2) The *NameNode* divides the data into m blocks of the same size. r copies (replicas) of every block are produced for fault tolerance. (r is the replication factor).
3) The master node picks up the idle slave nodes to schedule map and $reduce$ tasks.
4) User submits a job to the *JobTracker* for processing corresponding data blocks.
5) A slave node that is executing a map task parses the data block and assigns each (key, $value$) pair to the Map function. The intermediate (key, $value$) pairs are buffered in memory at corresponding nodes.
6) The buffered (key, $value$) pairs are written to the data residing nodes at fixed intervals and divided into R sections by means of a (configurable) partition function (default is *hash (intermediate key) mod R)*. The identical (key, $value$) pairs go to the same partition. When the task completes, the slave node sends the location information of partition to the master node.
7) The node which contains $Reducer$ function reads the data using remote procedure calls. It sorts and groups the data by *intermediate key* so that all values of the same key are grouped. It is called shuffling of the task.
8) The master node produces the final output after execution of all map and $reduce$ tasks.

In the MapReduce framework, the job execution process has two phases, namely, a *Map* phase and a *Reduce* phase. The *Map* phase assigns each *map* task to a block of the input data. The number of data blocks determines the number of *map* tasks. Execution of a *map* task consists of the following steps [16]:

1) The task's slice is read and organized into records (*key, value*) pairs, and the *Map* function are applied to each block.
2) After the *Map* function's completion, the *commit* phase registers the final output with the *TaskTracker*, which

DOI:10.32377/cvrjst1216

notifies the *JobTracker* about the task's completion.

3) The Output Collector stores the *Map* output in *Intermediate keys*
4) The Output Collector 'spills' this information to the disks. A spill of the in-memory buffer contains sorted records; first by partition number, and second by (*key, value*) pairs. The buffer information is written to a local file system as a data file and an index file.
5) In the commit phase, the final output of a *map* task is produced by integrating all the split files created by this *map* task into a single pair of data and index files.

These files are recorded with the *TaskTracker* before the task is completed. *TaskTracker* reads these files to service requests from *reduce* tasks.

Reduce phase contains following three stages: shuffle, sort, and reduce.

1) In the *shuffle* stage, the intermediate output produced by the *Map* phase is collected. Every *reduce* task is allotted a part of the intermediate output with a static *key* range.
2) In the *sort* stage, output with the same *key* are grouped together to be processed by the *reduce* stage.
3) In the *reduce* stage, the user-defined *Reduce* function is applied to every *key* and corresponding list of *values*.

III. Scheduling Algorithms for Hadoop MapReduce

We present a mathematical model to describe general scheduling problems in a Distributed Environment.

Let $M = \{M_1, M_2, ..., M_m\}$ be the set of machines or computer nodes which have to process n jobs represented by the set $J = \{J_1, J_2, ..., J_n\}$. A job J_i usually consists of several tasks $T_1, T_2, ..., T_k$. Each task T_i consists of n_i operations $O_{i1}, O_{i2}, ..., O_{in}$. To every O_{ij} a process requirement p_{ij} is associated. Each operation O_{ij} is associated with a set of parallel machines M_{ij}.

The goal is to design a scheduler, which is responsible for making decisions to execute a task at some time and on some machine. The most common objective of scheduling is to reduce the completion time of a parallel application by properly assigning the tasks to the processors. Inappropriate scheduling of tasks would fail to exploit the true potential of the system [17].

The MapReduce task scheduling is an NP-Hard problem [19] as it needs to achieve a balance between the job performance, data locality, user fairness or priority, resource utilization, network congestion, and reliability. If the scheduling policy considers data locality for selecting a task, it may have to compromise on the fairness as the node available may have data of some job, which is not on head-of-line as per the fairness policy. Similarly, if a task is scheduled based on job's priority, it is not necessary that it would have local data on the available node. It would impact job performance and

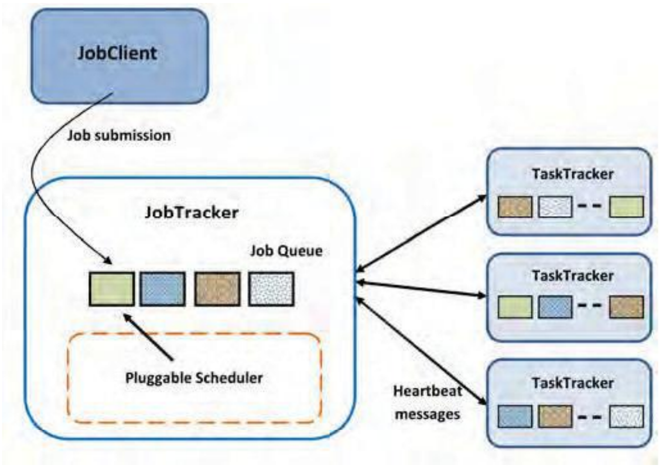

Figure 4. Adding pluggable scheduler [19]

network utilization. Some data centers or users want to achieve higher performance; some want high data locality, some want to improve resource utilization and so on. The scheduling policies need to be designed differently for achieving different objectives in different scenarios.

The demand for scheduling adaptation in MapReduce comes from following points: the heterogeneity of cluster nodes, the data locality-awareness, and the diversity of job execution times. Job scheduling or task scheduling in the MapReduce is employed to manage workload efficiently between the computing nodes and effectively share the resources of a Hadoop cluster among various jobs and nodes [18]. The performance of the Hadoop framework can be affected by the imbalance workload distribution and partial resources sharing because of not having a sophisticated scheduling mechanism.

At each heartbeat, the *TaskTracker* notifies the *JobTracker* the number of available slots it currently has. MapReduce slots define the maximum number of *map* and *reduce* tasks that can run in parallel on a cluster node. The number of slots depends on the number of cores on that node. The *JobTracker* assigns tasks depending on job's priority, the number of non-running tasks and potentially other criteria. Since bug report Hadoop-3412, Hadoop has been modified to accept pluggable schedulers as shown in Figure 4 that allows the use of new scheduling algorithms to help optimize jobs with different specific characteristics.

A. Taxonomy for MapReduce Scheduling Algorithms

A significant characteristic of scheduling algorithms is their runtime performance. It means just how schedulers adapt to the heterogeneous cluster. Schedulers can run in a static (non-adaptive) or dynamic (adaptive) environments at runtime [9]. The dynamic nature of MapReduce framework is centered on several things, like resource, data, workload, and the job. We categorize the scheduling algorithms into adaptive and non-adaptive based on their runtime flexibility. The classification of MapReduce schedulers is as shown in Figure 5.

In **Non-adaptive** algorithms, scheduling, order for users,

DOI:10.32377/cvrjst1216

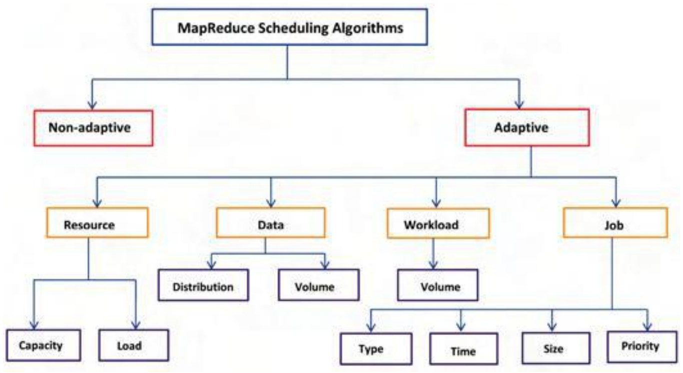

Figure 5. Taxonomy for MapReduce scheduling algorithms

jobs and tasks are static at runtime based on predefined policies. For example, The Hadoop first-in-first-out and priority-based schedulers practice predefined policies for scheduling jobs or tasks in order.

In a MapReduce system, the features of the data, physical resources, and workload vary at runtime. An **Adaptive** scheduling algorithm assigns a task to a node based on the availability, capacity, and load on nodes in the cluster at runtime. It may additionally choose an ideal task assignment approach predicated on the dynamic arrival attributes of the workload. We further categorize the algorithms based on the runtime features of the MapReduce framework.

1) Resource-Adaptive: The resources existing in a Hadoop cluster have distinct features. A CPU resource can be categorized by its processing speed, network bandwidth, and a system can be considered by grouping its CPU speed, RAM, and disk storage capacity. An additional essential characteristic of a resource is the processing load. Processing load of a resource is used to decide the available capacity of the resource. Some of the schedulers execute tasks based on the nodes according to their speeds to guarantee fast execution and approximately based on the existing capacity to avoid resource conflicts.

a) Resource Capacity: The capacity of a system contains the resources like the size of RAM, the number of CPUs, and disk capacity along with their processing speed. The systems used in heterogeneous clusters may have a varied capacity of these resources. Therefore, a scheduler has to reflect this while launching different kinds of tasks on them.

b) Resource Load: Overloading of systems usually has consequences of longer execution times and failures as a result of resource conflicts. Overloading moreover raises the temperature of the system and consumption of power. Therefore, schedulers have to adapt to decrease overloading and avoid resource skewness.

2) Data-Adaptive : To minimize the data transfers delays and network conflicts, a scheduler has to perform a significant part in getting the execution of a task nearer to the data residing node. Therefore, while scheduling tasks, a scheduler is necessary to be responsive to the data location and the data volume.

a) Data Distribution: The Hadoop distributed file system divides the input data into multiple blocks and distributes various replicas of blocks across existing nodes. The MapReduce framework executes a task in parallel on these distributed data blocks. The MapReduce scheduler requires being responsive to the location of data blocks to exploit the data local task executions. The scheduling of the *reduce* task has to be done based on the intermediate data locations. Therefore, the nodes that run local *map* tasks of the job must be set preference.

b) Data Volume: The MapReduce jobs are executed on the nodes of the cluster that store a massive amount of data. If a scheduler recognizes the nodes having the input data of a given job correctly, then the performance of a job can be enhanced. The chances of identifying a right data block on a node are less if the amount of data to be processed is enormous. Hence, to quickly recognize the local data nodes for a job, the scheduler has to be adaptive in nature.

3) Workload-Volume-Adaptive: In distributed and parallel environments several jobs are executed at a time. The volume of data turns out to be important to ensure Quality of Service necessities of response time and throughput.

4) Job-Adaptive: The important features of a MapReduce job are its type, priority, size, and execution time. A scheduler can adapt to one or more features as job contains a *map* and *reduce* tasks, its type and time are resulting from the corresponding task features.

a) Job Type: A job can be I/O-intensive or CPU-intensive, or a combination of both. If the *map* tasks of two CPU-intensive jobs are scheduled on a node, then both the jobs will contend for the allocation of CPU, and it can take a longer execution time. However, if the tasks of a CPU and I/O-intensive jobs are scheduled on a node, they can execute in parallel.

b) Job Time: A scheduler has to confirm that at least a few tasks of a job are continuously executed in the cluster. For that, it has to consider the response time, task execution time, and the actual response time with different task schedules.

c) Job Size: The job sizes are characteristically measured as the number of *map* and *reduce* tasks essential to complete that job. These tasks are subject to the size of data to be executed by that job. The scheduler decides each job task be scheduled based on their size.

d) Job Priority: The MapReduce framework runs different job types. Scheduling algorithms should be capable of scheduling jobs and tasks depending on their priorities. These priorities can be fixed by the administrator depending on the user inputs. In a few circumstances, priorities can be identified by the scheduler depending on further requests, for instance, fairness and response times.

IV. MapReduce Scheduling Issues

This section defines data locality, speculative execution, and heterogeneity regarding the MapReduce framework because these are the important performance issues for MapReduce

DOI:10.32377/cvrjst1216

scheduling algorithms. We present the classification of literature as shown in Figure 6.

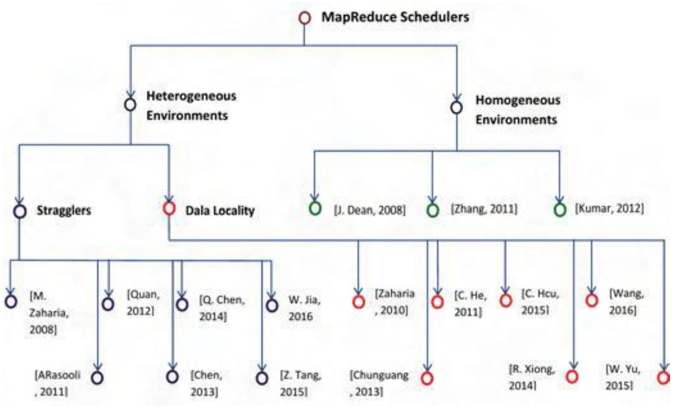

Figure 6. A Classification Literature of MapReduce Scheduling Approaches

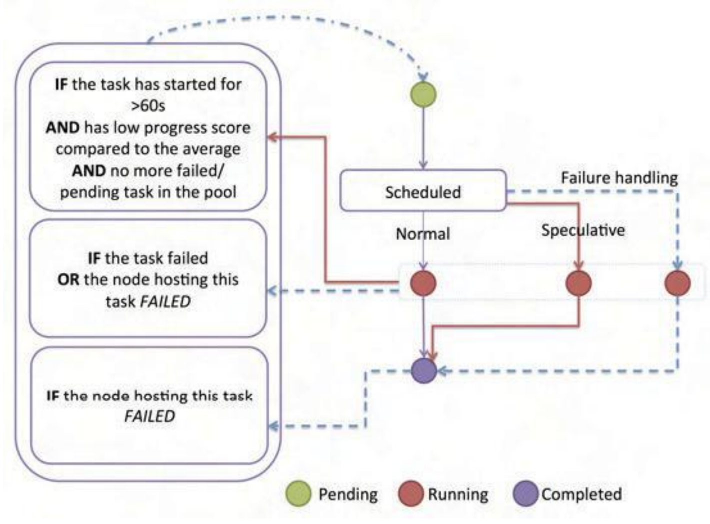
IF the task has started for >60s AND has low progress score compared to the average AND no more failed/ pending task in the pool
IF the task failed OR the node hosting this task FAILED
IF the node hosting this task FAILED
Scheduled
Failure handling
Normal
Speculative
Pending
Running
Completed

Figure 7. Life cycle of a task in Hadoop

A. Data locality

To process an enormous amount of data, Hadoop has to provide an efficient data locality scheduling approach for improving its performance in a heterogeneous environment. One of the Hadoop principles is that moving computation is cheaper than moving data [17]. This principle says that it is better to move the computation close to data location rather than to move data to the computing node. It applies in particular when the data size is vast because the moving of computation minimizes the network congestion and enhances the system performance. Data locality is about executing tasks to their input data as close as possible. These days clusters have many nodes and transmit large data that enforce network load and create congestion. Therefore, designing a scheduler that can avoid unnecessary data transmission in the Hadoop cluster is a crucial factor for the MapReduce performance as network bandwidth is a scarce resource for these systems.

For each node, all tasks are categorized according to the distance between the input data node and computation node. The best efficient locality is where the task processing is done on the same node with corresponding input data block named as node-level locality. When a task cannot achieve the node level locality, then scheduler executes the task on the node where the processing node and data node located in the same rack named as rack-level locality. If the task still fails to achieve the rack-level locality, then the scheduler schedules the task on a node in another rack which is named after off-rack-level locality. If the data locality is not accomplished, transferring of data and I/O cost can adversely affect the performance due to shared network bandwidth.

B. Speculative Execution

One of the common causes that prolong the execution of a MapReduce job is a “straggler”. A Straggler in simple terms is a task (*map* or *reduce*) that takes longer execution time as compared to other tasks in the cluster. Straggler tasks can cause resource wastage and hamper the performance of a job in the cluster, i.e. if a single task is delayed it affects the execution time of the whole job. It is commonly seen that the longer that a job runs, the chances of its getting delayed increase. The causes can be many; malfunctioning of hardware, bugs in software and configurations, different hardware and dynamic aspects like CPU time variability, network traffic, disk contention, etc. These can be difficult to identify as in most of the cases; jobs execute effectively. As the stragglers may take longer execution time, MapReduce goes for a redundant speculative execution on other nodes in an attempt to minimize job execution time as shown in Figure 7.

MapReduce executes a speculative copy of its task (also called a backup task) on another node to finish the computation faster. The objective of speculative execution is to minimize the job execution time. A speculative task is executed based on a simple heuristic that compares each task progress to the average progress of a job. If the *JobTracker* finds that the task is running very slow (about other tasks of the same job) or showing minimal progress over time, it schedules a copy of the same task on another node without suspending the initial task. Moreover, both the tasks run simultaneously and separately. When a task executes well, then any duplicate tasks that are running are dismissed as they are not required. Therefore, if the original task completes its execution before the speculative task, then the speculative task is terminated. Conversely, if the speculative task finishes first, then the original task is terminated, which is termed as speculative execution.

Scheduling of speculative tasks is complicated as it is difficult to differentiate the tasks that are marginally slower than the average completion time, especially in heterogeneous environments. If straggler tasks are identified initially, then the completion time of a job can be minimized. Even though MapReduce schedulers attempt to launch backup tasks for

DOI:10.32377/cvrjst1216

stragglers, they fail to identify correct straggler tasks because of errors and difficulties in estimating the remaining execution time of tasks. Wrong identification of straggler tasks eventually gives rise to two problems; First, the execution times of real stragglers are extended because launching a backup task for wrongly identified stragglers will not improve the MapReduce performance. Second, the system resources are misused when launching backup tasks for wrong stragglers. Straggler tasks are undesirable since they extend the job execution time and thus degrade the performance of the MapReduce framework.

C. *Heterogeneity*

Hadoop was initially aimed for homogeneous cluster environments, but now it is commonly used in various heterogeneous environments [19]. The homogeneity assumption is that all the nodes in the cluster will have the same processing capacity. This assumption can degrade the MapReduce performance because there is certain diversity in the hardware. In current day scenarios, financially constrained entities like universities and colleges would like to have a cluster with a mix of legacy hardware with newer ones. Advancement of hardware technology is another practical reason for heterogeneous clusters to increase, as hardware sourced at different times in technology cycles can be brought together in a better way. Therefore, coping with heterogeneous cluster hardware would be a major goal to increase the scope of MapReduce. In a heterogeneous environment, it is important to schedule a job with proper resources to achieve high performance. MapReduce jobs have heterogeneous resource demands as jobs may be CPU or I/O-intensive.

Heterogeneity is categorized as below which is increasing in both workloads and cluster configurations.

1) In Heterogeneous environments, each node in the cluster can have different physical parameters such as processing speed and disk capacity.
2) MapReduce jobs can be heterogeneous on various task features such as data, the number of tasks, job execution times, and computation requirements.

However, current MapReduce schedulers are not correctly adapted for heterogeneous systems. Research in this paper is originally motivated by addressing the scheduling challenges arising due to increase in the heterogeneity of distributed systems. This system introduces novel scheduling challenges and directly affects the system performance.

V. Conclusions

This paper presented background work on the evaluation of Hadoop framework. A detailed explanation of the MapReduce framework is given and presented a taxonomy of MapReduce scheduling algorithms. This paper also reviewed some of the MapReduce scheduling approaches for heterogeneous environments which are related to the job or task scheduling, speculative execution, and data locality.

References

[1] J. Dean and S. Ghemawat. “MapReduce: Simplified data processing on large clusters”. In *Communications of the ACM*, volume 51, number 1, pages 107–113, January 2008.

[2] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. “The performance of MapReduce: an in-depth study”. In *Proc. VLDB Endow.*, volume 3, number 1–2, pages 472–483, September 2010.

[3] J. Tan, X. Meng, and L. Zhang. “Delay tails in MapReduce scheduling”. In *SIGMETRICS Perform. Eval. Rev.*, volume 40, number 1, pages 5–16, June 2012.

[4] T. White. “Hadoop: The Definitive Guide”. In *OReilly Media*, 2015.

[5] J. Xie, F. Meng, H. Wang, H. Pan, J. Cheng, X. Qin. “Research on scheduling scheme for Hadoop clusters”. In *Procedia Computer Science*, volume 18, pages 24682471, June 2013.

[6] MD. Assuncao, RN. Calheiros, S. Bianchi, M. Netto, R. Buyya. “Big data computing and clouds: trends and future directions”. In *Journal of Parallel and Distributed Computing (JPDC)*, volumes 79–80, pages 3–15, May 2015.

[7] T. Chih-Fong, L. Wei-Chao, K. Shih-Wen. “Big data mining with parallel computing: A comparison of distributed and MapReduce methodologies”. In *Journal of Systems and Software*, volume 122, pages 83–92, December 2016.

[8] S. Y. Hsieh, C. T. Chen, C. H. Chen, T. H. Yen, H. C. Hsiao, R. Buyya. “Novel Scheduling Algorithms for Efficient Deployment of MapReduce Applications in Heterogeneous Computing Environments”. In *IEEE Transactions on Cloud Computing*, volume PP, number 99, pages 1-1, April 2016.

[9] A. Rasooli and D. G. Down. “COSHH: A classification and optimization based scheduler for heterogeneous Hadoop systems”. In *Journal of Future Generation Computer Systems*, volume 36, pages 1–15, July 2014.

[10] Q. Chen, C. Liu and Z. Xiao. “Improving MapReduce Performance Using Smart Speculative Execution Strategy”. In *IEEE Transactions on Computers*, volume 63, number 4, pages 954-967, April 2014.

[11] J. Lu and J. Feng. “A survey of MapReduce based parallel processing technologies”. In *China Communications*, volume 11, number 14, pages 146-155, September 2014.

[12] C. Hsu, K. Slagter, and Y. Chung. “Locality and loading aware virtual machine mapping techniques for optimizing communications in MapReduce applications”. In *Future Generation Computer Systems*, volume 53, pages 43–54, December 2015.

[13] K. Slagter, C. Hsu, and Y. Chung. “An Adaptive and Memory Efficient Sampling Mechanism for Partitioning in MapReduce”. In *International Journal of Parallel Programming*, volume 43, issue 3, pages 489–507, June 2015.

[14] Z. Tang, L. Jiang, J. Zhou, and K. Li. “A self-adaptive scheduling algorithm for reduce start time”. In *Future Generation Computer Systems*, volume 43, pages 51–60, February 2015.

[15] W. Yu, Y. Wang, X. Que and C. Xu. “Virtual Shuffling for Efficient Data Movement in MapReduce”. In *IEEE Transactions on Computers*, volume 64, number 2, pages 556-568, February 2015.

[16] L. Mashayekhy, M. M. Nejad, D. Grosu, Q. Zhang and W. Shi. “Energy-Aware Scheduling of MapReduce Jobs for Big Data Applications”. In *IEEE Transactions on Parallel and Distributed Systems*, volume 26, number 10, pages 2720-2733, October 2015.

[17] W. Jia and L. Xiaoping. “Task Scheduling for MapReduce in Heterogeneous Networks”. In *Cluster Computing*, volume 19, number 1, pages 197–210, March 2016.

[18] M. C. Lee, J. C. Lin and R. Yahyapour. “Hybrid Job-Driven Scheduling for Virtual MapReduce Clusters”. In *IEEE Transactions on Parallel and Distributed Systems*, volume 27, number 6, pages 1687-1699, June 2016.

[19] P. P. Nghiem, S. M. Figueira. “Towards efficient resource provisioning in MapReduce”. In *Journal of Parallel and Distributed Computing*, volume 95, pages 29–41, September 2016.

DOI:10.32377/cvrjst1216