IJDI-ERET

INTERNATIONAL JOURNAL OF DARSHAN INSTITUTE ON ENGINEERING RESEARCH AND EMERGING TECHNOLOGIES
Vol. 10, No. 2, 2021
www.ijdieret.in

(Review Article)

# Analysis of MapReduce operation in Hadoop YARN and Rack-Aware Resource Management System for YARN

**T. Moses[1*], T. A. Badmos[2], R. Abdulkarim[3]**

[1*3]*Department of Computer Science, Faculty of Computing, Federal University of Lafia, Nasarawa State - Nigeria*
[2]*Industy and Innovation Institute, Sheffield Hallam University, United Kingdom*

***Abstract***

*Hadoop MapReduce has been the major computational paradigm for the analysis and exploration of big data. It facilitates concurrent processing of big data by splitting data into chunks and processing these data using commodity cluster processors. The processed data are aggregated from the multiple commodity clusters to return a consolidated output. Hadoop YARN has been a major enabler for this computation but its central resource manager is a bottleneck. Rack-aware resource management system developed to overcome this bottleneck decentralized the responsibilities of the resource manager by providing another layer in the architecture of Hadoop called Rack Unit Resource Manager Layer. This work, therefore, analyses the MapReduce operation between these two architectures to understand the behaviour of each data chunk (block). Wordcount operation was used for this analysis and the result obtained showed that the rack-aware system performed better as data grows bigger.*

*Keywords: MapReduce, Hadoop YARN, Rack-Aware system, Resource Manager, Data Chunk, Rack Unit Resource Manager*

## 1. Introduction

Hadoop Yet Another Resource Manager (YARN) is an open-source Apache Software Foundation (ASF) project which was written in Java programming language that provides cost-effective and scalable infrastructure for distribution and parallel processing of large datasets across commodity of clusters [1]. The programming paradigm was inspired by Google File System (GFS) [2] and Google's MapReduce distributed computing environment. The idea was first conceived by Dough Cutting, and together with Professor Mike Cafarella of the University of Michigan, developed Hadoop later called Apache Hadoop [3]. Hadoop was named after Dough Cutting's son toy elephant [4]. Hadoop has been used to process highly distributable problems across a large number of datasets with commonly available, inexpensive internal disk drives [5]. There are two components in Hadoop (i) Hadoop Distributed File System (HDFS) and (ii) MapReduce framework.

HDFS is a master/slave architecture consisting of NameNode called master, a secondary node called a checkpoint, and several DataNodes called slaves [5].

[*]*Corresponding Author: e-mail: visittim@yahoo.com*
*Tel- +234 7065538559*
ISSN 2320-7590


The major/centralized controller that handles all file system operations is the NameNode hence; any request to the file system (like create, delete and read a file) must go through the NameNode. NameNode also handles block mappings of input files. Each file is divided into blocks (default is 64MB) with each independently replicated across DataNodes for redundancy. Block creation, deletion, and replication are managed by the DataNode upon instruction from the NameNode [5]. A periodic heartbeat message is always sent from the DataNodes to NameNode (usually, default heartbeat is 3s) to be sure that there is no loss of connectivity between the two. If NameNode is unable to establish this periodic heartbeat from DataNode, it considers such DataNode out of service, unavailable, or dead and hence, will not forward any new request to such DataNode. The NameNode at this point schedules the creation of new replicas of those blocks in the unavailable DataNode on another DataNode [5]. Because NameNode is the central coordinator for data block creation, deletion, and replication, a performance bottleneck is possible. [7] addressed this performance bottleneck by proposing a rack-aware model for high availability of HDFS.

Hadoop YARN MapReduce is also based on centralized master/slave architecture. Resource Manager of YARN runs as a daemon on a dedicated machine and acts as central authority managing resources among various competing nodes in the cluster. Job submission to the resource manager passes through a public submission protocol and goes through an admission control phase. This is done to make sure that

security credentials are validated and various operational and administrative checks are performed [6]. Once a job is accepted, a scheduler in YARN is triggered. If the scheduler has enough resources, the application is moved from an accepted state to a running state. Apart from internal bookkeeping that helps in task management, the next step is allocating a container for the application master and spawning it on a node in the cluster [6]. The centralized resource manager is also a performance bottleneck. To solve this, [8] developed a rack-aware system for the management of resources in Hadoop. The system decouples the responsibilities of the resource manager by providing another layer where each daemon called Rack_Unit Resource Manager (RU_RM) carries out the responsibility of allocating resources to compute nodes within its local rack. This ensures low latency for large files on compute nodes within the same local rack. This work, therefore, analyses the MapReduce operation on each data block in these two architectures to know which architecture is better looking at the data to be analyzed.

## 2. Literature Review

*2.1 Analysis of Hadoop YARN:* Resource Manager of YARN runs as a daemon on a dedicated machine and acts as central authority managing resources among various competing nodes in the cluster (see Figure 1). Resource manager enforces rich familiar properties such as fairness and locality across commodity servers. Base on the need for an application, scheduling priorities, and availability of resources, the resource manager dynamically allocates leases called containers to applications to run on particular nodes in the cluster [6].

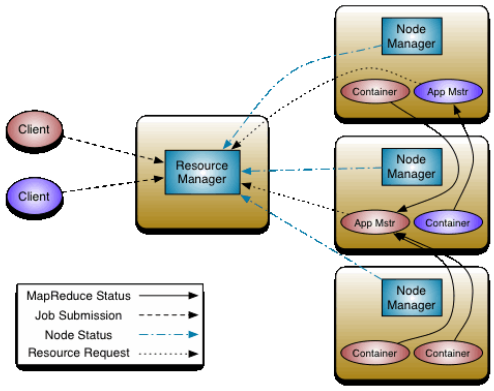


**Figure 1:** Architecture of YARN (Vinod *et al*., 2013)

Containers are a logical bundle of resources (e.g. 4GB of RAM, 2CPU) bound to a particular node (Vinod *et al*., 2013). To track the number of containers in the cluster, the resource manager uses a special system daemon called Node Manager (NM) running on each commodity server. The Node Managers are responsible for the monitoring of resource availability, containers lifecycle management (e.g. start, kill), and reporting any possible fault to the resource manager. Namenodes achieve this through a heartbeat communication protocol (Ibrahim *et al*., 2016).

Job submission to the resource manager passes through a public submission protocol and goes through an admission control phase. This is done to make sure that security credentials are validated and various operational and administrative checks are performed (Vinod *et al*., 2013). Once a job is accepted, a scheduler in YARN is triggered. If the scheduler has enough resources, the application is moved from an accepted state to a running state. Apart from internal bookkeeping that helps in task management, the next step is allocating a container for the application manager and spawning it on a node in the cluster [6]. A record of accepted job is normally written to persistent storage and recovered in case of RM restart or failure.

The application manager serves as the "head" of a job [6]. It manages all lifecycle aspects of the job including dynamically increasing and decreasing resource consumption, controlling the flow of execution (mappers and reducers), handling faults and computation skew, and also perform local optimizations of commodity servers [9]. Delegating all these functions to the application manager helps YARN architecture gained a great deal of scalability, flexible programming model, and improved upgrade/testing capability. To complete a job, the application manager will need to harness resources like CPU, RAM, and disk available on multiple nodes. For application manager to obtain a container, it issues a resource request to the resource manager with the specification of locality preference and what properties a container should possess. Once a resource is released on behalf of an application manager, the resource manager will generate a lease for the resource, which is pulled by a subsequent application manager heartbeat. To guarantee authenticity when the application manager presents the container lease to the resource manager, a token-based security mechanism is put in place [6]. Once the application manager discovers that a container is ready for its use, it encodes an application-specific launch request with the least [9]. A running container communicates with the application manager through this application-specific protocol to report status and liveliness and receive framework-specific commands. In all of these, the resource manager serves as a central arbiter for successful MapReduce operation in the architecture.

*2.2 Analysis of Rack-aware system for Hadoop YARN:* The rack-aware system for Hadoop YARN decentralized the global control of Resource Manager in the YARN framework by providing another layer called Rack Unit Resource Manager (RU_RM) layer. This layer aimed to make compute nodes on each rack be controlled by their corresponding Rack Unit Resource Manager instead of a single Resource Manager controlling all the compute nodes in the network [8]. The architecture of the system is shown in Figure 2. The system

parallelizes the global control of the Resource Manager in the YARN framework.

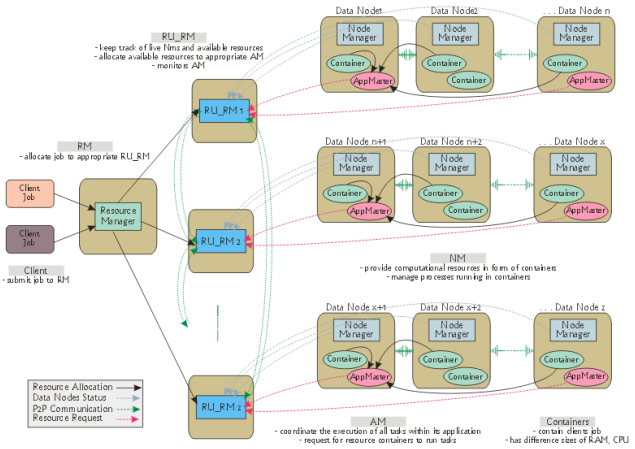


**Figure 2:** Architecture of rack-aware model [8]

Central Resource Manager obtain information on which rack contains 2/3 of the total number of blocks for each job from Name Node and pushes job to the corresponding RU_RM. RU_RM decides on resource lease requests from the application master in the rack [8]. With this model, compute node replicates each block on a different node but the same local rack. Compute node with the replicated block will replicate the third copy on a node in another rack. Each of the three computes nodes will communicate the NameNode once replication is over. The NameNode will then update the metadata server [8].

## 3. Description of wordcount operation using MapReduce

WordCount is the Hadoop benchmarking workload used for this work. WordCount is a typical two-phase Hadoop workload with the map task counting the frequency of individual words in a subset data file while the reduce task shuffles and gather the frequency of all the words. The input data for this work, therefore, is any text file. The input format is described in Table 1.

**Table 1**: Typical input data format for map and reduce task

| |
|---|
| Hi, how are you |
| How is your job |
| How is your sister |
| How is your brother |
| What is the time now |
| What is the strength of Hadoop |

To perform wordcount operation on the input data in Table 1, we assume that the file name is file.txt and the size is 140MB. If 64MB is the size of each block to be stored in HDFS, the text file will be partitioned into 2blocks of 64MB and a block will contain 8MB as shown in Table 2.

**Table 2:** Block partitions for input data

| | |
|---|---|
| Hi, how are you<br>How is your job | 64BM |
| How is your sister<br>How is your brother | 64MB |
| What is the time now<br>What is the strength of Hadoop | 8MB |

The number of input splits for a file depends on the number of blocks you have for the job. Since there are three blocks in Table 2, we have three input splits and there are three corresponding mappers and reducers (one input split to a mapper and a reducer). From the blocks in Table 2, block1 will be allocated to the first input split, block2 to the second, and block3 to the third input split. Hadoop runs MapReduce jobs in the form of (key, value) pair. For the text file to be read and converted into (key, value) pair, there is an interface called RecordReader. The RecordReader reads each line in the text and converts it into (key, value) pair with the format (byteoffset, entireline). The 'byteoffset' represents row number in the text while 'entireline' is the whole text in the line. For example, to read block1 in the file.txt, (byteoffset, entireline) will be (0, hi how are you). The RecordReader gets the next byteoffset by reading the number of characters in the first row. The first row has the text 'hi_how_are_you_ = 15characters including spaces. Hence, the next (byteoffset, entireline) = (16, how is your job).

The generated data from the mapper form another (key, value) pair which is referred to as intermediate data (output). Once these intermediate data are generated, the reducer function is triggered to combine all intermediate data into the final output. The shuffling phase combines all single keys to produce these intermediate data described in Table 3.

**Table 3:** Format of intermediate output from MapReduce task

| |
|---|
| Hi, [1] |
| how, [1,1,1,1] |
| are, [1] |
| you [1] |
| is [1,1,1,1,1] |
| your [1,1,1] |
| job [1] |
| sister [1] |
| brother [1] |
| the [1,1] |
| time [1] |
| now [1] |
| strength [1] |
| of [1] |
| Hadoop [1] |

The reducer has RecordWriter. Once the intermediate data has been shuffled, the reducer will sort it and pass it to RecordWriter which produces the output file into HDFS. Table 4 describes the output file format.

**Table 4:** Output file format

| |
|---|
| Hi, 1 |
| how, 4 |
| are, 1 |
| you, 1 |
| is, 5 |
| your, 3 |
| job, 1 |
| sister, 1 |
| brother, 1 |
| the, 2 |
| time, 1 |
| now, 1 |
| strength, 1 |
| of, 1 |
| Hadoop, 1 |

## 4. Performance evaluation of YARN and rack-aware system for YARN using wordcount

This section evaluates the YARN and the rack-aware model for Hadoop YARN by processing a typical Hadoop workload called WordCount. To fairly capture the timestamp of each task, the execution time for each block of a single task was recorded. This was done for both architectures. Two different file sizes were used for this experiment, with a finished time for each block of the file recorded. data was partitioned into 6kB.

Experiment 1: Figure 3 shows results of WordCount operation with text file of 30.5kB in size

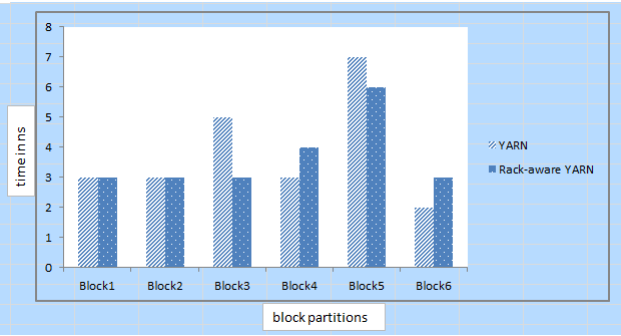


**Figure 3.** Block partitions result of wordcount operation for both architectures with file size = 30.5kB

Experiment 2: The second experiment shows a larger file size of 92kB. Figure 4 shows the results of the WordCount operation performed on this file.

*4.1 Discussion:* Two performance metrics were defined for this work; efficiency and average task-delay ratio.

$$\text{Efficiency} = (T_{actual} / T_{ideal}) * 100\% \quad (1)$$

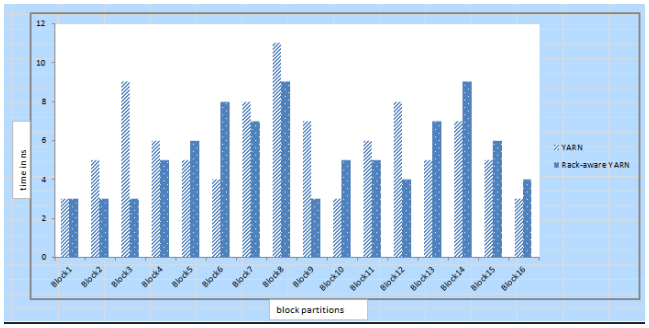


**Figure 4.** Block partitions result of wordcount operation for both models with file size = 92kB

To obtain $T_{actual}$ for this work, we run a task of file size = 6kB. Since this workload is contained in just a block partition, it was assumed that no scheduling overhead is needed. Hence, finish time (which is approximately 3ms = 3000000000ns) forms $T_{actual}$ for the work.

*Performance analysis for Experiment 1*

$T_{ideal}$ for YARN and rack-aware model are 23000000000ns and 22000000000ns respectively.

$T_{actual}$ for Exp.1 = 3000000000ns * number of blocks - (2)
$T_{actual}$ = 3000000000ns * 6blocks = 18000000000ns

Efficiency of YARN from (1)
= 18000000000ns/23000000000ns * 100%
= 0.783 * 100%
= 78.3%
Efficiency of rack-aware model from (1)
= 18000000000ns/22000000000ns * 100%
= 0.818 * 100%
= 81.8%

*Performance analysis for Experiment 2*

$T_{ideal}$ for YARN and rack-aware model are 95000000000ns and 87000000000ns respectively.

$T_{actual}$ for Exp.2 = 3000000000ns * number of blocks - (2)
$T_{actual}$ = 3000000000ns * 16blocks = 48000000000ns

Efficiency of YARN from (1)
= 48000000000ns/95000000000ns * 100%
= 0.505 * 100%
= 50.5%
Efficiency of rack-aware model from (1)
= 48000000000ns/87000000000ns * 100%
= 0.552 * 100%
= 55.2%

*Average Task-Delay Ratio* is represented by (3).

$$r_{td} = (T_{itf} - T_{atf}) / T_{atf} \quad (3)$$

Where.

$$T_{itf} = T_{ideal} / \text{number of blocks} \quad (4)$$

$$T_{atf} = T_{actual\ of\ a\ single\ block} \quad (5)$$

*Average Task-Delay Ratio performance analysis for Experiment 1*

From (3), YARN

$r_{td}$ = [(23000000000ns/6) – 3000000000ns] / 3000000000ns
= (3833333333.3ns – 3000000000ns) / 3000000000ns
= 833333333.3ms / 3000000000ns
= 0.278ns

From (3), Rack-aware Model

$r_{td}$ = [(22000000000ns/6) – 3000000000ns] / 3000000000ns
= (3666666666.7ns – 3000000000ns) / 3000000000ns
= 666666666.7ns / 3000000000ns
= 0.222ns

*Average Task-Delay Ratio performance analysis for Experiment 2*

From (3), YARN

$r_{td}$ = [(95000000000ns/16) – 3000000000ns] / 3000000000ns
= (5937500000ns – 3000000000ns) / 3000000000ns
= 2937500000ns / 3000000000ns
= 0.979ns

From (3), Rack-aware Model

$r_{td}$ = [(87000000000ns/16) – 3000000000ns] / 3000000000ns
= (543750000ns – 3000000000ns) / 3000000000ns
= 2437500000ns / 3000000000ns
= 0.813ns

## 5. Conclusion

The rack-aware model for Hadoop YARN is an improvement over the Hadoop YARN model. Results obtained from computation of efficiency and average task-delay ratio showed that as data gets bigger, the rack-aware model performed better than YARN. This is observed from the results obtained from the experiments. A difference of 3.5ns and 4.7ns for file sizes 30.5kB and 92kB respectively was obtained with the first metric (efficiency). Also, the difference in delay for file size 30.5kB and 92kB are 0.056ns and 0.166ns respectively. This shows that, with bigger data, delay time in YARN model will be higher than the delay time in rack-aware system for Hadoop YARN. Since Hadoop was developed for big data analytics, the rack-aware model will be a better solution for a scalable and efficient resource management framework.

## References

1. Shvachko, K., Kuang, H., Radia, S., and Chansler, R., The hadoop distributed file system. *2010 IEEE 26th symposium on Mass Storage Systems and Technologies (MSST)*, Washington D. C., 2010, USA: IEEE Computer Society.
2. Ghemawat, S., Gobioff, H., and Leung, S.-T., The Google file system. *Paper presented at the ACM SIGOPS operating systems review*, New York City, 2003, NY: ACM.
3. Shouvik, B., and Daniel, A. M., The anatomy of MapReduce jobs, scheduling and performance challenges. *Proceedings of the 2013 conference of the Computer Measurement Group*, San Diego, 2013, CA: Semantic Scholar
4. White, T.,. *Hadoop: The definitive guide*. Sebastopol, CA: O'Reilly Media, 2009.
5. Ibrahim, A. T. H., Nor, B. A., Abdullah, G., Ibrar, Y., Feng, X., and Samee, U. K., MapReduce: Review and Challenges. *Springer Journal*, 109(1), 389-421, 2016. http://www.doi.org/10.1145/1327452.1327492
6. Vinod, K. V., Arun, C. M., Chris, D., Sharad, A., Mahadev, K., Robert, E., Thomas, G., Jason, L., Hitesh, S., Siddaharth, S., Bikas, S., Carlo, C., Owen, O. M., Sanjay, R., Benjamin, R., and Eric, B., Apache Hadoop YARN: Yet Another Resource Negotiator. *SOCC '13 Proceedings of the 4th annual symposium on Cloud Computing*, New York, 2013, NY: ACM. http://dx.doi.org/10.1145/2523616.2523633
7. Moses, T., A proposed rack-aware model for high-availability of Hadoop Distributed File System (HDFS) architecture, *University of Pitesti Scientific Bulletin: Electronics and Computer Science*, Vol. 20, No. 1, 2020.
8. Moses, T., Inyiama, H. C., and Anigbogu, S. O., A rack-aware scalable resource management system for Hadoop YARN, *International Journal of High Performance Computing and Networking*, Vol. 16, No 1, 2020.
9. Apache, Apache Hadoop. Retrieved from https://hadoop.apache.org/. on 3rd March, 2017.

**Biographical notes**

**Timothy Moses** has received his Ph.D. from Nnamdi Azikiwe University, Awka – Nigeria. He is currently a Senior Lecturer at the Department of Computer Science, Federal University of Lafia – Nasarawa State – Nigeria.

**Tajudeen Adeleke Badmos** has received his Ph.D. from Sheffield Hallam University, United Kingdom. He is a researcher at the Industry and Innovation Research Institute of the university.

**Rukayyat Abdulkarim** has received her MSc in Computer Science from Ahmadu Bello University, Zaria – Nigeria. She is a Lecturer at the Department of Computer Science, Federal University of Lafia, Nasarawa State – Nigeria.