Advanced MR Concepts and Hadoop 2.0

Big Data Analysis

Revisiting the MR Framework

The *MapReduce* (MR) framework consists of two functions:

1. Map(k1, v1) → list(k2, v2)
2. Reduce(k2, list(v2)) → list(v3)

The *map* function receives a list of key and value pairs, and it outputs a list of key and value pairs. The *reduce* function receives a key and an iterator of values, and it outputs a list of values. Values with the same key go to the same reducer.

Revisiting Example #2 from Last Module

Input to the Map Function

(3424, 'the dog sat on the mat')
(3447, 'the parrot sat on the sofa')

Output from the Map Function

('the', 1), ('dog', 1), ('sat', 1), ('on', 1), ('the', 1), ('mat', 1)
('the', 1), ('parrot', 1), ('sat', 1), ('on', 1), ('the', 1), ('sofa', 1)

Data Sent to the Reducer

(Note: the keys are in sorted order.)

('dog', [1])

('mat', [1])

('on', [1, 1])

('parrot', [1])

('sat', [1, 1])

('sofa', [1])

('the', [1, 1, 1, 1])

Output from the Reducer

('dog', 1)

('mat', 1)

('on', 2)

('parrot', 1)

('sat', 2)

('sofa', 1)

('the', 4)

Advanced MR Concepts: Combiner

MapReduce jobs are restricted by the bandwidth available on the cluster, so it helps to reduce the amount of data transferred between map and rescue tasks. The *combiner* function serves as an optimizer, minimizing the data transferred between *map* and *reduce* tasks (see Figure 1).

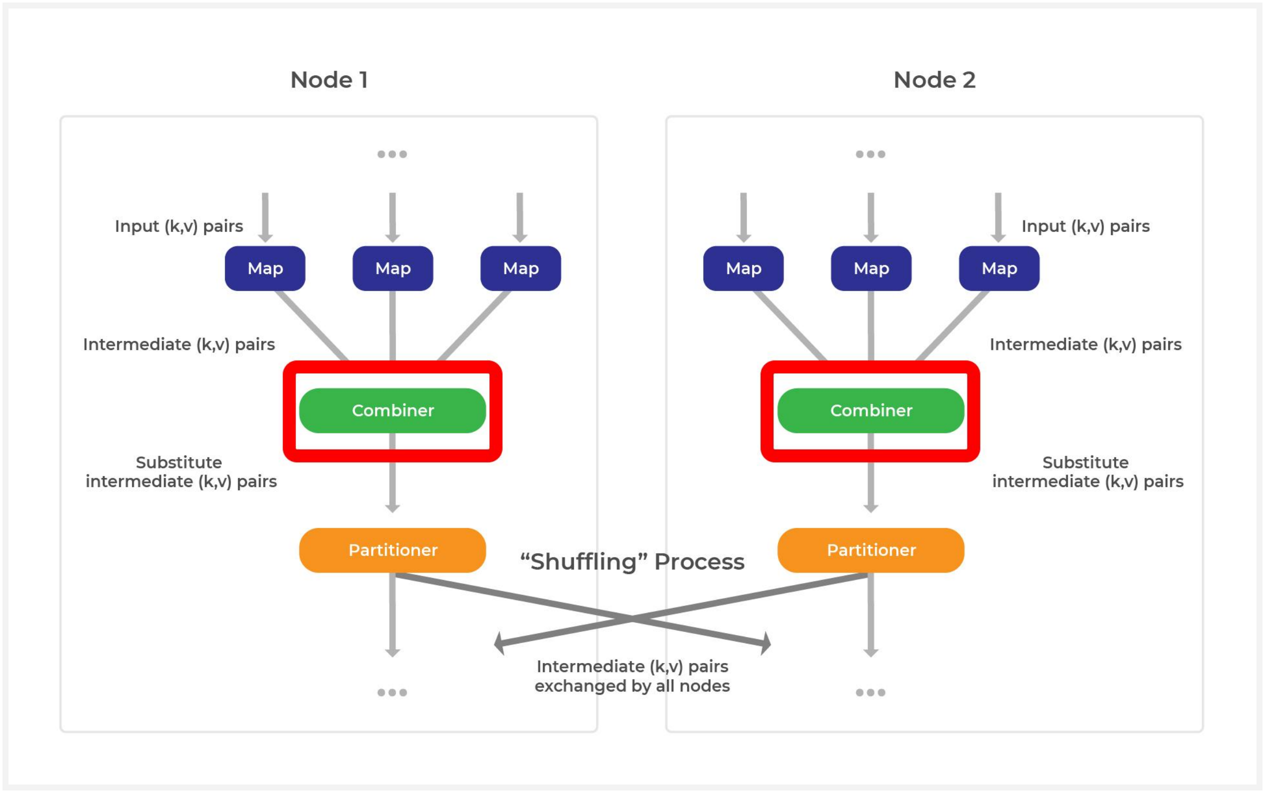

Figure 1: Combiner function of a MapReduce data flow

The Hadoop framework allows a user to define a combiner function to be executed on the output emitted by the map function. The combiner function's output forms the input to the reduce function.

Revisiting Example #2 (with the Combiner Construct)

Input to the Map Function

(3424, 'the dog sat on the mat')
(3447, 'the parrot sat on the sofa')

Output from the Map Function

('the', 1), ('dog', 1), ('sat', 1), ('on', 1), ('the', 1), ('mat', 1)
('the', 1), ('parrot', 1), ('sat', 1), ('on', 1), ('the', 1), ('sofa', 1)

Data Sent to the Reducer (Using a Combiner)

(Note: the keys are in sorted order.)

('dog', [1])

('mat', [1])

('on', [2])

('parrot', [1])

('sat', [2])

('sofa', [1])

('the', [4])

Output from the Reducer

('dog', 1)

('mat', 1)

('on', 2)

('parrot', 1)

('sat', 2)

('sofa', 1)

('the', 4)

It is important to note that the combiner function does not replace the reduce function. The reduce function is still necessary to process records with the same key from different map functions.

What is YARN?

Apache Hadoop *YARN* (which stands for *Yet Another Resource Negotiator*) is the evolution of Hadoop 1.0. Originally called *MapReduce 2*, YARN (or Hadoop 2.0) was developed to overcome the limitations of Hadoop 1.0. Let's look at the limitations of Hadoop 1.0 and then go over how YARN helps to solve these problems.

Limitations of Hadoop 1.0

- **Scalability**: The maximum cluster size is limited to 4000 nodes, and the maximum number of concurrent tasks cannot exceed 40,000 across the nodes in a cluster.

- **Single Point of Failure**: The NameNode or the JobTracker could become the "choking" point; failure abandons all queued and running jobs, and the jobs need to be resubmitted.
- **Restart is tricky**: If a failure is encountered, then restart is extremely tricky due to the complexities associated with managing state across the nodes within the cluster.

JobTracker and YARN

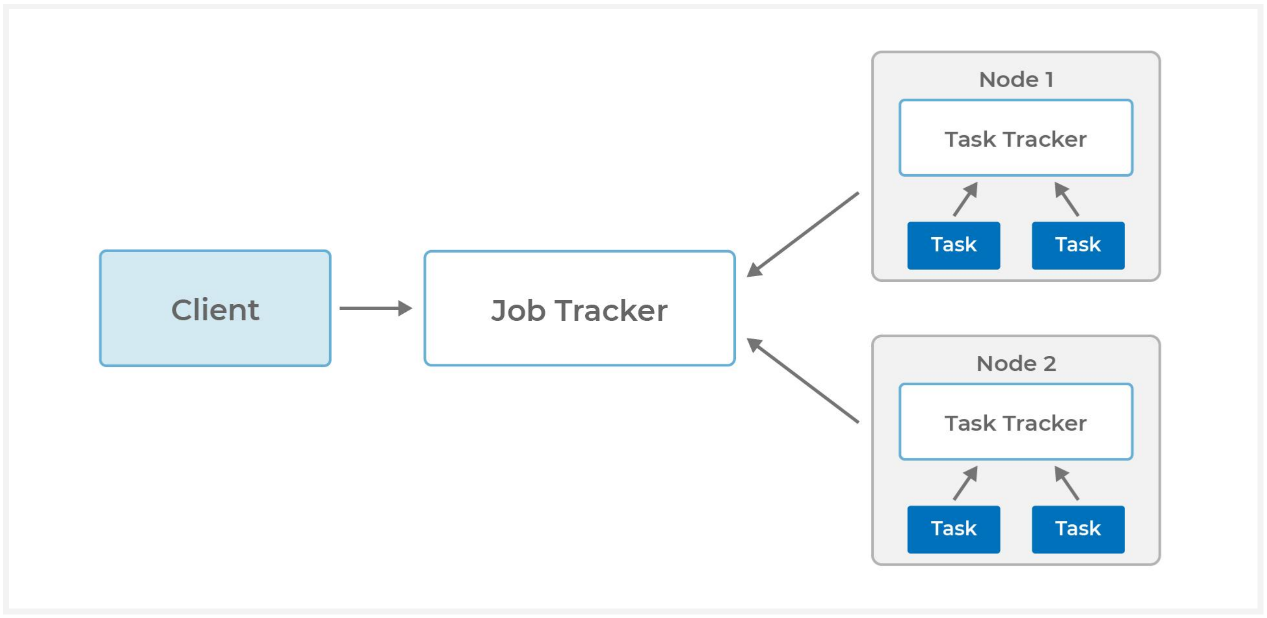

Figure 2: JobTracker

The two main functions of the JobTracker are *resource management* and *job scheduling/monitoring*. The JobTracker serves as the *single point of control* in the design, and as such, the workload processed by the JobTracker runs into problems due to competing demand for resources and for job execution cycles. The fundamental purpose of YARN (i.e., Hadoop 2) is to split up the two main functions of the JobTracker into separate processes, thereby segregating the resource management function from the job scheduling/monitoring function (see Figure 2).

The YARN Process

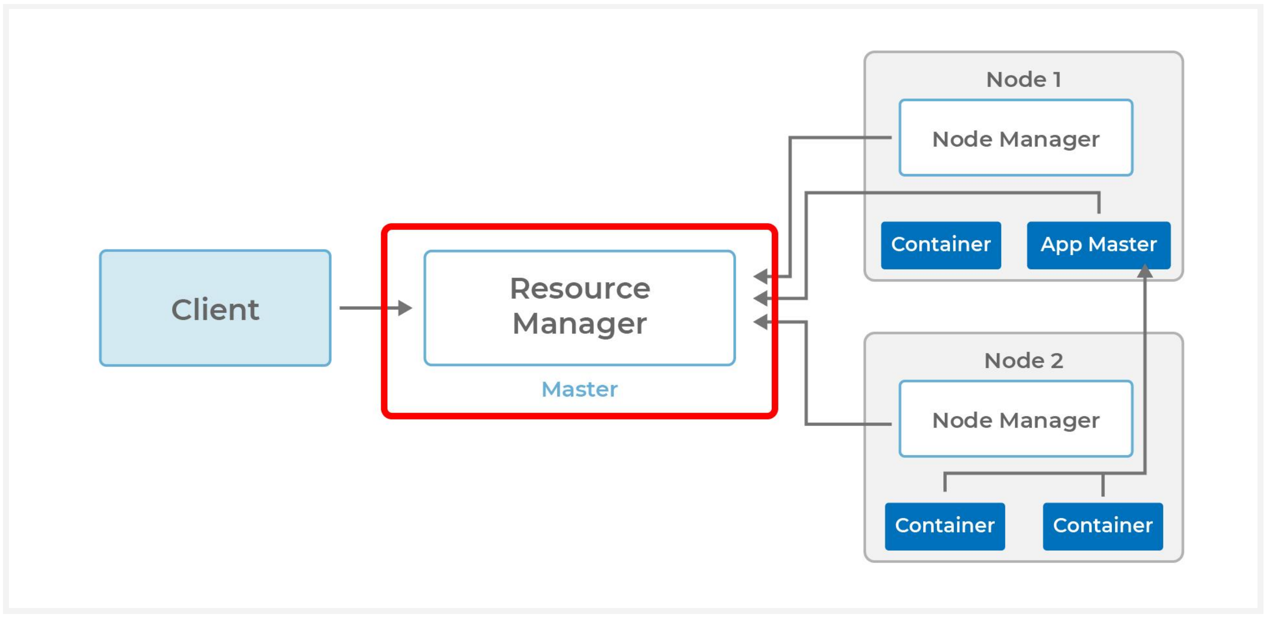

Figure 3: The Yarn process: Resource manager

In the new Hadoop 2.0 architecture, there are two main modules: a global *ResourceManager* and a per application *ApplicationMaster* (or AppMaster). The ResourceManager runs on the master node and serves as an arbitrator that allocates available resources in the cluster amongst competing applications (see Figure 3).

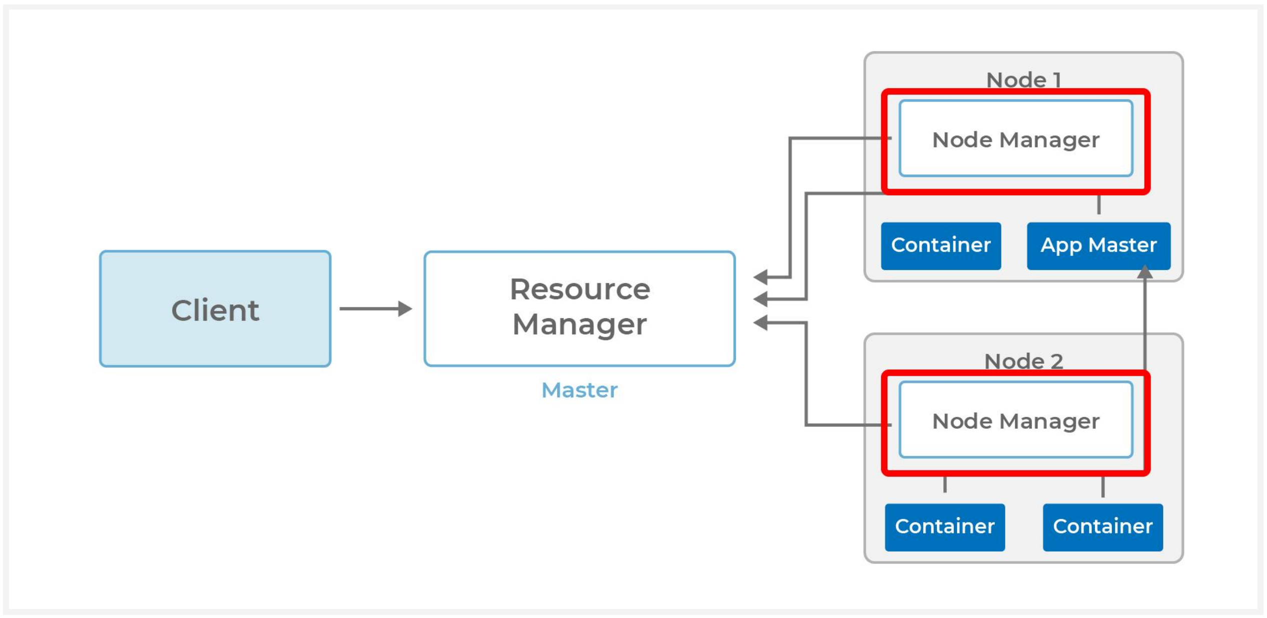

Figure 4: The Yarn process: NodeManager

A NodeManager runs on each slave node in the cluster and takes directions from the ResourceManager (see Figure 4). It is accountable for managing resources on a single node in the cluster.

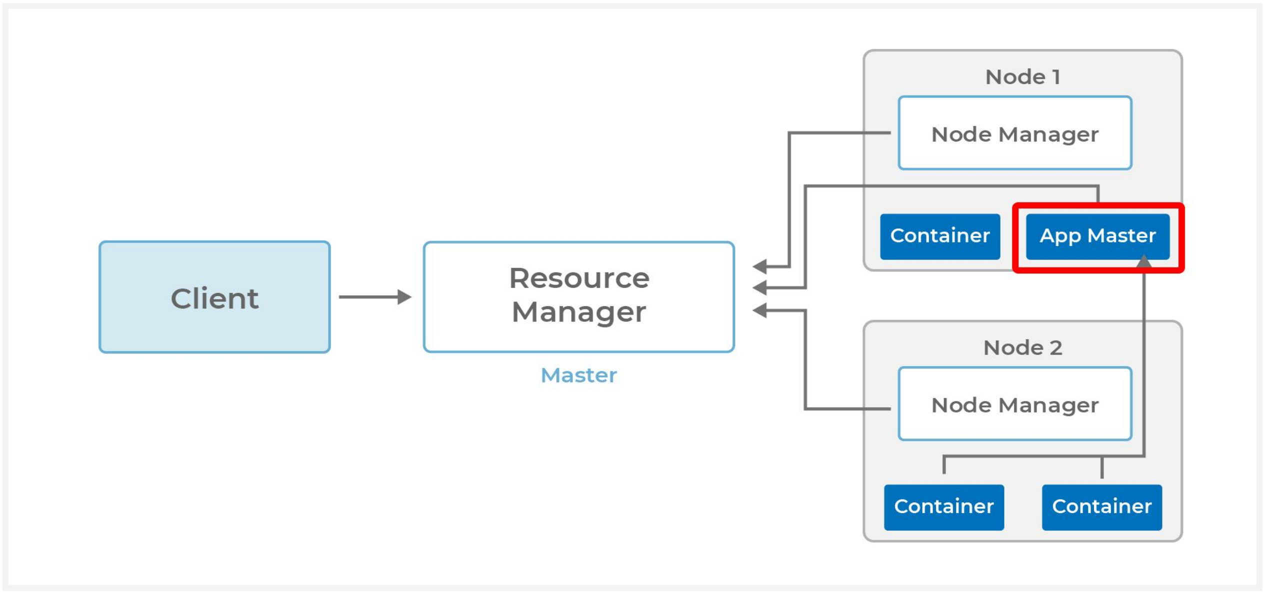

Figure 5: The Yarn process: ApplicationMaster

An AppMaster executes a specific YARN job and is accountable for negotiating resources with the ResourceManager. The AppMaster also works in close coordination with the NodeManager to monitor the *Container* (see Figure 5). There is one AppMaster per application.

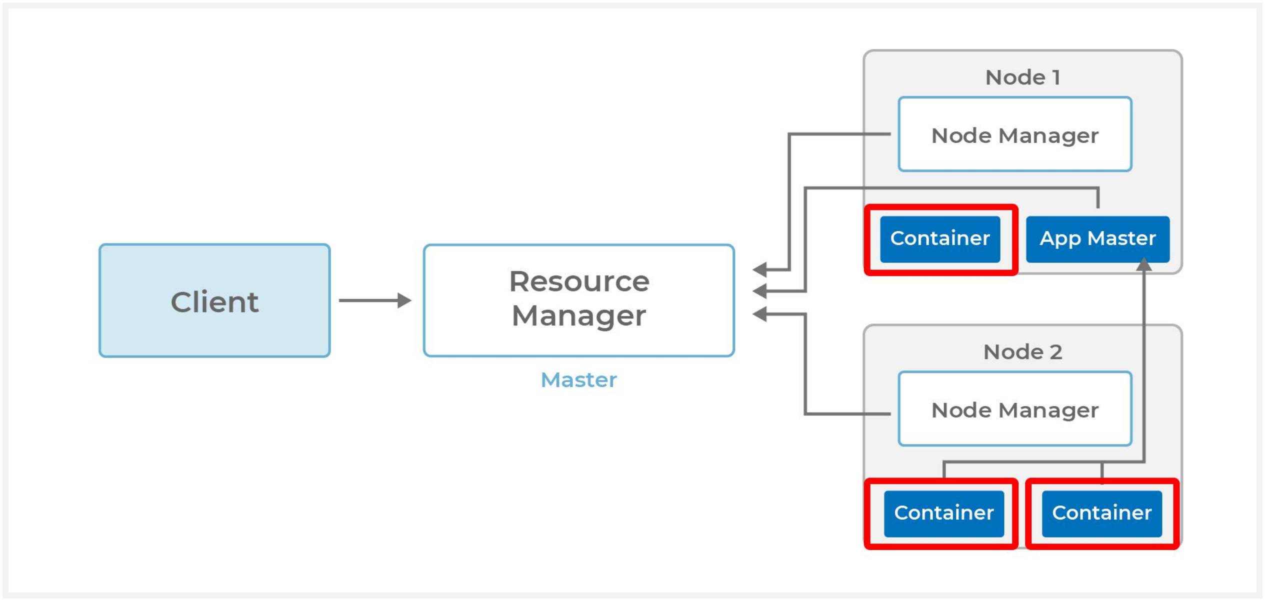

Figure 6: The Yarn process: Containers

The Container is created by the ResourceManager. On request, a container allocates a certain number of resources on a slave node. Applications execute within one or more containers (see Figure 6).

Running an Application in a YARN Cluster

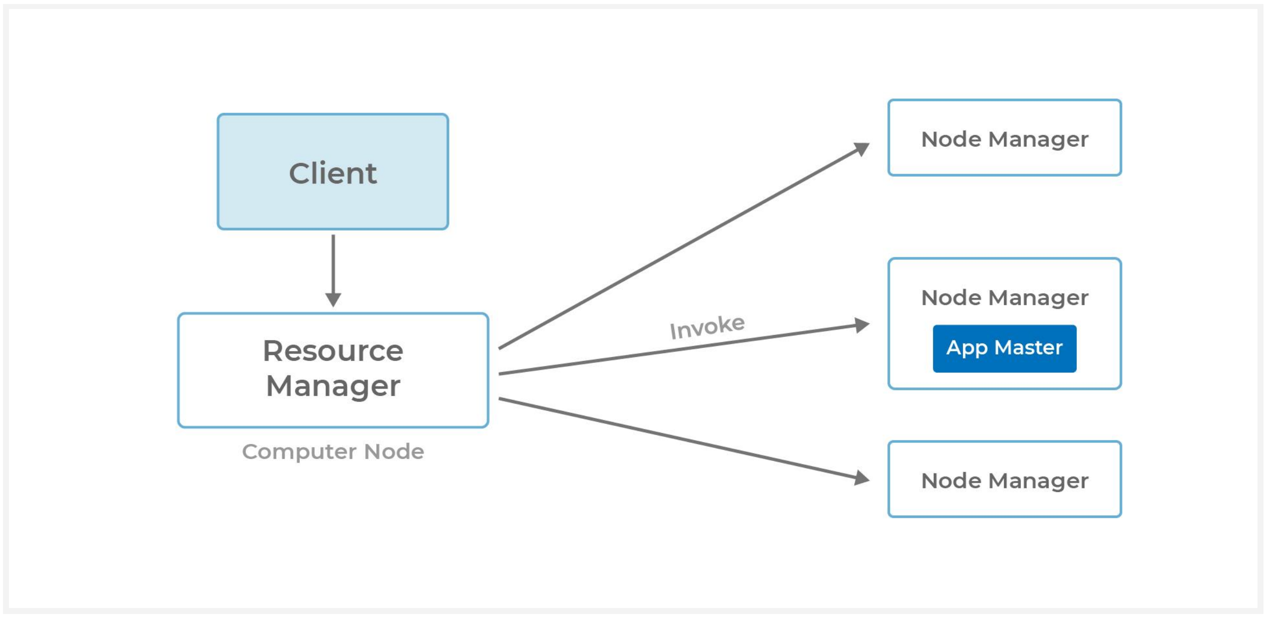

Figure 7: A YARN cluster

The ResourceManager communicates with a NodeManager to invoke an AppMaster (see Figure 7). The AppMaster executes in a container. If more containers are required to run the application, the AppMaster negotiates for more resources from the ResourceManager.

In this case, two additional containers are launched to service the application as depicted below (Figure 8). For the purposes of this example, the application that is being serviced is called *APP*.

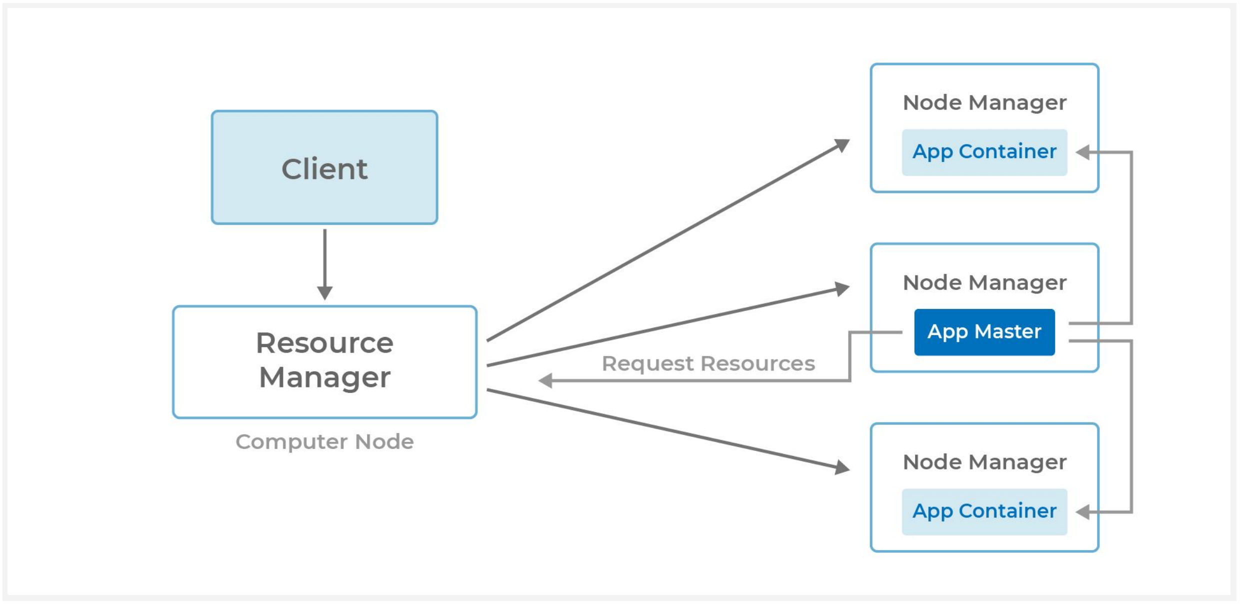

Figure 8: Containers service the application

The AppMaster can launch additional application tasks related to the application "APP" in the containers allocated by the ResourceManager. The application code running within the Container provides requisite information such as status and progress to the AppMaster. During the execution of the application, the client that submitted the program communicates directly with the AppMaster to source progress and status updates regarding the execution of the application code. Once the application execution is completed, the AppMaster de-registers with the ResourceManager and shuts down, which in turn causes the containers to be repurposed.

With YARN, Hadoop has been reoriented into a more powerful data platform wherein Hadoop is no longer restricted to supporting just batch processing. This allows Hadoop to be utilized in the context of interactive and streaming workloads where Hadoop serves as the file system (data repository) and YARN acts as the operating system, orchestrating resources across a cluster.

References

Cloudera. (2019). Developing Apache Spark Applications.

Harvey, C. (2017). Big Data Use Cases.

Hurwitz, J., Nugent, A. Halper, F. & Kaufman, M. (2013). Chapter 12: Defining Big Data Analytics. *Big Data for Dummies.*

Karau, H., Konwinski, A., Wendell P. & Zaharia, M. (2015). Chapter 1. Introduction to Data Analysis with Spark. *Learning Spark.*