SNS COLLEGE OF TECHNOLOGY
COIMBATORE-35

DEPARTMENT OF INFORMATION TECHNOLOGY

19ITE305 – BIG DATA ANALYTICS

UNIT II: INTRODUCTION TO TECHNOLOGY LANDSCAPE

Topic 6 and 7: Processing Data with Hadoop - Managing Resources and Applications with Hadoop YARN

Building Blocks of Hadoop

1. HDFS (The storage layer)

As the name suggests, Hadoop Distributed File System is the storage layer of Hadoop and is responsible for storing the data in a distributed environment (master and slave configuration). It splits the data into several blocks of data and stores them across different data nodes. These data blocks are also replicated across different data nodes to prevent loss of data when one of the nodes goes down.

It has two main processes running for processing of the data: –

a. NameNode

It is running on the master machine. It saves the locations of all the files stored in the file system and tracks where the data resides across the cluster i.e. it stores the metadata of the files. When the client applications want to make certain operations on the data, it interacts with the NameNode. When the NameNode receives the request, it responds by returning a list of Data Node servers where the required data resides.

b. DataNode

This process runs on every slave machine. One of its functionalities is to store each HDFS data block in a separate file in its local file system. In other words, it contains the actual data in form of blocks. It sends heartbeat signals periodically and waits for the request from the NameNode to access the data.

2. MapReduce (The processing layer)

It is a programming technique based on Java that is used on top of the Hadoop framework for faster processing of huge quantities of data. It processes this huge data in a distributed environment using many Data Nodes which enables parallel processing and faster execution of operations in a fault-tolerant way.

A MapReduce job splits the data set into multiple chunks of data which are further converted into key-value pairs in order to be processed by the mappers. The raw format of the data may not be suitable for processing. Thus, the input data compatible with the map phase is generated using the InputSplit function and RecordReader.

InputSplit is the logical representation of the data which is to be processed by an individual mapper. RecordReader converts these splits into records which take the form of key-value pairs. It basically converts the byte-oriented representation of the input into a record-oriented representation.

These records are then fed to the mappers for further processing the data. MapReduce jobs primarily consist of three phases namely the Map phase, the Shuffle phase, and the Reduce phase.

a. Map Phase

It is the first phase in the processing of the data. The main task in the map phase is to process each input from the RecordReader and convert it into intermediate tuples (key-value pairs). This intermediate output is stored in the local disk by the mappers.

The values of these key-value pairs can differ from the ones received as input from the RecordReader. The map phase can also contain combiners which are also called as local reducers. They perform aggregations on the data but only within the scope of one mapper.

As the computations are performed across different data nodes, it is essential that all the values associated with the same key are merged together into one reducer. This task is performed by the partitioner. It performs a hash function over these key-value pairs to merge them together.

It also ensures that all the tasks are partitioned evenly to the reducers. Partitioners generally come into the picture when we are working with more than one reducer.

b. Shuffle and Sort Phase

This phase transfers the intermediate output obtained from the mappers to the reducers. This process is called as shuffling. The output from the mappers is also sorted before transferring it to the reducers. The sorting is done on the basis of the keys in the key-value pairs. It helps the reducers to perform the computations on the data even before the entire data is received and eventually helps in reducing the time required for computations.

As the keys are sorted, whenever the reducer gets a different key as the input it starts to perform the reduce tasks on the previously received data.

c. Reduce Phase

The output of the map phase serves as an input to the reduce phase. It takes these key-value pairs and applies the reduce function on them to produce the desired result. The keys and the values associated with the key are passed on to the reduce function to perform certain operations.

We can filter the data or combine it to obtain the aggregated output. Post the execution of the reduce function, it can create zero or more key-value pairs. This result is written back in the Hadoop Distributed File System.

3. YARN (The management layer)

Yet Another Resource Navigator is the resource managing component of Hadoop. There are background processes running at each node (Node Manager on the slave machines and Resource Manager on the master node) that communicate with each other for the allocation of resources. The Resource Manager is the centrepiece of the YARN layer which manages resources among all the applications and passes on the requests to the Node Manager.

The Node Manager monitors the resource utilization like memory, CPU, and disk of the machine and conveys the same to the Resource Manager. It is installed on every Data Node and is responsible for executing the tasks on the Data Nodes.

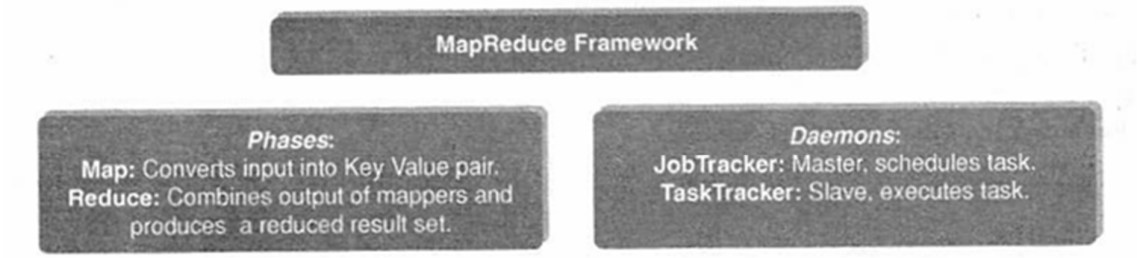


Figure 5.21 MapReduce Programming phases and daemons.

Map Reduce daemons

1. **JobTracker:** It provides connectivity between Hadoop and your application. When you submit code to cluster, JobTracker creates the execution plan by deciding which task to assign to which node. It also monitors all the running tasks. When a task fails, it automatically re-schedules the task to a different node after a predefined number of retries. JobTracker is a master daemon responsible for executing overall MapReduce job. There is a single JobTracker per Hadoop cluster.
2. **TaskTracker:** This daemon is responsible for executing individual tasks that is assigned by the JobTracker. There is a single TaskTracker per slave and spawns multiple Java Virtual Machines (JVMs) to handle multiple map or reduce tasks in parallel. TaskTracker continuously sends heartbeat message to JobTracker. When the JobTracker fails to receive a heartbeat from a TaskTracker, the JobTracker assumes that the TaskTracker has failed and resubmits the task to another available node in the cluster. Once the client submits a job to the JobTracker, it partitions and assigns diverse MapReduce tasks for each TaskTracker in the cluster. Figure 5.22 depicts JobTracker and TaskTracker interaction.

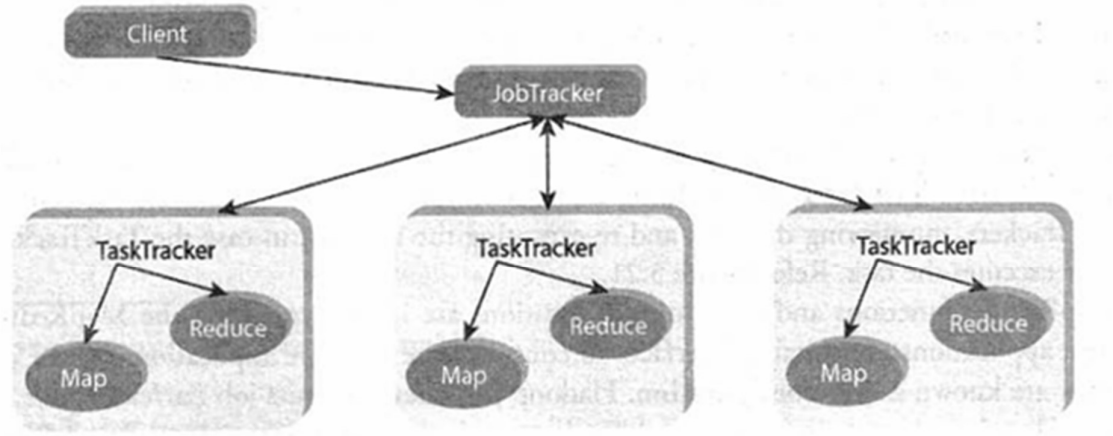


Figure 5.22 JobTracker and TaskTracker interaction.

How does map reduce works?

MapReduce divides a data analysis task into two parts – **map** and **reduce**. Figure 5.23 depicts how the MapReduce Programming works. In this example, there are two mappers and one reducer. Each mapper

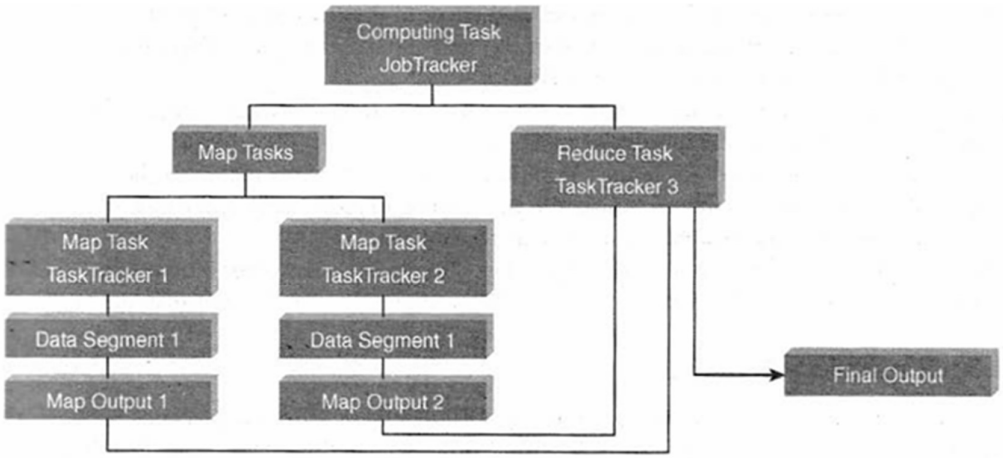


Figure 5.23 MapReduce programming workflow.

works on the partial dataset that is stored on that node and the reducer combines the output from the mappers to produce the reduced result set.

Reference: Wrox Big Data Certification Materials.

Figure 5.24 describes the working model of MapReduce Programming. The following steps describe how MapReduce performs its task.

1. First, the input dataset is split into multiple pieces of data (several small subsets).
2. Next, the framework creates a master and several workers processes and executes the worker processes remotely.

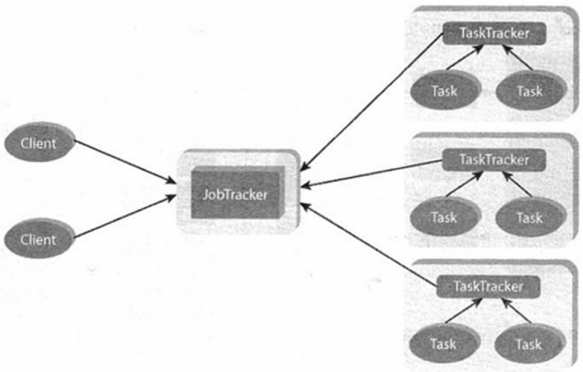


Figure 5.24 MapReduce programming architecture.

3. Several map tasks work simultaneously and read pieces of data that were assigned to each map task. The map worker uses the map function to extract only those data that are present on their server and generates key/value pair for the extracted data.
4. Map worker uses partitioner function to divide the data into regions. Partitioner decides which reducer should get the output of the specified mapper.
5. When the map workers complete their work, the master instructs the reduce workers to begin their work. The reduce workers in turn contact the map workers to get the key/value data for their partition. The data thus received is shuffled and sorted as per keys.
6. Then it calls reduce function for every unique key. This function writes the output to the file.
7. When all the reduce workers complete their work, the master transfers the control to the user program.

Map reduce example

The famous example for MapReduce Programming is **Word Count.** For example, consider you need to count the occurrences of similar words across 50 files. You can achieve this using MapReduce Programming. Refer Figure 5.25.

Word Count MapReduce Programming using Java

The MapReduce Programming requires three things.

1. **Driver Class:** This class specifies **Job Configuration** details.
2. **Mapper Class:** This class overrides the **Map Function** based on the problem statement.
3. **Reducer Class:** This class overrides the **Reduce Function** based on the problem statement.

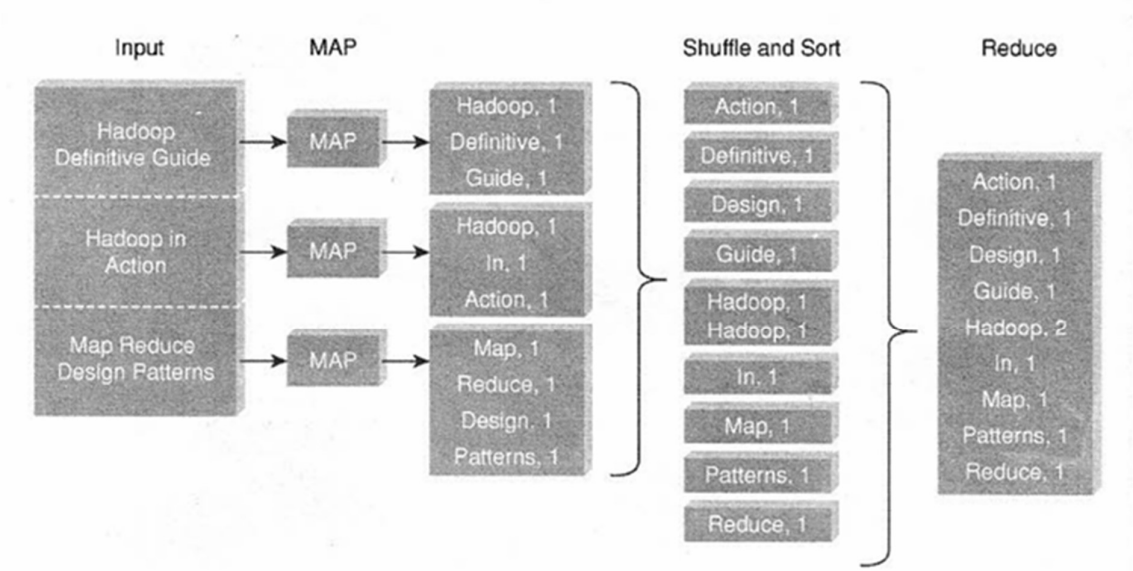
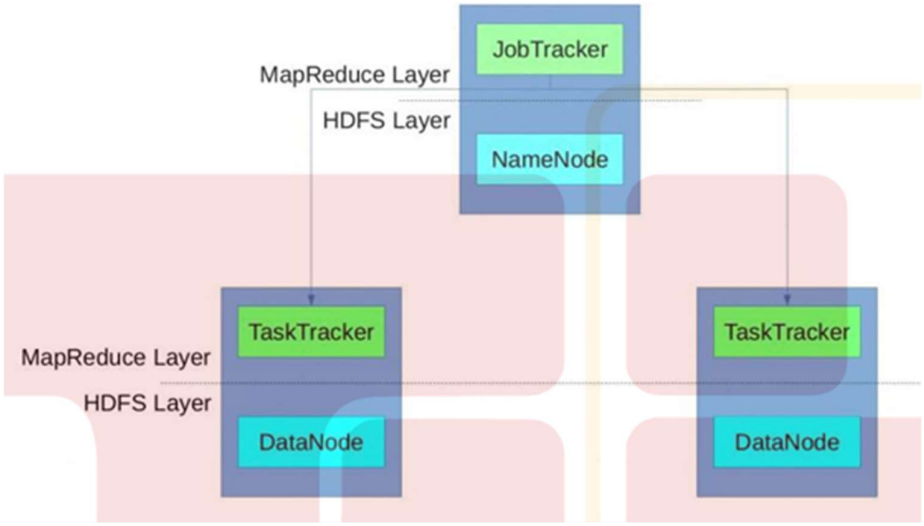


Figure 5.25 Wordcount example.

Managing Resources and Applications with Hadoop YARN

YARN (Yet Another Resource Negotiator) has been introduced to Hadoop with version 2.0 and solves a few issues with the resources scheduling of MapReduce in version 1.0. In order to understand the benefits of YARN, we have to review how resource scheduling worked in version 1.0.



A MapReduce job is split by the framework into tasks (Map tasks, Reducer tasks) and each task is run on of the DataNode machines on the cluster. For the execution of tasks, each DataNode machine provided a predefined number of slots (map slots, reducers slots). The JobTracker was responsible for the reservation of execution slots for the different tasks of a job and monitored their execution. If the execution failed, it reserved another slot and re-started the task. It also cleaned up temporary resources and make the reserved slot available to other tasks. The fact that there was only one JobTracker instance in Hadoop 1.0 led to the problem that the whole MapReduce execution could fail, if the the JobTracker fails (single point of failure). Beyond that, having only one instance of the JobTracker limits scalability (for very large clusters with thousands of nodes). The concept of predefined map and reduce slots also caused resource problems in case all map slots are used while reduce slots are still available and vice versa. In general it was not possible to reuse the MapReduce infrastructure for other types of computation like real-time jobs. While MapReduce is a batch framework, applications that want to process large data sets stored in HDFS and immediately inform the user about results cannot be implemented with it. Beneath the fact that MapReduce 1.0 did not offer realtime provision of computation results, all other types of applications that want to perform computations on the HDFS data had to be implemented as Map and Reduce jobs, which was not always possible. Hence Hadoop 2.0 introduced YARN as resource manager, which no longer uses slots to manage resources. Instead nodes have "resources" (like memory and CPU cores) which can be allocated by applications on a per request basis. This way MapReduce jobs can run together with non-MapReduce jobs in the same cluster. The heart of YARN is the Resource Manager (RM) which runs on the master node and acts as a global resource scheduler. It also arbitrates resources between competing applications. In contrast to the Resource Manager, the Node Managers (NM) run on slave nodes and communicates with the RM. The NodeManager is responsible for creating containers in which the applications run, monitors their CPU and memory usage and reports them to the RM. Each application has its own ApplicationMaster (AM) which runs within a container and negotiates resources with the RM and works with the NM to execute and monitor tasks. The MapReduce implementation of Hadoop 2.0 therefore ships with an AM (named MRAppMaster) that requests containers for the execution of the map tasks from the RM, receives the container IDs from the RM and then executes the map tasks within the provided containers. Once the map tasks have finished, it requests new containers for the execution of the reduce tasks and starts their execution on the provided containers. If the execution of a task fails, it is restarted by the ApplicationMaster. Should the ApplicationMaster fail, the RM will attempt to the restart the whole application (up

to two times per default). Therefore the ApplicationMaster can signal if it supports job recovery. In this case the ApplicationMaster receives the previous state from the RM and can only restart incomplete tasks. If a NodeManager fails, i.e the RM does not receive any heartbeats from it, it is removed from the list of active nodes and all its tasks are treated as failed. In contrast toversion 1.0 of Hadoop, the ResourceManager can be configured for High Availability.