Components of Hadoop 2.x

Hadoop is an Apache open source framework written in java that allows distributed processing of large datasets across clusters of computers using simple programming models. A Hadoop frame-worked application works in an environment that provides distributed storage and computation across clusters of computers. Hadoop is designed to scale up from single server to thousands of machines, each offering local computation and storage.

**Hadoop 2.x components**

Hadoop framework includes following four modules:

* **Hadoop Common:** These are Java libraries and utilities required by other Hadoop modules. These libraries provide file system and OS level abstractions and contains the necessary Java files and scripts required to start Hadoop.
* **Hadoop YARN:** This is a framework for job scheduling and cluster resource management.
* **Hadoop Distributed File System (HDFS™):** A distributed file system that provides high-throughput access to application data.
* **Hadoop MapReduce:** This is YARN-based system for parallel processing of large data sets.

In Hadoop 2x YARN is introduced. With YARN, Apache Hadoop is recast as a significantly more powerful platform – one that takes Hadoop beyond merely batch applications to taking its position as a ‘data operating system’ where HDFS is the file system and YARN is the operating system.

**Hadoop Yarn**:-

YARN is a re-architecture of Hadoop that allows multiple applications to run on the same platform. With YARN, applications run “in” Hadoop, instead of “on” Hadoop

The fundamental idea of YARN is to split up the two major responsibilities of the JobTracker and TaskTracker into separate entities. In Hadoop 2.0, the JobTracker and TaskTracker no longer exist and have been replaced by three components:

* **ResourceManager:** a scheduler that allocates available resources in the cluster amongst the competing applications.
* **NodeManager:** runs on each node in the cluster and takes direction from the ResourceManager. It is responsible for managing resources available on a single node.
* **ApplicationMaster:** an instance of a framework-specific library, an ApplicationMaster runs a specific YARN job and is responsible for negotiating resources from the ResourceManager and also working with the NodeManager to execute and monitor Containers.

The actual data processing occurs within the Containers executed by the ApplicationMaster. A Container grants rights to an application to use a specific amount of resources (memory, cpu etc.) on a specific host.

YARN is not the only new major feature of Hadoop 2.0. HDFS has undergone a major transformation with a collection of new features that include:

* **NameNode HA:** automated failover with a hot standby and resiliency for the NameNode master service.
* **Snapshots:** point-in-time recovery for backup, disaster recovery and protection against use errors.
* **Federation:** a clear separation of namespace and storage by enabling generic block storage layer.

NameNode HA is achieved using existing components like ZooKeeper along with new components like a quorum of JournalNodes and the ZooKeeper Failover Controller (ZKFC) processes.

Federation enables support for multiple namespaces in the cluster to improve scalability and isolation. Federation also opens up the architecture, expanding the applicability of HDFS cluster to new implementations and use cases.

The ResourceManager has two main components: Scheduler and ApplicationsManager.

The Scheduler is responsible for allocating resources to the various running applications subject to familiar constraints of capacities, queues etc. The Scheduler is pure scheduler in the sense that it performs no monitoring or tracking of status for the application. Also, it offers no guarantees about restarting failed tasks either due to application failure or hardware failures. The Scheduler performs its scheduling function based on the resource requirements of the applications; it does so based on the abstract notion of a resource *Container* which incorporates elements such as memory, cpu, disk, network etc.

The Scheduler has a pluggable policy which is responsible for partitioning the cluster resources among the various queues, applications etc. The current schedulers such as the Capacity Scheduler and the FairScheduler would be some examples of plug-ins.

The ApplicationsManager is responsible for accepting job-submissions, negotiating the first container for executing the application specific ApplicationMaster and provides the service for restarting the ApplicationMaster container on failure. The per-application ApplicationMaster has the responsibility of negotiating appropriate resource containers from the Scheduler, tracking their status and monitoring for progress.

MapReduce in hadoop-2.x maintains **API compatibility** with previous stable release (hadoop-1.x). This means that all MapReduce jobs should still run unchanged on top of YARN with just a recompile.

YARN also supports the notion of **resource reservation** via the Reservation System, a component that allows users to specify a profile of resources over-time and temporal constraints (e.g., deadlines), and reserve resources to ensure the predictable execution of important jobs.The *ReservationSystem* tracks resources over-time, performs admission control for reservations, and dynamically instruct the underlying scheduler to ensure that the reservation is fullfilled.

## MapReduce

Hadoop **MapReduce** is a software framework for easily writing applications which process big amounts of data in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.

The term MapReduce actually refers to the following two different tasks that Hadoop programs perform:

* **The Map Task:** This is the first task, which takes input data and converts it into a set of data, where individual elements are broken down into tuples (key/value pairs).
* **The Reduce Task:** This task takes the output from a map task as input and combines those data tuples into a smaller set of tuples. The reduce task is always performed after the map task.

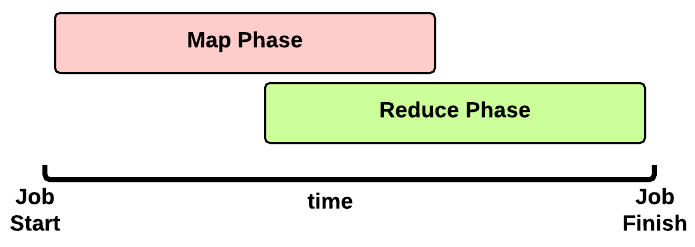
Typically both the input and the output are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.

The MapReduce framework consists of a single master **JobTracker** and one slave **TaskTracker** per cluster-node. The master is responsible for resource management, tracking resource consumption/availability and scheduling the jobs component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves TaskTracker execute the tasks as directed by the master and provide task-status information to the master periodically.

The JobTracker is a single point of failure for the Hadoop MapReduce service which means if JobTracker goes down, all running jobs are halted.

In MapReduce, a YARN application is called a **Job**. The implementation of the Application Master provided by the MapReduce framework is called **MRAppMaster**.

### [Timeline of a MapReduce Job](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html#timeline-of-a-mapreduce-job)

This is the timeline of a MapReduce Job execution:

* **Map Phase**: several **Map Tasks** are executed
* **Reduce Phase**: several **Reduce Tasks** are executed

Notice that the Reduce Phase may start before the end of Map Phase. Hence, an interleaving between them is possible.

### [Map Phase](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html#map-phase)

A key decision is how many MapTasks the Application Master needs to start for the current job.

#### [What does the user give us?](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html#what-does-the-user-give-us)

Let’s take a step back. When a client submits an application, several kinds of information are provided to the YARN infrastucture. In particular:

* a configuration: this may be partial (some parameters are not specified by the user) and in this case the default values are used for the job. Notice that these default values may be the ones chosen by a Hadoop provider like Amanzon.
* a JAR containing:
  + a map() implementation
  + a combiner implementation
  + a reduce() implementation
* input and output information:
  + input directory: is the input directory on HDFS? On S3? **How many files?**
  + output directory: where will we store the output? On HDFS? On S3?

The number of files inside the input directory is used for deciding the number of Map Tasks of a job.

#### [How many Map Tasks?](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html#how-many-map-tasks)

The Application Master will launch one MapTask for each map split. Typically, there is a map split for each input file. If the input file is too big (bigger than the HDFS block size) then we have two or more map splits associated to the same input file. This is the pseudocode used inside the method getSplits() of the FileInputFormat class:

num\_splits = 0

for each input file f:

remaining = f.length

while remaining / split\_size > split\_slope:

num\_splits += 1

remaining -= split\_size

where:

split\_slope = 1.1

split\_size =~ dfs.blocksize

Notice that the configuration parameter mapreduce.job.maps is ignored in MRv2 (in the past it was just an hint).

#### [MapTask Launch](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html#maptask-launch)

The MapReduce Application Master asks to the Resource Manager for Containers needed by the Job: one MapTask container request for each MapTask (map split).

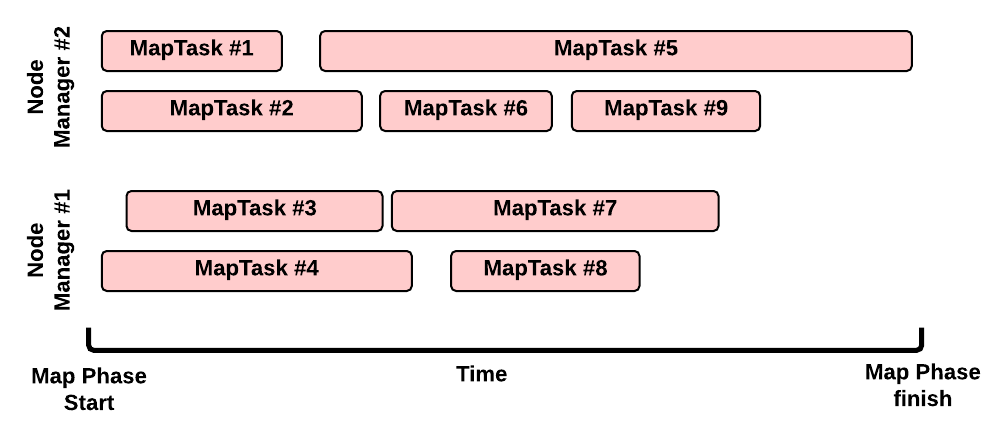
A container request for a MapTask tries to exploit data locality of the map split. The Application Master asks for:

* a container located on the same Node Manager where the map split is stored (a map split may be stored on multiple nodes due to the HDFS replication factor);
* otherwise, a container located on a Node Manager in the same rack where the the map split is stored;
* otherwise, a container on any other Node Manager of the cluster

This is just an hint to the Resource Scheduler. The Resource Scheduler is free to ignore data locality if the suggested assignment is in conflict with the Resouce Scheduler’s goal.

When a Container is assigned to the Application Master, the MapTask is launched.

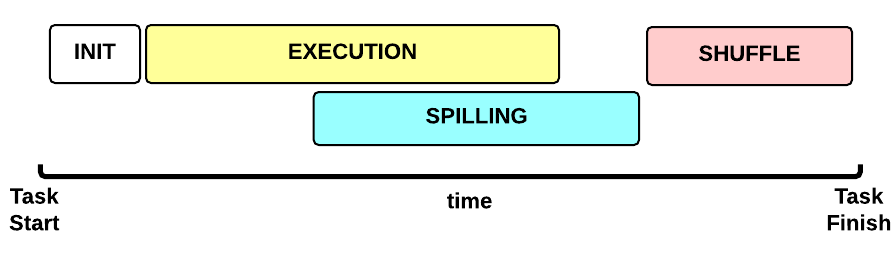
#### [Map Phase: example of an execution scenario](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html#map-phase-example-of-an-execution-scenario)



This is a possible execution scenario of the Map Phase:

* there are two Node Managers: each Node Manager has 2GB of RAM (NM capacity) and each MapTask requires 1GB, we can run in parallel 2 containers on each Node Manager (this is the best scenario, the Resource Scheduler may decide differently)
* there are no other YARN applications running in the cluster
* our job has 8 map splits (e.g., there are 7 files inside the input directory, but only one of them is bigger than the HDFS block size so we split it into 2 map splits): we need to run 8 Map Tasks.

#### [Map Task Execution Timeline](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html#map-task-execution-timeline)

Let’s now focus on a single Map Task. This is the Map Task execution timeline:

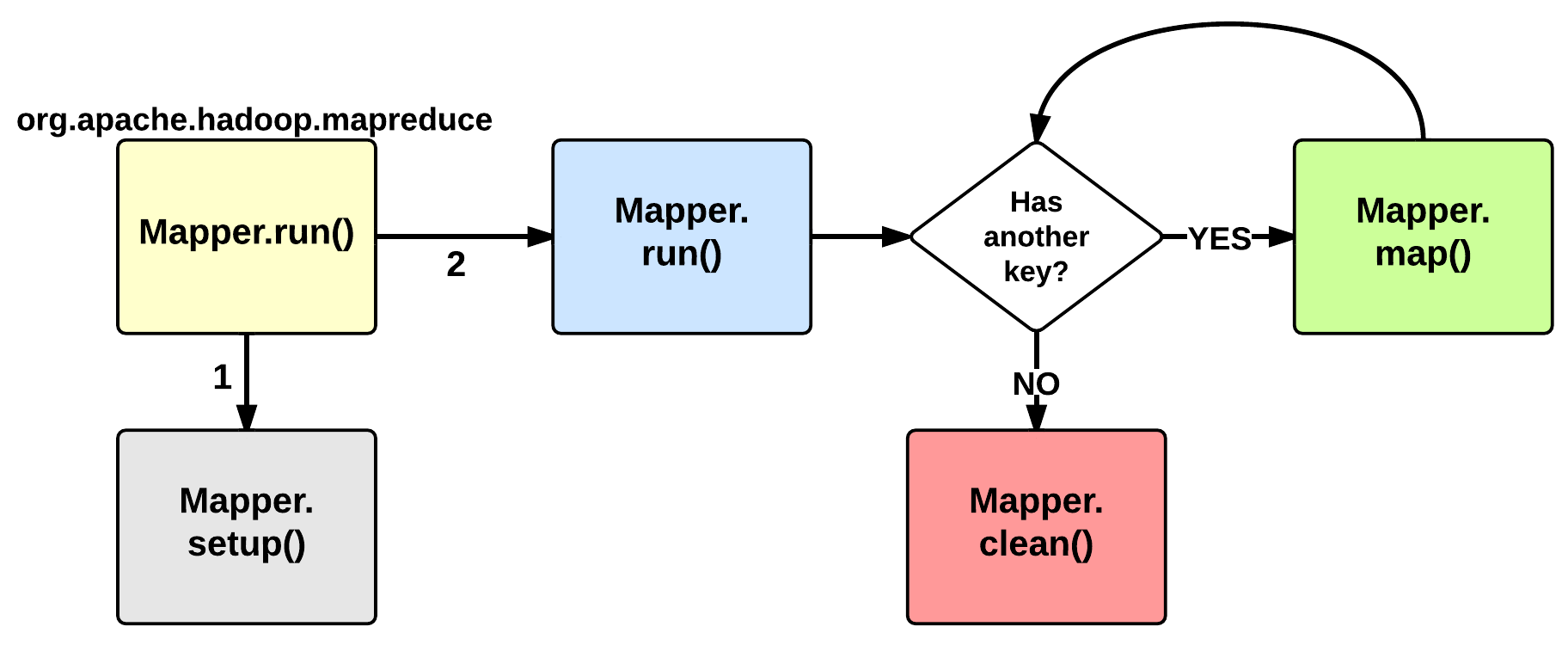
* **INIT** phase: we setup the Map Task
* **EXECUTION** phase: for each (key, value) tuple inside the map split we run the map() function
* **SPILLING** phase: the map output is stored in an in-memory buffer; when this buffer is almost full then we start (in parallel) the spilling phase in order to remove data from it
* **SHUFFLE** phase: at the end of the spilling phase, we merge all the map outputs and package them for the reduce phase

#### [MapTask: INIT](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html#maptask-init)

During the INIT phase, we:

1. create a context (TaskAttemptContext.class)
2. create an instance of the user Mapper.class
3. setup the input (e.g., InputFormat.class, InputSplit.class, RecordReader.class)
4. setup the output (NewOutputCollector.class)
5. create a mapper context (MapContext.class, Mapper.Context.class)
6. initialize the input, e.g.:
7. create a SplitLineReader.class object
8. create a HdfsDataInputStream.class object

#### [MapTask: EXECUTION](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html#maptask-execution)

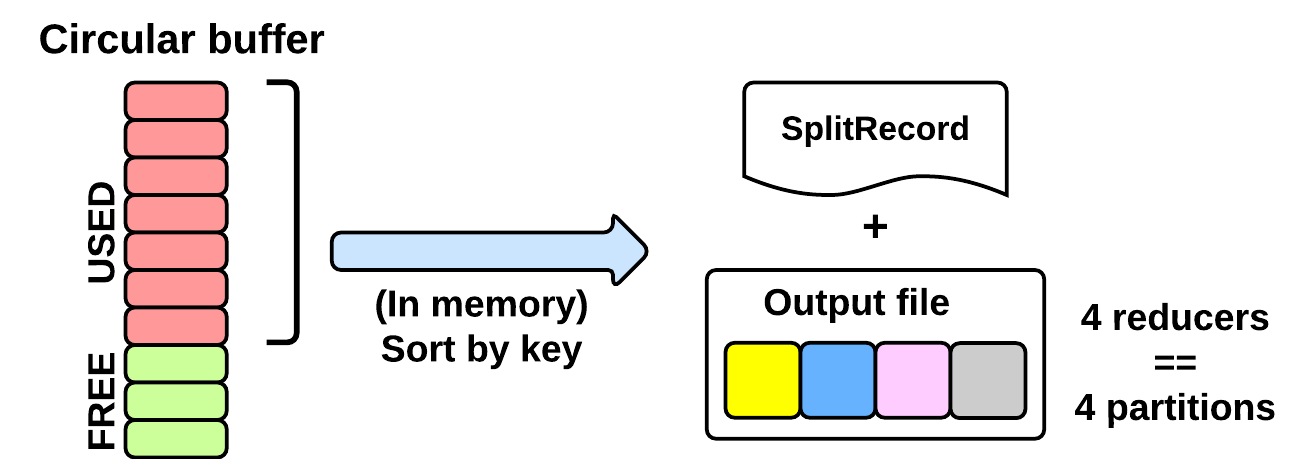


The EXECUTION phase is performed by the run method of the Mapper class. The user can override it, but by default it will start by calling the setup method: this function by default does not do anything useful but can be override by the user in order to setup the Task (e.g., initialize class variables). After the setup, for each <key, value> tuple contained in the map split, the map() is invoked. Therefore, map() receives: a key a value, and a mapper context. Using the context, a map stores its output to a buffer.

Notice that the map split is fetched chuck by chunk (e.g., 64KB) and each chunk is split in several (key, value) tuples (e.g., using SplitLineReader.class). This is done inside the Mapper.Context.nextKeyValue method.

When the map split has been completely processed, the run function calls the clean method: by default, no action is performed but the user may decide to override it.

#### [MapTask: SPILLING](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html#maptask-spilling)



As seen in the EXECUTING phase, the map will write (using Mapper.Context.write()) its output into a circular in-memory buffer (MapTask.MapOutputBuffer). The size of this buffer is fixed and determined by the configuration parameter mapreduce.task.io.sort.mb (default: 100MB).

Whenever this circular buffer is almost full (mapreduce.map. sort.spill.percent: 80% by default), the SPILLING phase is performed (in parallel using a separate thread). Notice that if the splilling thread is too slow and the buffer is 100% full, then the map() cannot be executed and thus it has to wait.

The SPILLING thread performs the following actions:

1. it creates a SpillRecord and FSOutputStream (local filesystem)
2. in-memory sorts the used chunk of the buffer: the output tuples are sorted by (partitionIdx, key) using a quicksort algorithm.
3. the sorted output is split into partitions: one partition for each ReduceTask of the job (see later).
4. Partitions are sequentially written into the local file.

##### [How Many Reduce Tasks?](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html#how-many-reduce-tasks)

The number of ReduceTasks for the job is decided by the configuration parameter mapreduce.job.reduces.

#### [What is the partitionIdx associated to an output tuple?](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html#what-is-the-partitionidx-associated-to-an-output-tuple)

The paritionIdx of an output tuple is the index of a partition. It is decided inside the Mapper.Context.write():

partitionIdx = (key.hashCode() & Integer.MAX\_VALUE) % numReducers

It is stored as metadata in the circular buffer alongside the output tuple. The user can customize the partitioner by setting the configuration parameter mapreduce.job.partitioner.class.

#### [When do we apply the combiner?](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html#when-do-we-apply-the-combiner)

If the user specifies a combiner then the SPILLING thread, before writing the tuples to the file (4), executes the combiner on the tuples contained in each partition. Basically, we:

1. create an instance of the user Reducer.class (the one specified for the combiner!)
2. create a Reducer.Context: the output will be stored on the local filesystem
3. execute Reduce.run(): see Reduce Task description

The combiner typically use the same implementation of the standard reduce() function and thus can be seen as a local reducer.

### [MapTask: end of EXECUTION](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html#maptask-end-of-execution)

At the end of the EXECUTION phase, the SPILLING thread is triggered for the last time. In more detail, we:

1. sort and spill the remaining unspilled tuples
2. start the SHUFFLE phase

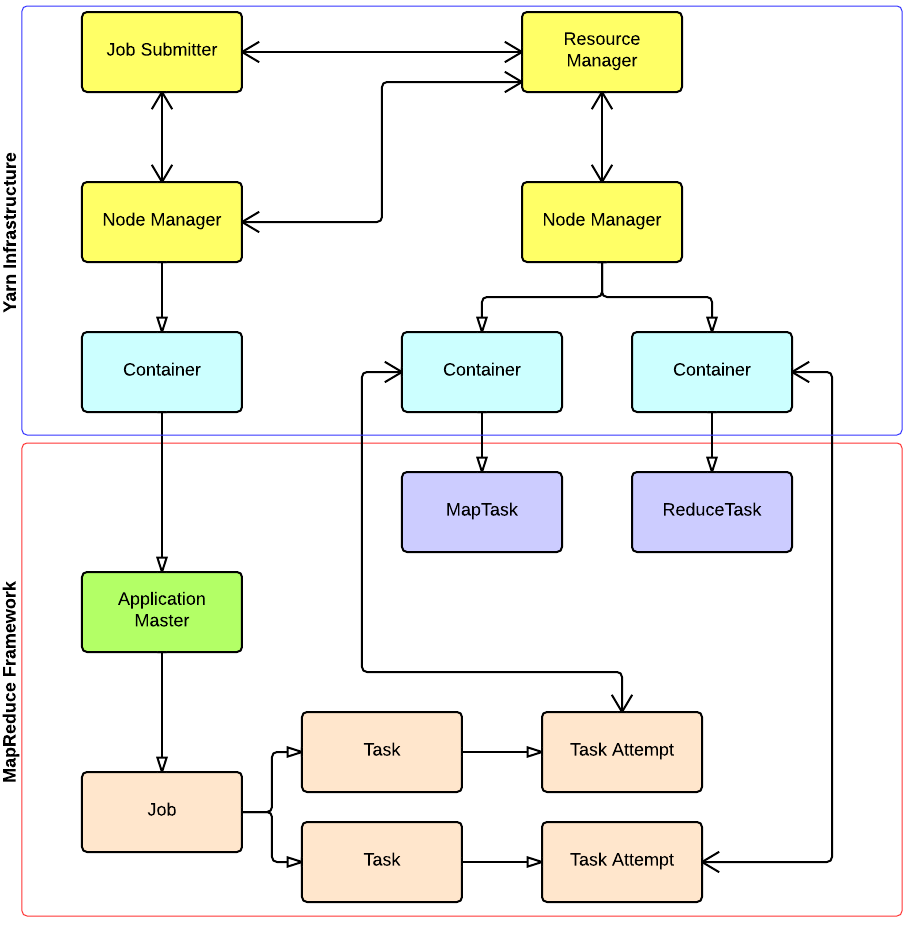
Notice that for each time the buffer was almost full, we get one spill file (SpillReciord + output file). Each Spill file contains several partitions (segments).

#### [MapTask: SHUFFLE](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html#maptask-shuffle)

### [Reduce Phase](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html#reduce-phase)

[…]

### [YARN and MapReduce interaction](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html#yarn-and-mapreduce-interaction)



## Hadoop Distributed File System

Hadoop can work directly with any mountable distributed file system such as Local FS, HFTP FS, S3 FS, and others, but the most common file system used by Hadoop is the Hadoop Distributed File System (HDFS).

The Hadoop Distributed File System (HDFS) is based on the Google File System (GFS) and provides a distributed file system that is designed to run on large clusters (thousands of computers) of small computer machines in a reliable, fault-tolerant manner.

HDFS uses a master/slave architecture where master consists of a single **NameNode** that manages the file system metadata and one or more slave **DataNodes** that store the actual data.

A file in an HDFS namespace is split into several blocks and those blocks are stored in a set of DataNodes. The NameNode determines the mapping of blocks to the DataNodes. The DataNodes takes care of read and write operation with the file system. They also take care of block creation, deletion and replication based on instruction given by NameNode.