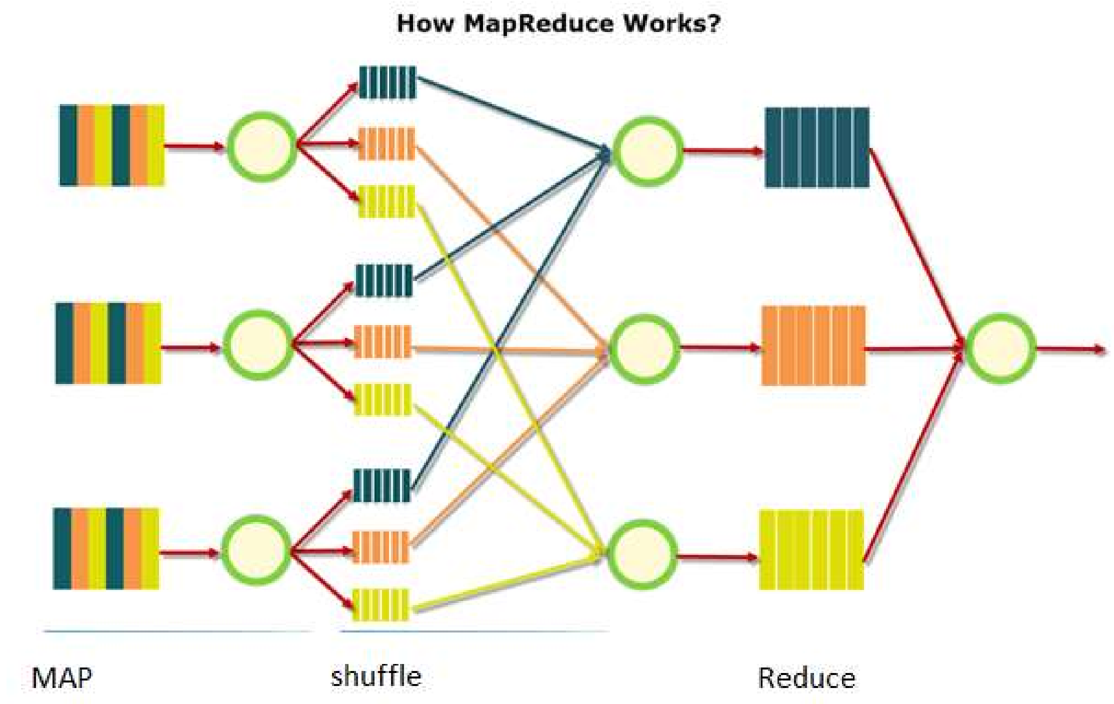
Parallel Processing

Dealing with massive amount of data sets requires efficient processing methods that will reduce run-time and cost in addition to overcome memory constraints. Parallel Processing is one of the most used methods by data scientists when it comes to compute and data-intensive tasks, where a task is broken up to multiple parts with a software tool and each part is distributed to a processor, then each processor will perform the assigned part. Finally, the parts are reassembled to deliver the final solution or execute the task.

What is MapReduce?

MapReduce was designed by Google as a programming model for processing large data sets with a parallel, distributed algorithm on a cluster. Though, MapReduce was originally Google proprietary technology, it has been quite a generalized term in the recent time.

MapReduce comprises a Map() and Reduce() procedures. Procedure Map() performance filtering and sorting operation on data where as procedure Reduce() performs a summary operation of the data. This model is based on modified concepts of the map and reduce functions commonly available in functional programming. The library where the procedure Map () and Reduce () belongs is written in many different languages



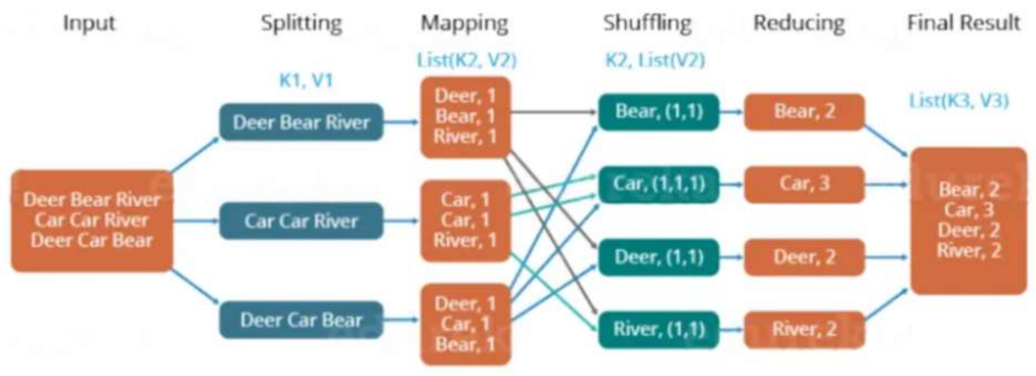
Map() Procedure

There is always a master node in this infrastructure which takes an input. Right after taking input master node divides it into smaller sub-inputs or sub-problems. These sub-problems are distributed to worker nodes. A worker node later processes them and does necessary analysis. Once the worker node completes the process with this sub-problem it returns it back to master node.

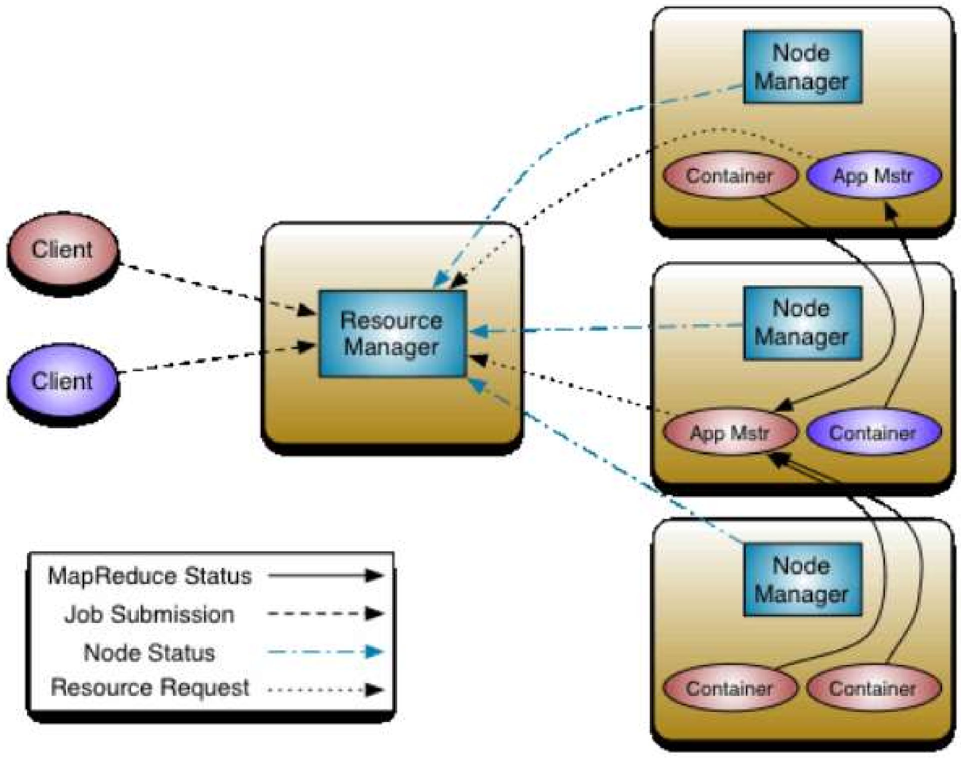
Reduce() Procedure

All the worker nodes return the answer to the sub-problem assigned to them to master node. The master node collects the answer and once again aggregate that in the form of the answer to the original big problem which was assigned master node.

Map Reduce & Yarn



Yarn:



What is YARN?

YARN (Yet Another Resource Negotiator) is a critical component of the Hadoop ecosystem. It functions as the cluster resource management layer, responsible for managing and allocating resources such as CPU, memory, and storage for distributed applications running on a Hadoop cluster.

Benefit of YARN

1. **Centralized Resource Management:** YARN provides a centralized resource management system that can dynamically allocate and manage resources for different frameworks/applications running on the Hadoop cluster.

2. **Flexibility:** The scheduling and resource management capabilities are separated from the data processing component. This allows for running various types of data processing applications on a single cluster.

3. **Better Cluster Utilization:** YARN's dynamic resource allocation ensures each application gets the resources it needs to run successfully without affecting other applications. Additionally, resources unutilized by one framework/application can be consumed by another.

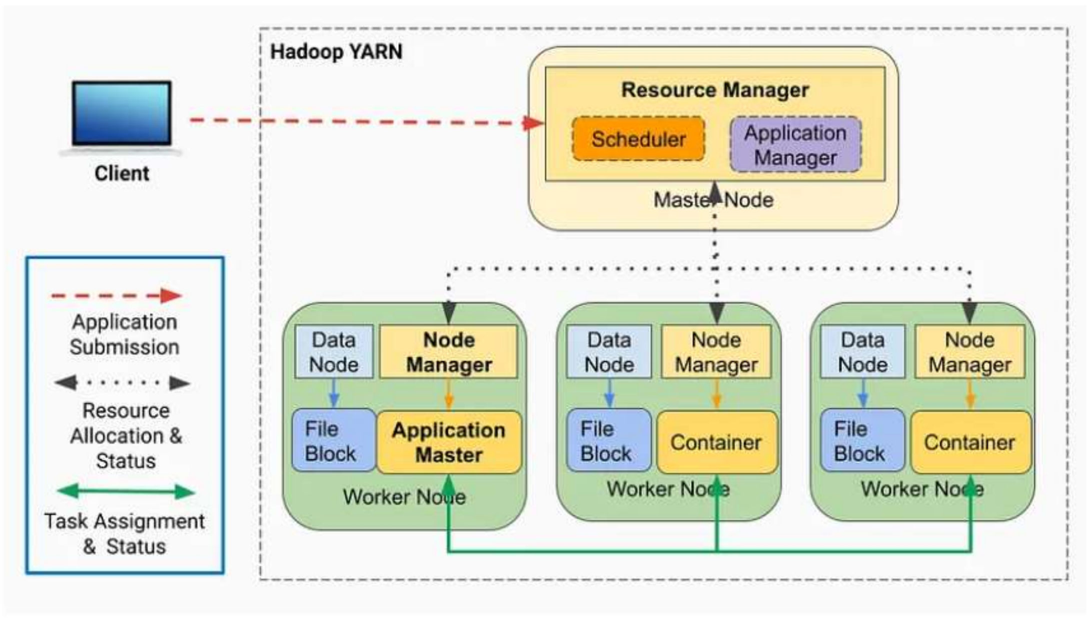
4. **Cost-Effective:** With YARN, one "do-it-all" Hadoop cluster can run a diverse set of workloads and support a variety of applications, making it a more efficient and cost-effective platform for big data processing.

5. **Reduced Data Movement:** As there is no need to move data between Hadoop, YARN, and systems running on different clusters of computers, data motion is reduced.

YARN Architecture and its Components

The three important elements of the YARN architecture are:

1. **Resource Manager (RM)**
2. **Application Master (AM)**
3. **Node Managers (NM)**



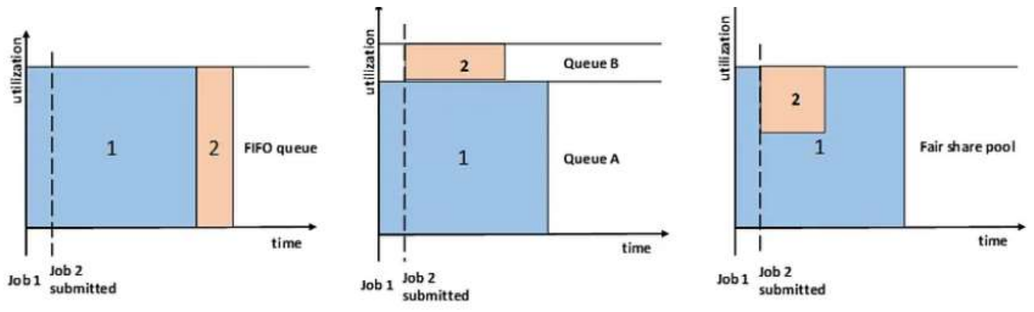
Resource Manager (RM)

The *ResourceManager* is a Java process running on the master node that is responsible for managing and allocating resources such as CPU, memory, and disk across the Hadoop cluster based on the needs of various jobs.

- **Cluster resource tracking:** It maintains a global view of the cluster and tracks the available resources on each node.
- **Cluster health monitoring:** It monitors the health of nodes in the cluster and manages the failover of resources in case of node failures.
- **Cluster resource allocation:** It receives resource requests from application masters and allocates the necessary resources to run the application.
- **Job scheduling:** See *Scheduler*
- **Application master management:** See *Applications Manager*

The ResourceManager has two main components:

1. **Scheduler:** It performs the scheduling of jobs based on policies and priorities defined by the administrator.
2. **Applications Manager:** It monitors the health of the Application Master in the cluster and manages failover in case of failures.



(a) FIFO scheduler (b) Capacity Scheduler (c) Fair Scheduler

Schedulers

The *scheduler* in YARN is responsible for **scheduling and mediating available resources in the cluster** among submitted applications, in accordance with a defined policy. It allows different policies for managing constraints such as capacity, fairness, and service level agreements (SLAs).

There are three types of scheduling policies available in YARN:

1. **FIFO Scheduler:** It is a simple scheduler that follows a first-come, first-served approach and it's suitable for smaller clusters and simple workloads.
2. **Capacity Scheduler:** It is a scheduler that divides cluster resources into multiple queues, each with its own reserved resources while able to dynamically utilize unused resources from other queues. It is suitable for large-scale, multi-user environments.
3. **Fair Scheduler:** It is a scheduler that is designed to balance resources fairly and equally among accepted jobs, without requiring a set amount of reserved capacity. It is suitable for the cluster that runs jobs with varying sizes and resource requirements.

Application Manager

The *ApplicationManager* is an interface that maintains a list of applications that are submitted, running, or completed. It is responsible for:

- **Handling job submission:** Accepting job submissions to YARN,

- **Negotiating resources for the ApplicationMaster:** Negotiating the first container for executing the application-specific application master, and
- **Managing failover of the ApplicationMaster:** Restarting the application master container on failure.

Application Master (AM)

The *ApplicationMaster* is a process that runs the main function/entry point of an application, such as the Spark driver. It has several responsibilities, including:

- **Requesting resources:** Negotiating with the Resource Manager to obtain resources for launching containers to execute tasks.
- **Running the Master/Driver program:** It runs the Master/Driver program, such as Spark Driver, which devises the job execution plan, assigns tasks in the allocated containers, tracks task execution status, monitors progress, and handles task failures.

Node Manager (NM)

NodeManagers are Java processes that run on slave/worker nodes in a Hadoop cluster. They are responsible for managing containers and resources on each worker node, providing a secure runtime environment for applications, and allowing for efficient and flexible resource allocation.

- **Reporting node health:** Each Node Manager announces itself to the ResourceManager and periodically sends a heartbeat to provide node status and information, including memory and virtual cores. In case of failure, the Node Manager reports any issues to the Resource Manager, diverting resource allocations to healthy nodes.

- **Launching Containers:** Node Managers take instructions from the ResourceManager, launch containers on their nodes, and set up the container environment with the specified resource constraints.

- **Container Management:** Node Managers manage container life cycle, dependencies, leases, resource usage, and log management.

Container

A container is a unit of resource allocation, an abstraction representing on a specific worker node in a Hadoop cluster. Containers are assigned to execute tasks from applications, such as MapReduce jobs or Spark tasks. Each container has a specific amount of resources allocated to it, such as CPU, memory, and disk space, allowing the task to run in a controlled and isolated environment.

- **ResourceManager:** The ResourceManager in YARN is responsible for *allocating containers to Application Masters* based on their resource requests. It *provides the container launch context (CLC)* that includes environment variables, dependencies, security tokens, and commands to create the application's launch process.

- **NodeManager:** The NodeManager in YARN is responsible for *launching the containers* with specified resource constraints (CLC).

- **ApplicationMasters:** The Application Masters *manage the execution of tasks within these containers*, monitor their progress, and handle any task failures or reassignments.

Example use case: what happens when a job is submitted in YARN ?

- **Step 1:** A client submits a job to the YARN Resource Manager.
- **Step 2:** The job enters a scheduler queue in the ResourceManager, waiting to be executed.
- **Step 3:** When it is time for the job to be executed, the ResourceManager finds a NodeManager capable of launching a container to run the ApplicationMaster.
- **Step 4:** The ApplicationMaster launches the Driver Program.
- **Step 5:** The ApplicationMaster calculates the required resources (CPU, RAM, number of executors) for the job and sends a request to the Resource Manager to launch the executors.
  > The ApplicationMaster communicates with the NameNode to determine the file (block) locations within the cluster using the HDFS protocol.

- **Step 6:** The Driver Program assigns tasks to the executor containers and keeps track of the task status.
- **Step 7:** The executor containers execute the tasks and return the results to the Driver Program. The Driver Program aggregates the results and produces the final output.

Container or Task Failure

- Container failure will be handled by node-manager. When a container fails or dies, node-manager detects the failure event and launches a new container to replace the failing container and restart the task execution in the new container.
- If no other container resource available in the same node, Resource manager notifies it to application master and Application master schedules it again on a different node requesting container from Node manager.
- Tasks fails because of a runtime exception in user code, or a bug in JVM which will be communicated to the AppMaster, hence AM request a new container to launch the failed task to RM.

Application Master Failure

- In this Resource manager will start a new Application Master under node manager.
- No task needs to re-run as the ability to recover the associated task state depends on the application-master implementation. MapReduce application-master has the ability to recover the state but it is not enabled by default.
- Job client when polls the application masters it will get a timeout exception as it caches the application master. So now it needs to go to resource manager and get the address of new application master.

Node Manager Failure

- In this case the tasks in containers which were running needs to be rescheduled.
- Application master gets another container for the tasks and runs it on a different machine under node manager.