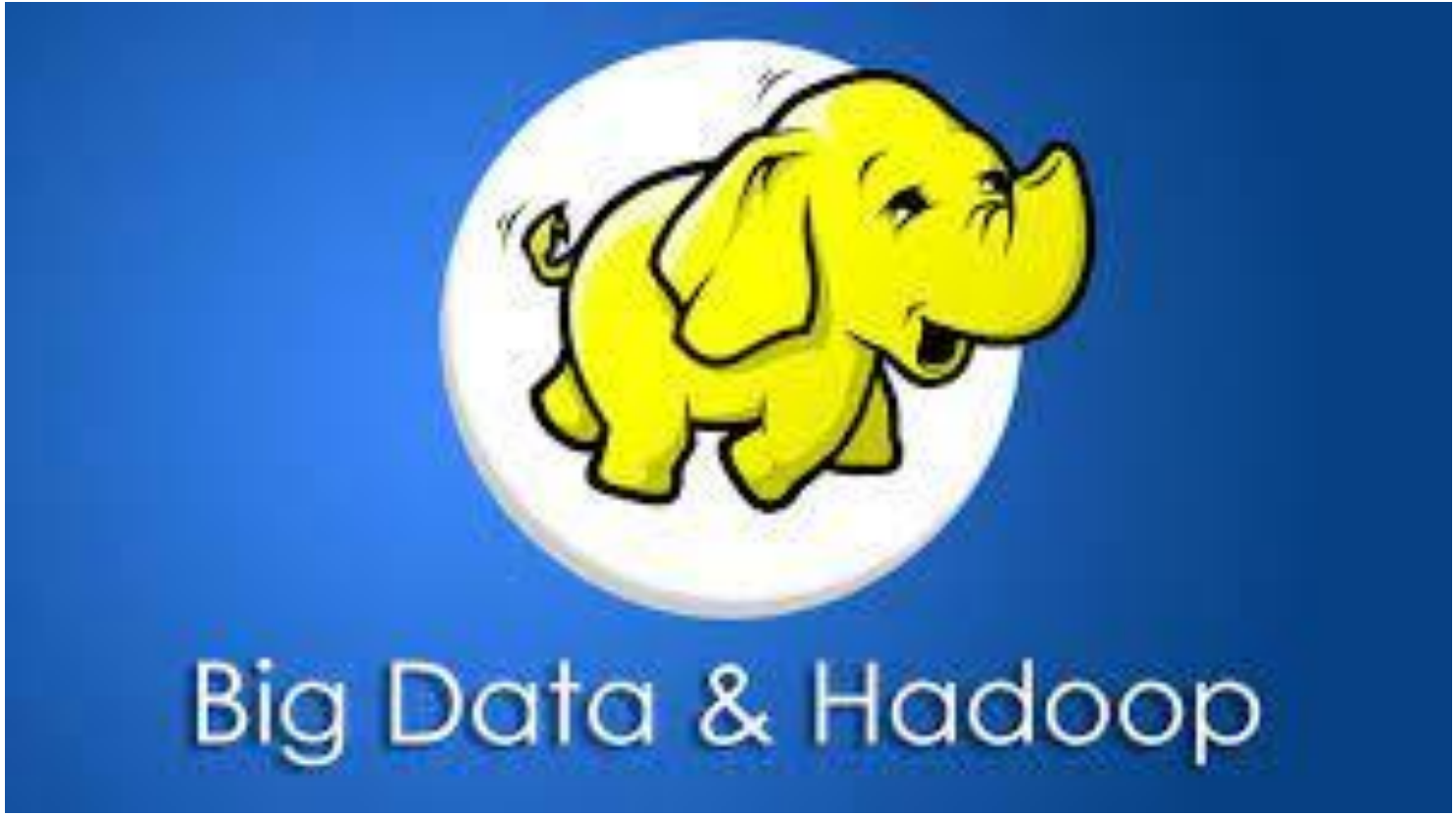
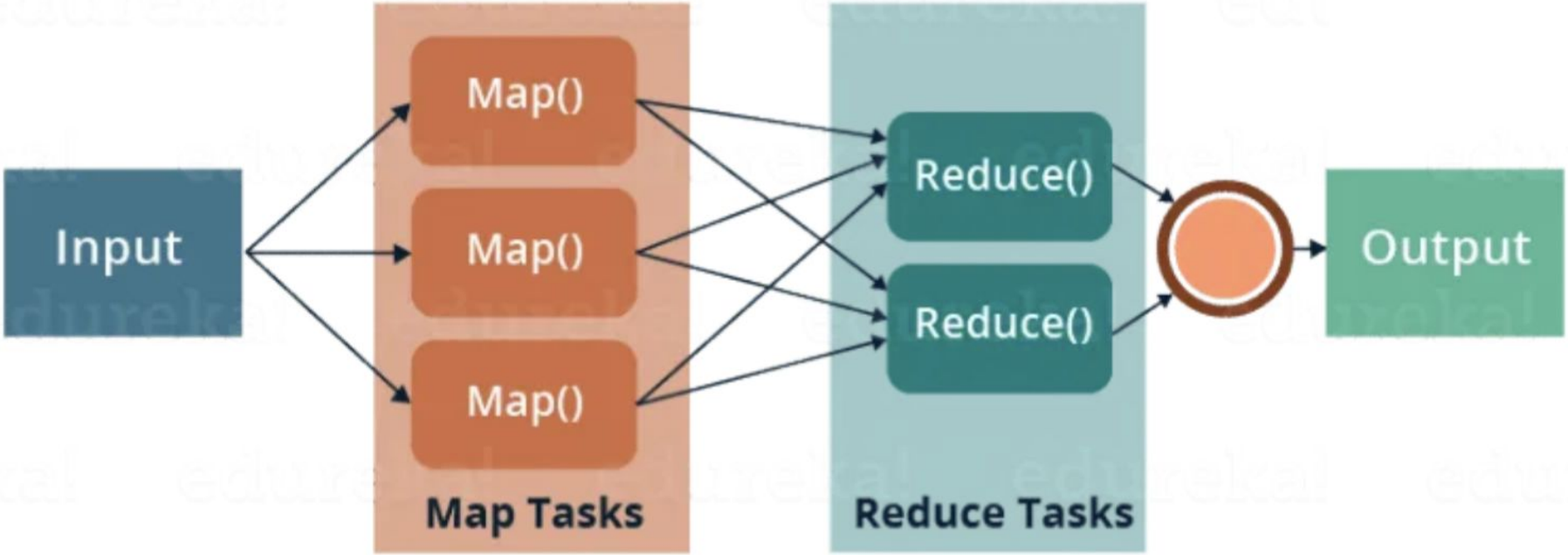

Big Data & Hadoop

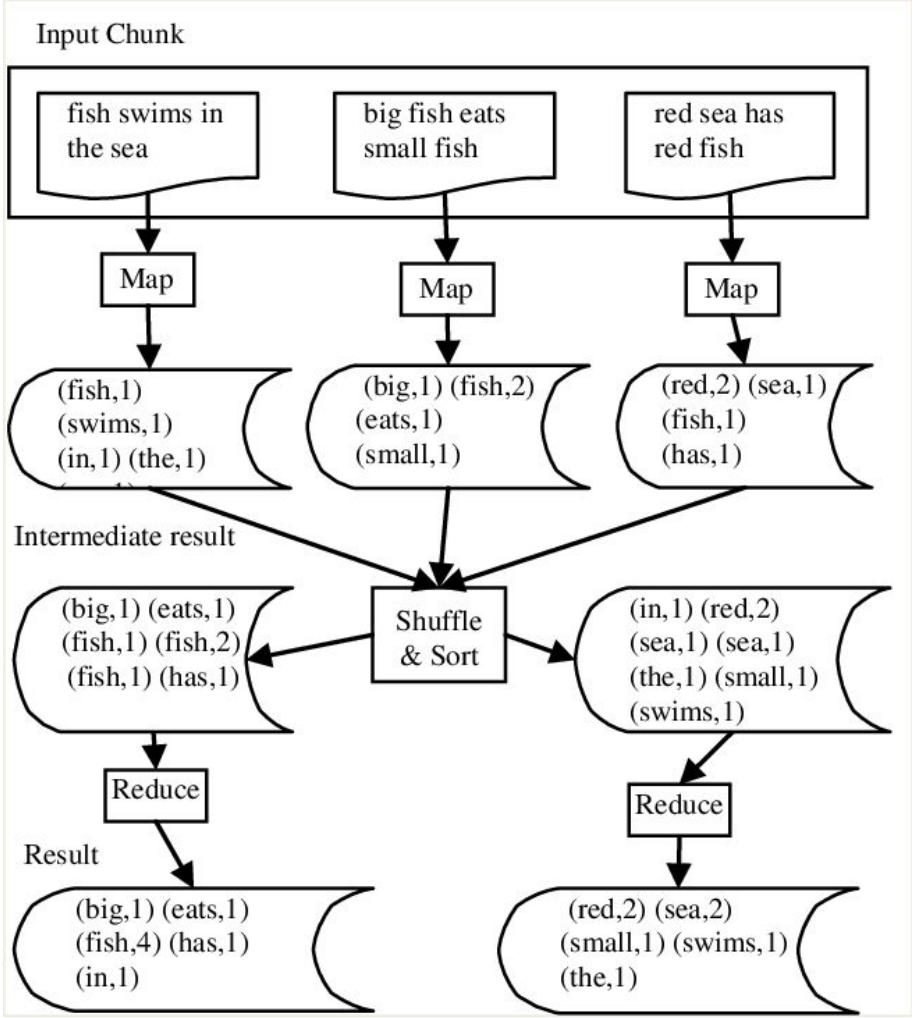
Map-Reduce

GROW DATA SKILLS

MapReduce is a programming framework that allows us to perform distributed and parallel processing on large data sets in a distributed environment.

- MapReduce consists of two distinct tasks — **Map** and **Reduce**.
- As the name MapReduce suggests, reducer phase takes place after the mapper phase has been completed.
- So, the first is the map job, where a **block of data is read and processed** to produce **key-value pairs** as intermediate outputs.
- The output of a Mapper or map job (key-value pairs) is **input to the Reducer**.
- The reducer receives the key-value pair from **multiple map jobs**.
- Then, the reducer **aggregates** those intermediate data tuples (intermediate key-value pair) into a smaller set of tuples or key-value pairs which is the final output.





Advantages of MapReduce

GROW DATA SKILLS

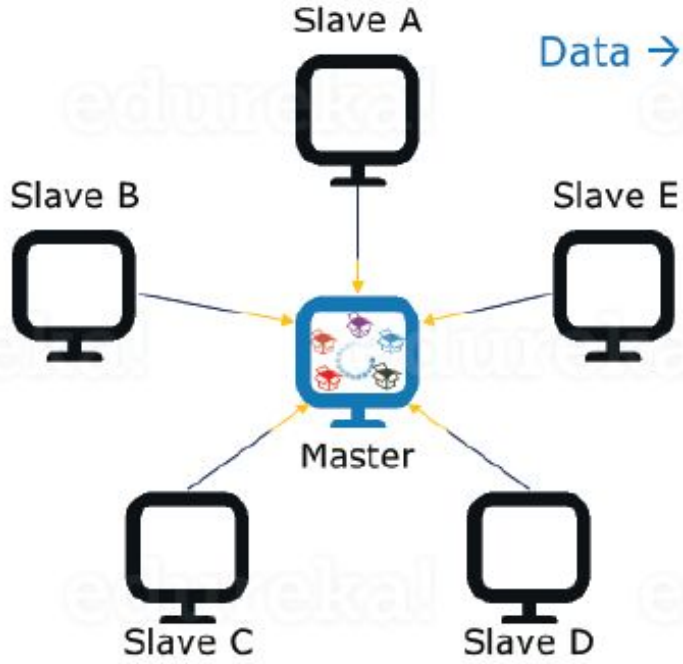
- **Parallel Processing:** In MapReduce, we are dividing the job among multiple nodes and each node works with a part of the job simultaneously. So, MapReduce is based on Divide and Conquer paradigm which helps us to process the data using different machines very quickly.

- **Data Locality:** Instead of moving data to the processing unit, we are moving the processing unit to the data in the MapReduce Framework. In the traditional system, we used to bring data to the processing unit and process it. But, as the data grew and became very huge, bringing this huge amount of data to the processing unit posed the following issues:

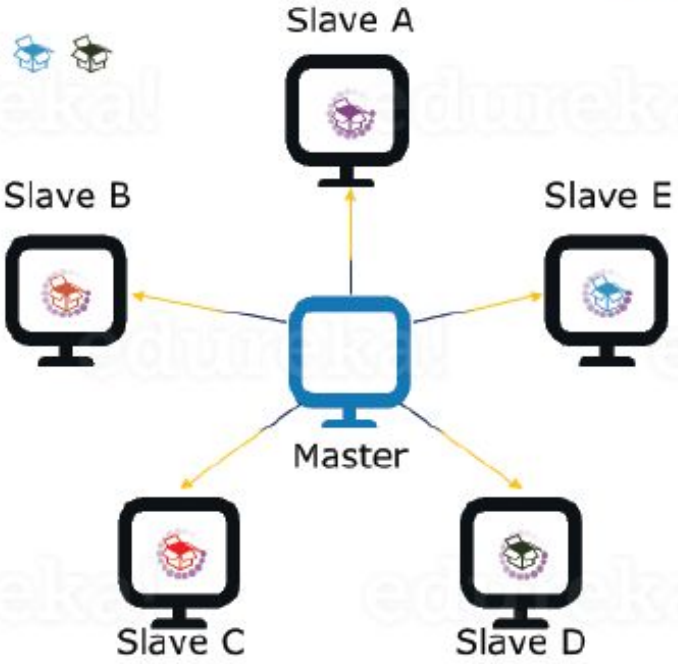
    - Moving huge data to processing is costly and deteriorates the network performance.
    - Processing takes time as the data is processed by a single unit which becomes the bottleneck.
    - The master node can get over-burdened and may fail.

    Now, MapReduce allows us to overcome the above issues by bringing the processing unit to the data. This allows us to have the following advantages:

    - It is very cost-effective to move processing unit to the data.
    - The processing time is reduced as all the nodes are working with their part of the data in parallel.
    - Every node gets a part of the data to process and therefore, there is no chance of a node getting overburdened.

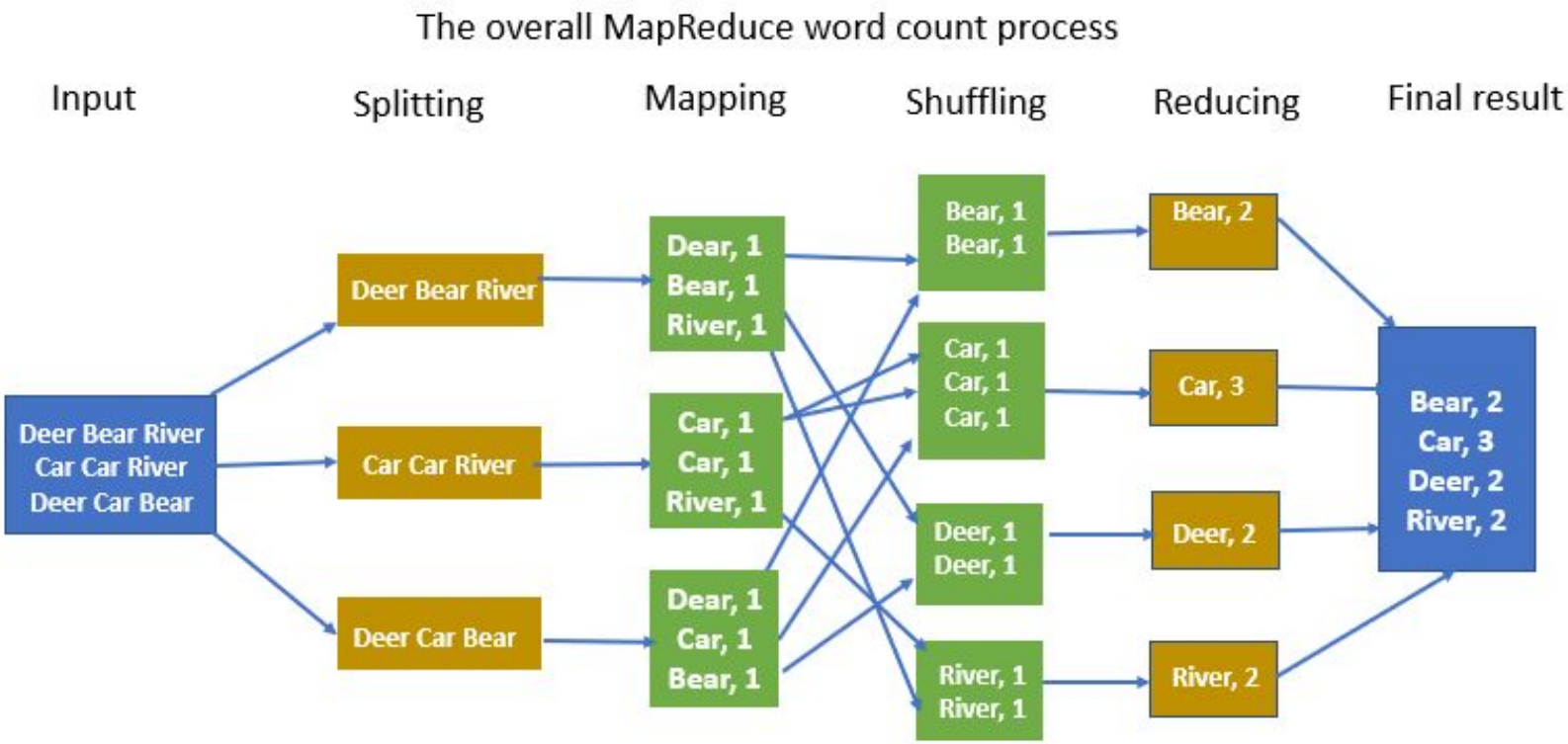


1. Moving data to the Processing Unit
(Traditional Approach)

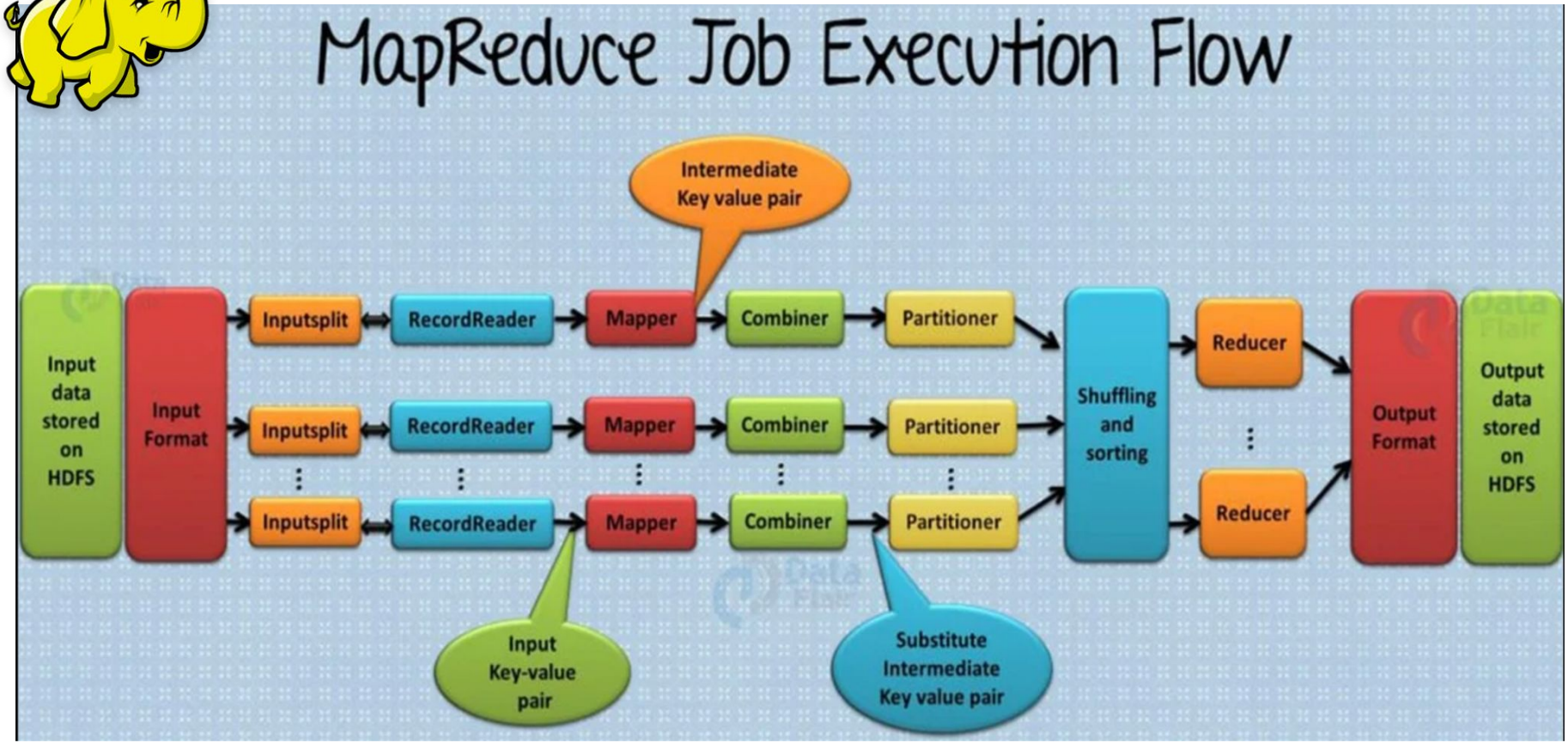


2. Moving Processing Unit to the data
(MapReduce Approach)

Word Count Problem



Map-Reduce Data Flow



- **Input Files:** The data for a MapReduce task is stored in input files, and input files typically lives in HDFS.
- **InputFormat:** Now, InputFormat defines how these input files are split and read. It selects the files or other objects that are used for input.
- **InputSplits:** It is created by InputFormat, logically represent the data which will be processed by an individual Mapper. **One map task** is created for each split; thus the number of **map tasks will be equal to the number of InputSplits**.
- **RecordReader:** It communicates with the InputSplit in Hadoop MapReduce and converts the data into key-value pairs suitable for reading by the mapper.
- **Mapper**: It processes each input record (from RecordReader) and generates new key-value pair, and this key-value pair generated by Mapper is completely different from the input pair. The output of Mapper is also known as intermediate output which is written to the local disk. The output of the Mapper is not stored on HDFS as this is temporary data and writing on HDFS will create unnecessary copies.
- **Combiner**: The combiner is also known as 'Mini-reducer'. Hadoop MapReduce Combiner performs local aggregation on the mappers' output, which helps to minimize the data transfer between mapper and reducer.
- **Partitioner:** Hadoop MapReduce, Partitioner comes into the picture if we are working on more than one reducer **(for one reducer partitioner is not used)**.
    - Partitioner takes the output from combiners and performs partitioning. Partitioning of output takes place on the basis of the key and then sorted. By hash function, key (or a subset of the key) is used to derive the partition.

    - According to the key value in MapReduce, each combiner output is partitioned, and a record having the same key value goes into the same partition, and then each partition is sent to a reducer.

- **Shuffling and Sorting:** Now, the output is Shuffled to the reduce node (which is a normal slave node but reduce phase will run here hence called as reducer node). The shuffling is the physical movement of the data which is done over the network. Once all the mappers are finished and their output is shuffled on the reducer nodes, then this intermediate output is merged and sorted, which is then provided as input to reduce phase.

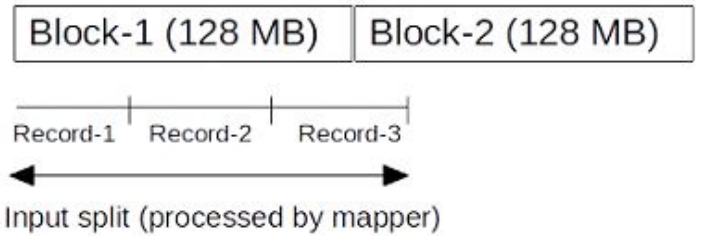
- **Reducer:** It takes the set of intermediate key-value pairs produced by the mappers as the input and then runs a reducer function on each of them to generate the output. The output of the reducer is the final output, which is stored in HDFS.

- **RecordWriter:** It writes these output key-value pair from the Reducer phase to the output files.

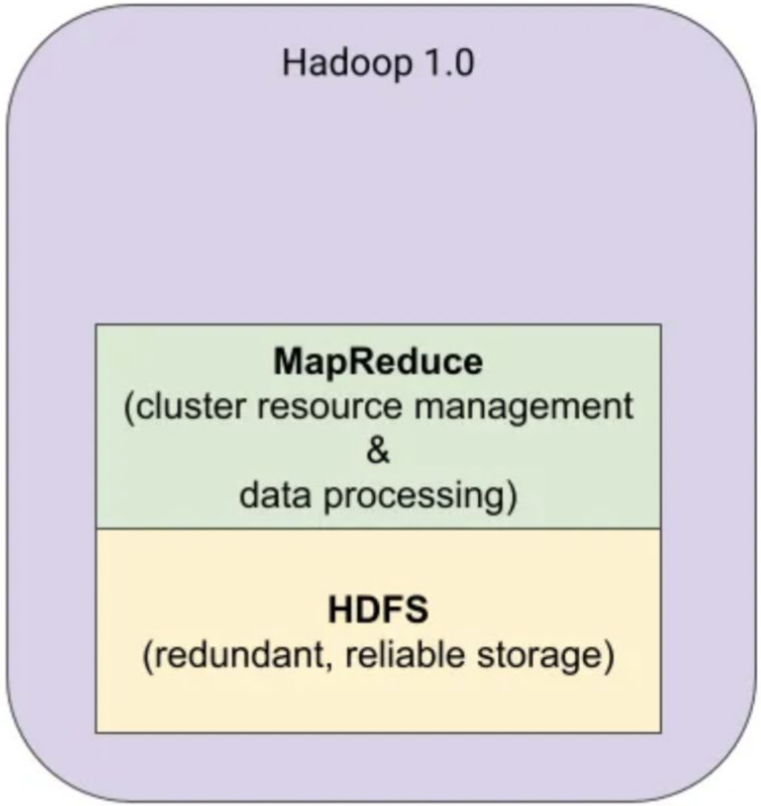
- **OutputFormat:** The way these output key-value pairs are written in output files by RecordWriter is determined by the OutputFormat.

Difference between Input Split & Block

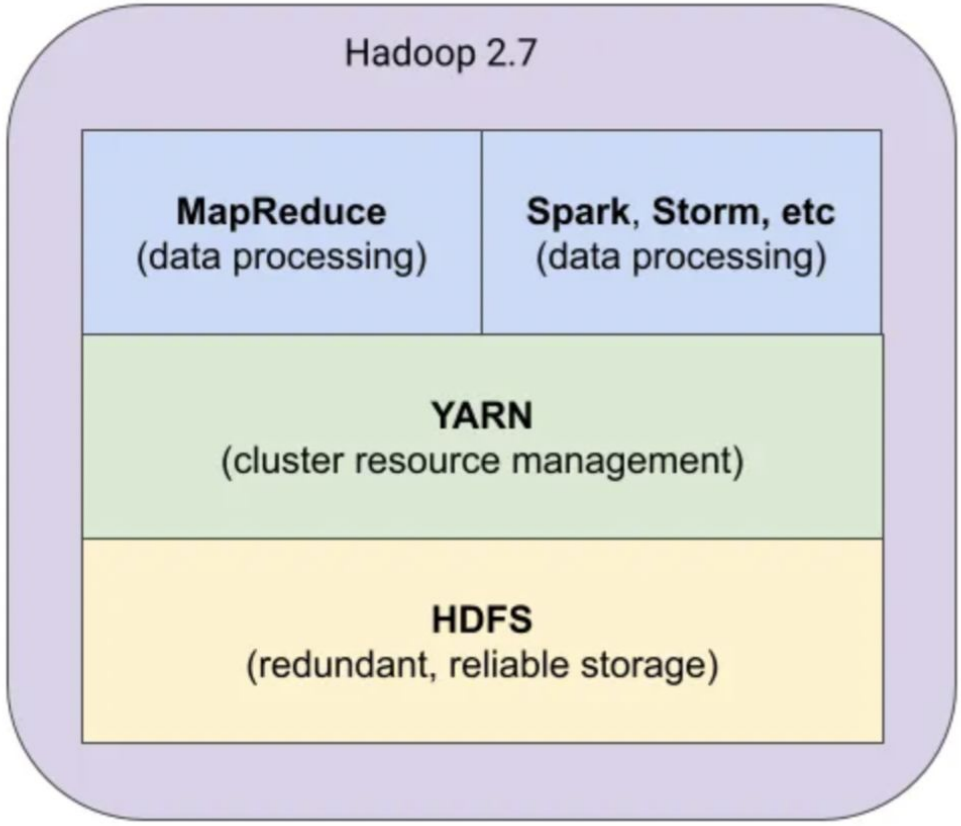
	Input Split	Block
Definition	Input Split is a logical division of data in Hadoop. Each map task processes a single Input Split.	A Block is a physical division of data in Hadoop. The default block size is 128MB in Hadoop 2.x, which can be configured as needed.
Purpose	The purpose of Input Split is to control the parallelism of the MapReduce job. The number of map tasks is equal to the number of Input Splits.	The purpose of a Block is to optimize the disk read performance. Data within a block can be read in a single disk seek.
Dependency	Input Splitting depends on the InputFormat used in the MapReduce job.	The Block size is a configuration parameter in the Hadoop system and is not dependent on the MapReduce job.
Location	Input Splits can cross the boundaries of blocks, but Hadoop tries to keep them within a single block where possible for data locality optimization.	Blocks are distributed across the storage in the Hadoop cluster.



Before 2012
Users could write MapReduce programs using scripting languages



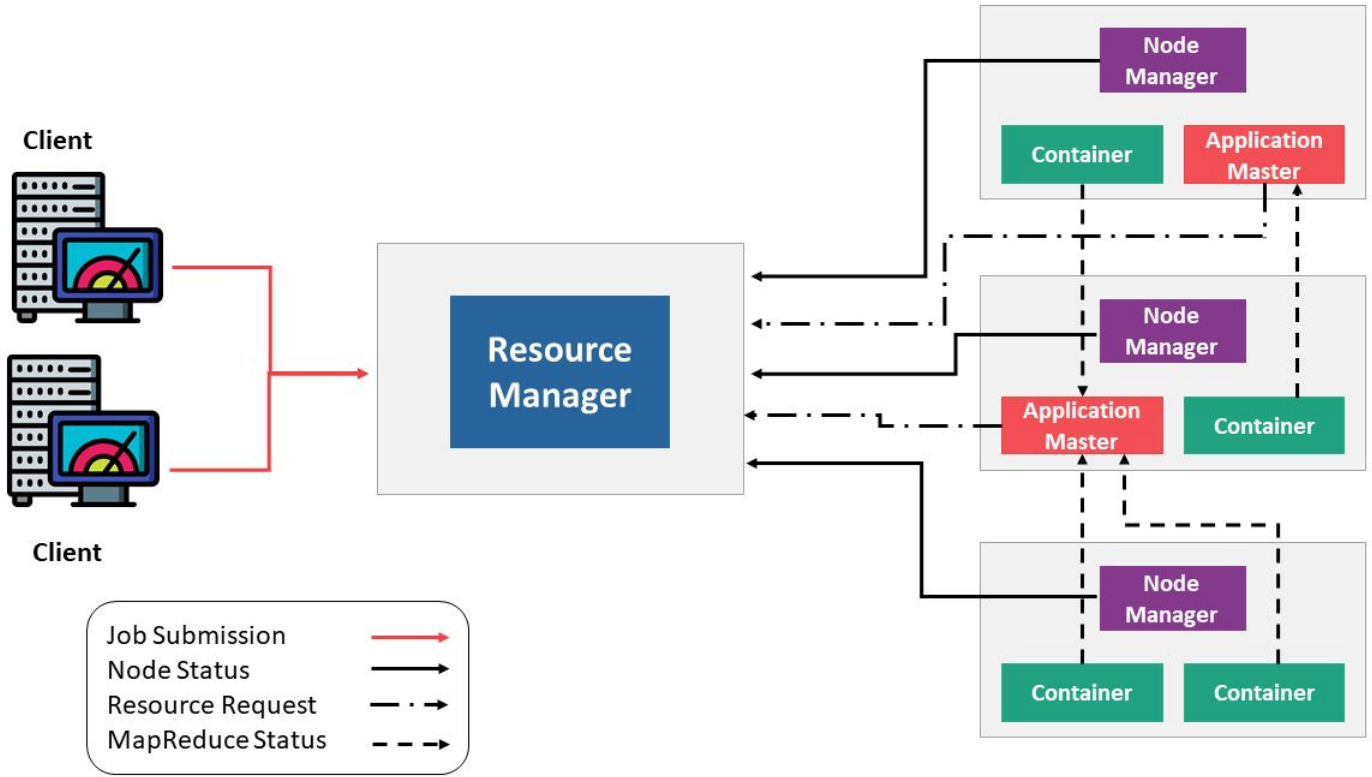
Since 2012
Users could work on multiple processing models in addition to MapReduce



YARN

Apache Hadoop YARN (**Yet Another Resource Negotiator**) is a resource management layer in Hadoop. YARN came into the picture with the introduction of Hadoop 2.x. It allows various data processing engines such as interactive processing, graph processing, batch processing, and stream processing to run and process data stored in HDFS (Hadoop Distributed File System).

Components Of YARN



Components Of YARN

1. **Resource Manager:** Resource Manager is the master daemon of YARN. It is responsible for managing several other applications, along with the global assignments of resources such as CPU and memory. It is used for job scheduling. Resource Manager has two components:
    a. **Scheduler**: Schedulers' task is to distribute resources to the running applications. It only deals with the scheduling of tasks and hence it performs no tracking and no monitoring of applications.

    b. **Application Manager**: The application Manager manages applications running in the cluster. Tasks, such as the starting of Application Master or monitoring, are done by the Application Manager.

2. **Node Manager:** Node Manager is the slave daemon of YARN. It has the following responsibilities:
    a. Node Manager has to monitor the container's resource usage, along with reporting it to the Resource Manager.
    b. The health of the node on which YARN is running is tracked by the Node Manager.
    c. It takes care of each node in the cluster while managing the workflow, along with user jobs on a particular node.
    d. It keeps the data in the Resource Manager updated
    e. Node Manager can also destroy or kill the container if it gets an order from the Resource Manager to do so.

Components Of YARN

3. **Application Master:** Every job submitted to the framework is an application, and every application has a specific Application Master associated with it. Application Master performs the following tasks:

- It coordinates the execution of the application in the cluster, along with managing the faults.
- It negotiates resources from the Resource Manager.
- It works with the Node Manager for executing and monitoring other components' tasks.
- At regular intervals, heartbeats are sent to the Resource Manager for checking its health, along with updating records according to its resource demands.
- Now, we will step forward with the fourth component of Apache Hadoop YARN.

4. **Container:** A container is a set of physical resources (CPU cores, RAM, disks, etc.) on a single node. The tasks of a container are listed below:

- It grants the right to an application to use a specific amount of resources (memory, CPU, etc.) on a specific host.
- YARN containers are particularly managed by a Container Launch context which is Container Life Cycle (CLC). This record contains a map of environment variables, dependencies stored in remotely accessible storage, security tokens, the payload for Node Manager services, and the command necessary to create the process.

Running an Application through YARN

GROW DATA SKILLS

1. **Application Submission:** The RM accepts the application, causing the creation of an ApplicationMaster (AM) instance. The AM is responsible for negotiating resources from the RM and working with the Node Managers (NMs) to execute and monitor the tasks.

2. **Resource Request:** The AM starts by requesting resources from the RM. It specifies what resources are needed, in which locations, and other constraints. These resources are encapsulated in terms of "Resource Containers" which include specifications like memory size, CPU cores, etc.

3. **Resource Allocation:** The Scheduler in the RM, based on the current system load and capacity, as well as policies (e.g., capacity, fairness), allocates resources to the applications by granting containers. The specific strategy depends on the scheduler type (e.g., FIFO, Capacity Scheduler).

4. **Container Launching:** Post-allocation, the RM communicates with relevant NMs to launch the containers. The Node Manager sets up the container's environment, then starts the container by executing the specified commands.

5. **Task Execution:** Each container then runs the task assigned by the ApplicationMaster. These are actual data processing tasks, specific to the application's purpose.

6. **Monitoring and Fault Tolerance:** The AM monitors the progress of each task. If a container fails, the AM requests a new container from the RM and retries the task, ensuring fault tolerance in the execution phase.

7. **Completion and Release of Resources:** Upon task completion, the AM releases the allocated containers, freeing up resources. After all tasks are complete, the AM itself is terminated, and its resources are also released.

8. **Finalization:** The client then polls the RM or receives a notification to know the status of the application. Once informed of the completion, the client retrieves the result and finishes the process.

Running an Application through YARN

GROW DATA SKILLS

1. Run job command
2. Get new application_id
3. Copy job resources
4. Submit Application
5.
    a. Start First Container
    b. Launch Application Master
6. Initialize Job
7. Retrieve Input Splits
8. Allocate Resources
9.
    a. Start requested containers
    b. Launch Task
10. Retrieve job resources