Unit-3

Introduction to Hadoop

Dr. Selva kumar S
Assistant Professor
Dept. of CSE, BMSCE

Learning Objectives and Learning Outcomes

Learning Objectives	Learning Outcomes
Introduction to Hadoop 1. To study the features of Hadoop. 2. To learn the basic concepts of HDFS and MapReduce Programming. 3. To study HDFS Architecture. 4. To study MapReduce Programming Model 5. To study Hadoop Ecosystem.	a) To comprehend the reasons behind the popularity of Hadoop. b) To be able to perform HDFS operations. c) To comprehend MapReduce framework. d) To understand the read and write in HDFS. e) To be able to understand Hadoop Ecosystem.

Agenda

- Processing Data with Hadoop
  - What is MapReduce Programming?
  - How does MapReduce Works?
  - MapReduce Word Count Example
- Managing Resources and Application with Hadoop YARN
  - Limitations of Hadoop 1.0 Architecture
  - Hadoop 2 YARN: Taking Hadoop Beyond Batch
- Hadoop Ecosystem
  - Pig
  - Hive
  - Sqoop
  - HBase

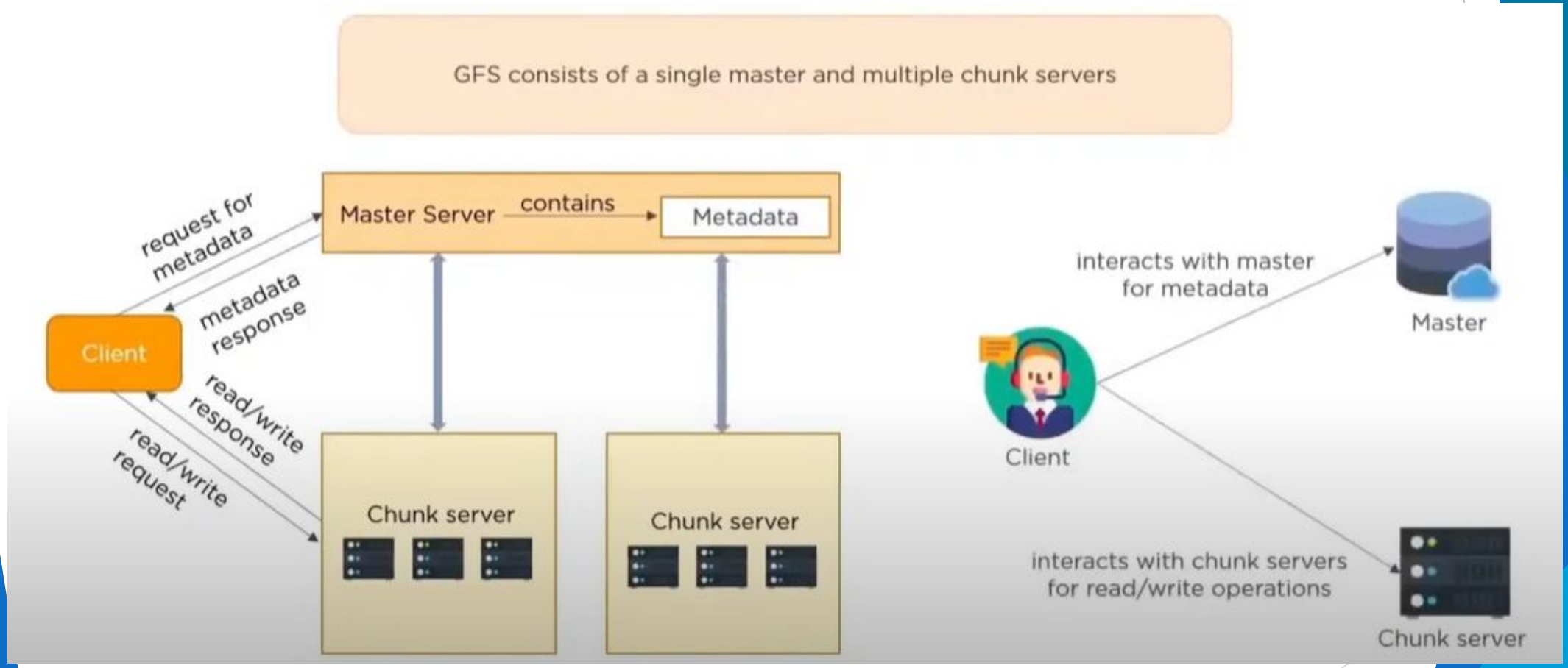
Hadoop

- Hadoop is an open source project by Apache foundation. It is a framework written in Java, originally developed by Doug Cutting in 2005 who named it after his son's toy elephant. He was working with Yahoo then. It was created to support distribution for "Nutch", the text search engine. Hadoop uses Google's MapReduce and Google File System technologies as its foundation.

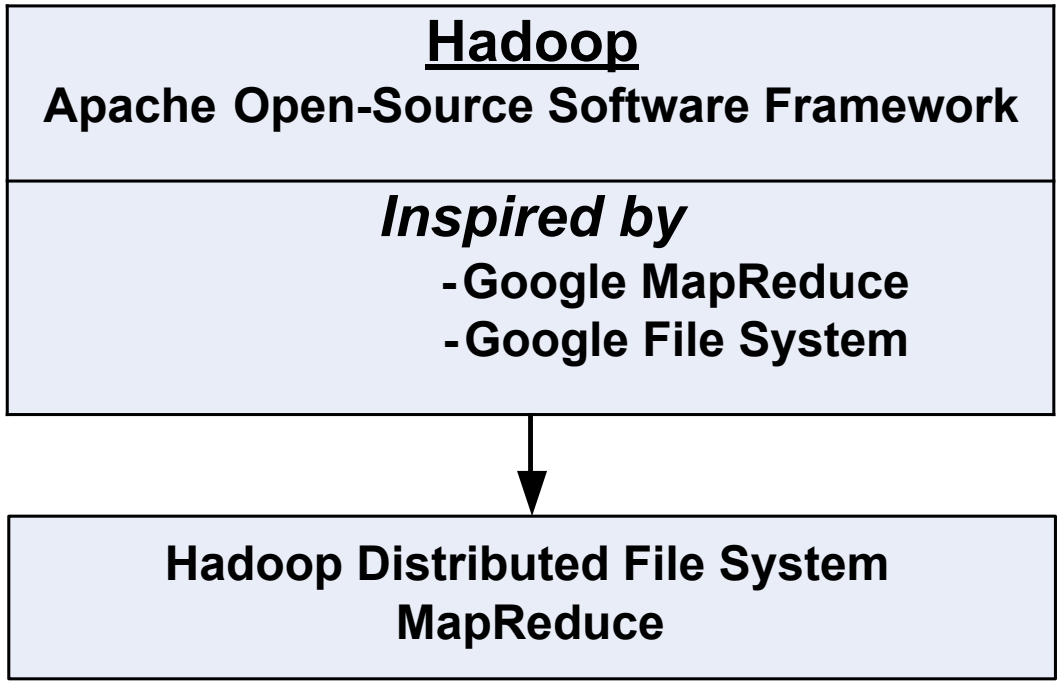
Features of Hadoop

- It is optimized to handle massive quantities of structured, semi-structured and unstructured data, using commodity hardware
- Hadoop has a shared nothing architecture
- It replicates its data across multiple computers
- It is for high throughput rather than low latency
- It complements On-Line Transaction Processing(OLTP) and On-Line Analytical Processing(OLAP) . However, it is not a replacement for DBMS
- It is not good when work cannot be parallelized or when there are dependencies within the data.
- It is not good for processing small files.

GFS Architecture



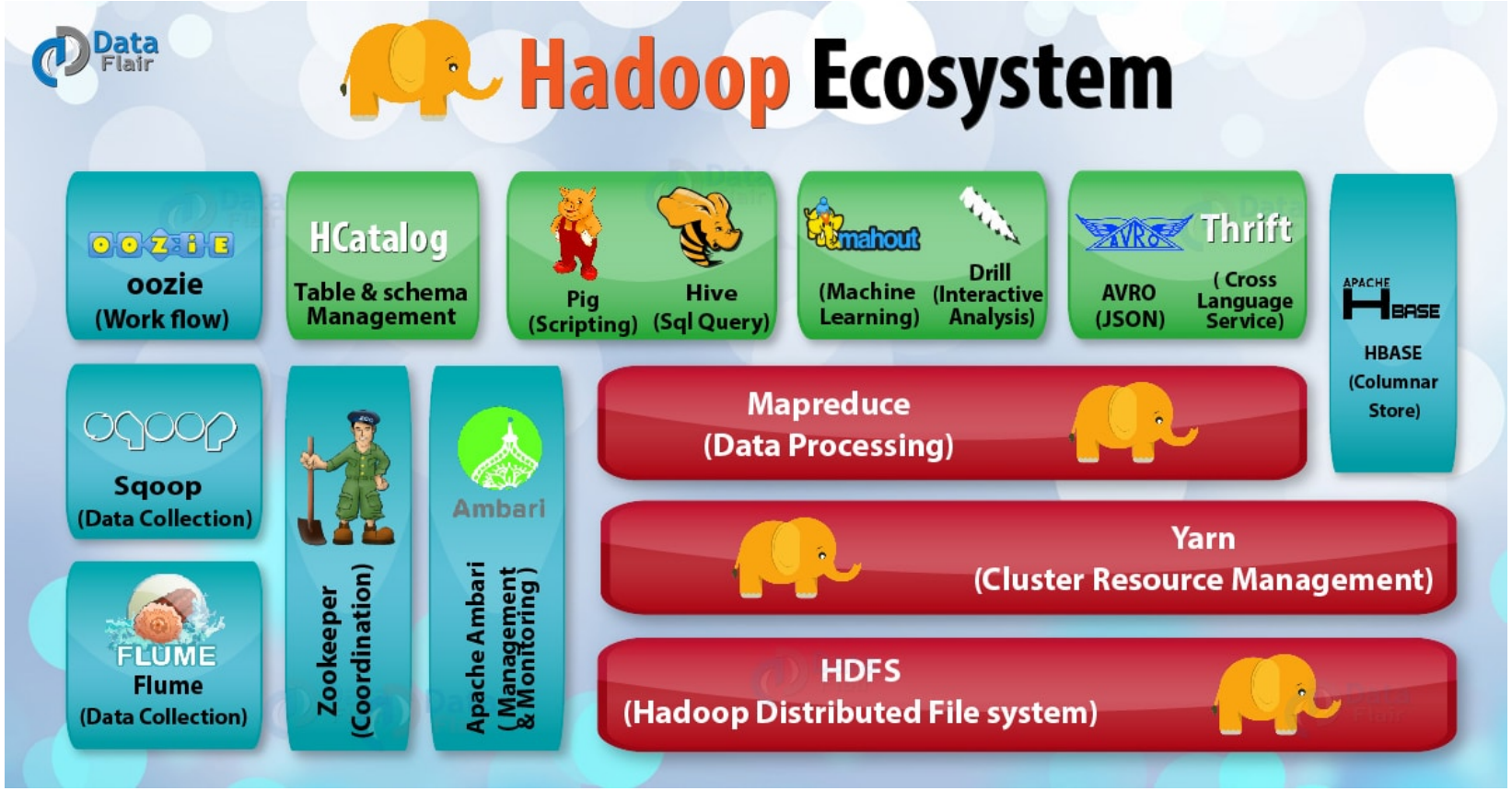
Hadoop


Hadoop
Apache Open-Source Software Framework
Inspired by
-Google MapReduce
-Google File System
Hadoop Distributed File System
MapReduce

Key Advantages of Hadoop

- Stores data in its native format
- Scalable
- Cost-effective
- Resilient to failure
- Flexibility
- Fast

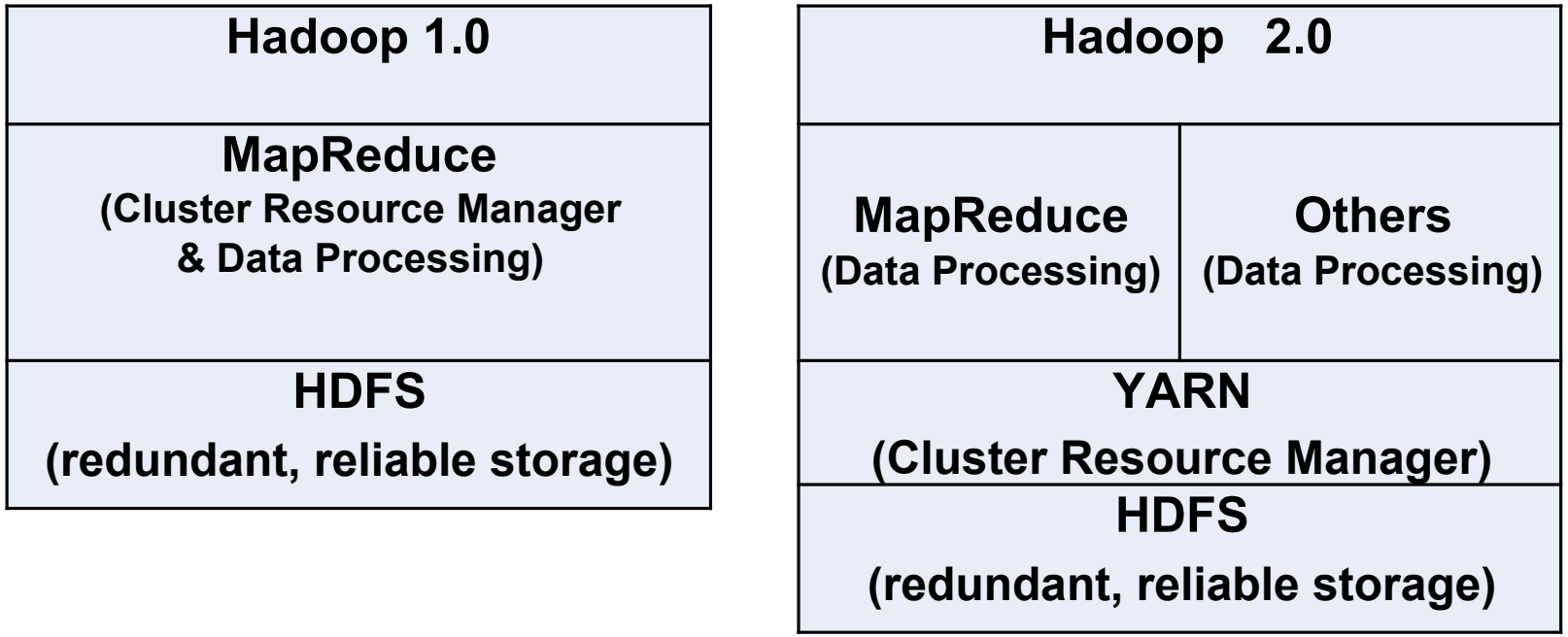
Hadoop EcoSystem



Hadoop Ecosystem

1. **HDFS**: Hadoop Distributed File System. It simply stored data files as close to the original form as possible.
2. **Hbase**: It is Hadoop's datbase and compares well with an RDBMS. It supports structured data storage for large tables.
3. **Hive**: It enables analysis of large data sets using a language very similar to standard ANSI SQL. This implies that anyone familiar with SQL should be able to access data stored on a Hadoop cluster.
4. **Pig**: Pig is an easy to understand data flow language. It helps with the analysis of large datasets which is quite the order with Hadoop.
5. **ZooKeeper**: It is a coordination service for distributed applications.
6. **Oozie**: It is a workflow scheduler system to manage Apache Hadoop jobs.
7. **Mahout**: It is a scalable machine learning and data mining library.
8. **Chukwa**: It is a data collection system for managing large distributed systems.
9. **Sqoop**: It is used to transfer bulk data between Hadoop and structured data stores such as relational databases.
10. **Ambari**: It is a web-based tool for provisioning, managing, and monitoring Apache hadoop clusters.

Versions of Hadoop



Hadoop 1.0 has two main parts

- **Data storage framework** – It is a general-purpose file system called Hadoop Distributed File System(HDFS). HDFS is schema-less. It simply stores data files and these data files can be in any format.
- **Data processing framework-** This is a simple functional programming model initially popularized by Google as MapReduce. It uses two functions- Map and Reduce functions to process data. The “Mappers” take in a set of key-value pairs and generate intermediate data.
- The “Reducers” then act on this input to produce the output data. The two functions work in isolation which enables the processing to be highly distributed in a highly-parallel, fault tolerant, and scalable way.

Hadoop 2.0

- In Hadoop 2.0, HDFS continues to be the data storage framework. A new and separate resource management framework called **Y**et **A**nother **R**esource **N**egotiator(YARN) has been added.
- YARN coordinates the allocation of subtasks of the submitted application, thereby further enhancing the flexibility, scalability, and efficiency of the applications.
- It works by having an ApplicationMaster in place of the JobTracker, running applications on resources governed by a new NodeManager.
- ApplicationMaster is able to run any application and not just MapReduce.

Limitations of Hadoop 1.0

- Requirement for MapReduce programming expertise along with proficiency required in other programming languages, notably Java.
- It supported only Batch processing.
- Tightly computationally coupled with MapReduce, which means that the established data management vendors were left with two options:
  - Either rewrite their functionality in MapReduce so that it could be executed in Hadoop or
  - extract the data from HDFS and process it outside of Hadoop.

Hadoop – An Introduction

What is Hadoop

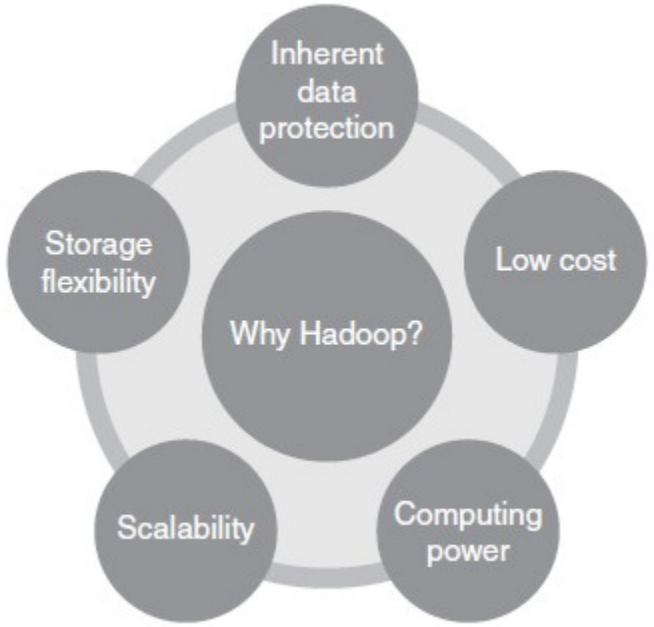
Hadoop is:

Ever wondered why Hadoop has been and is one of the most wanted technologies!!

The key consideration (the rationale behind its huge popularity) is:

Its capability to handle massive amounts of data, different categories of data – fairly quickly.

The other considerations are :



RDBMS versus HADOOP

RDBMS versus HADOOP

PARAMETERS	RDBMS	HADOOP
System	Relational Database Management System.	Node Based Flat Structure.
Data	Suitable for structured data.	Suitable for structured, unstructured data. Supports variety of data formats in real time such as XML, JSON, text based flat file formats, etc.
Processing	OLTP	Analytical, Big Data Processing
Choice	When the data needs consistent relationship.	Big Data processing, which does not require any consistent relationships between data.
Processor	Needs expensive hardware or high-end processors to store huge volumes of data.	In a Hadoop Cluster, a node requires only a processor, a network card, and few hard drives.
Cost	Cost around $10,000 to $14,000 per terabytes of storage.	Cost around $4,000 per terabytes of storage.

Distributed Computing Challenges

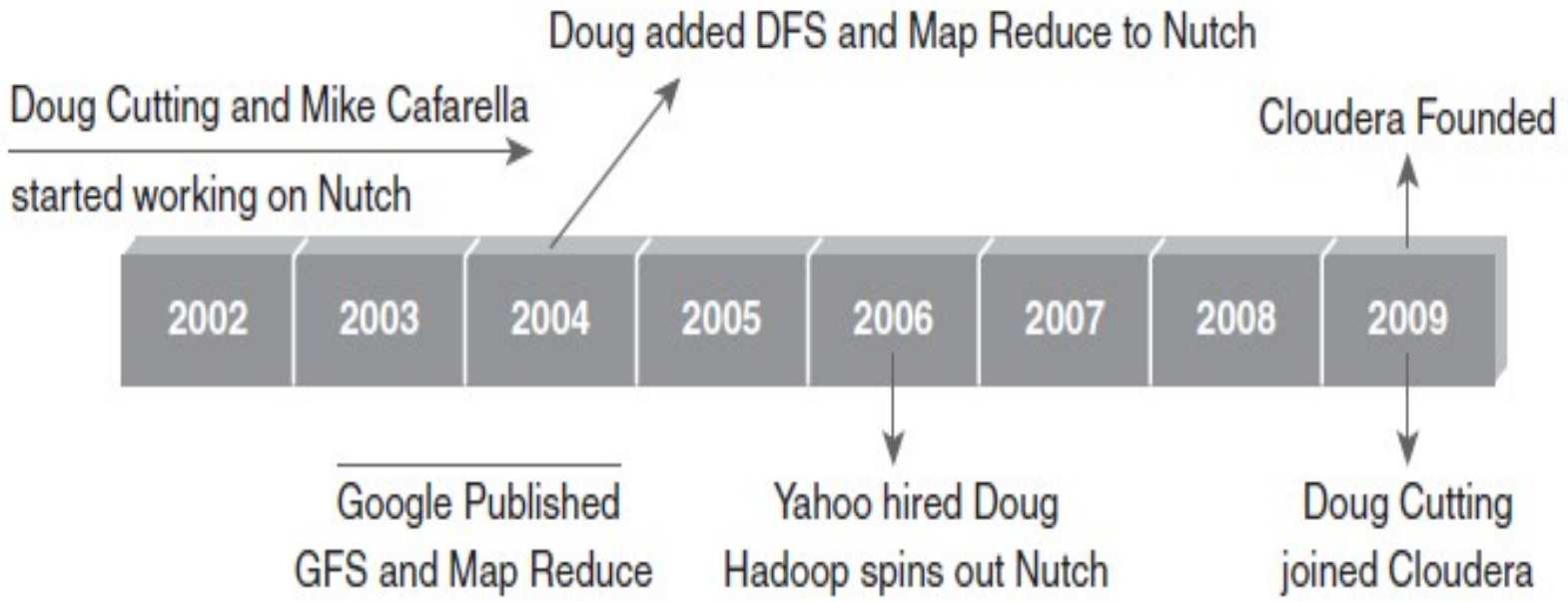
Distributed Computing Challenges

- Hardware Failure

- How to Process This Gigantic Store of Data?

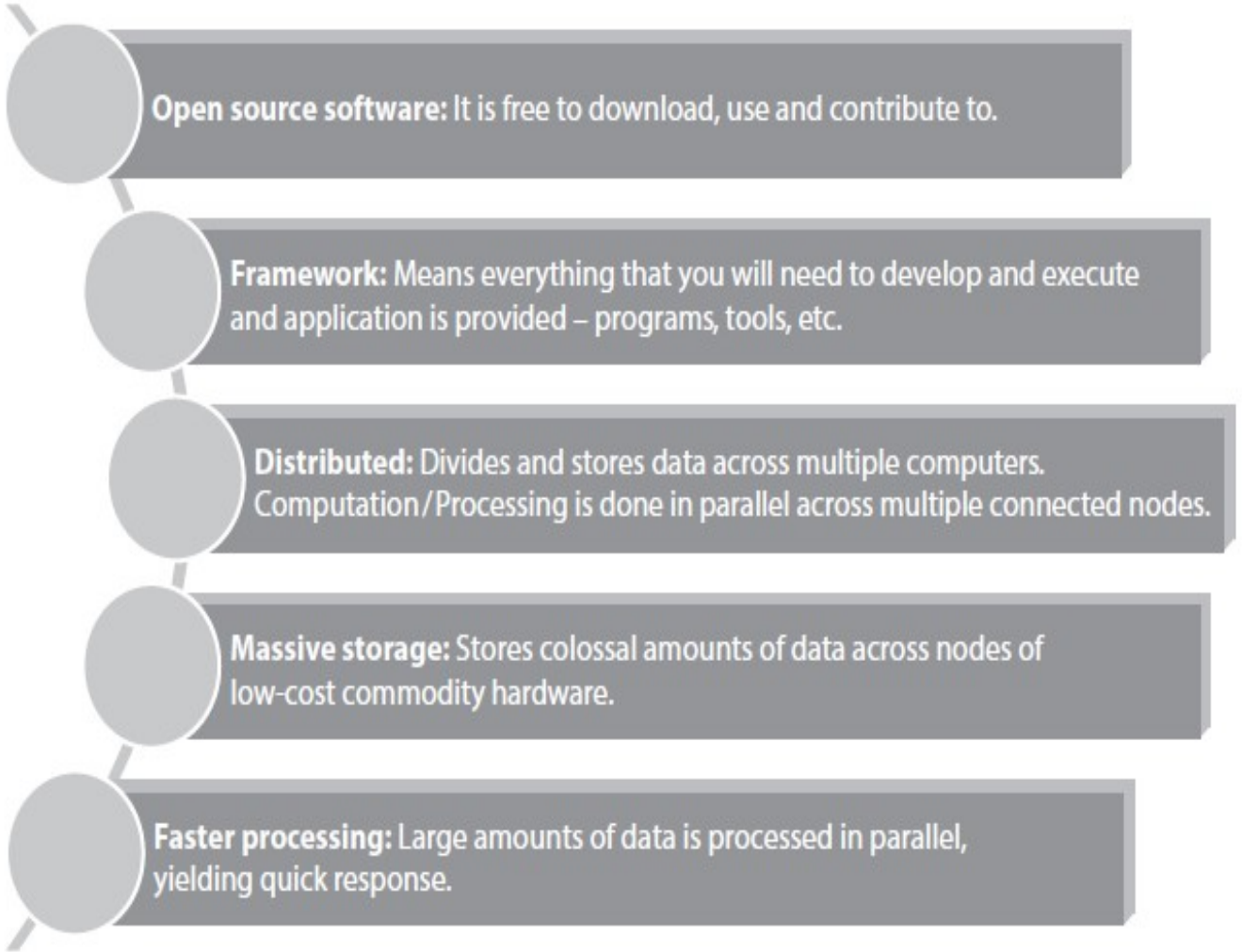
History of Hadoop

History of Hadoop

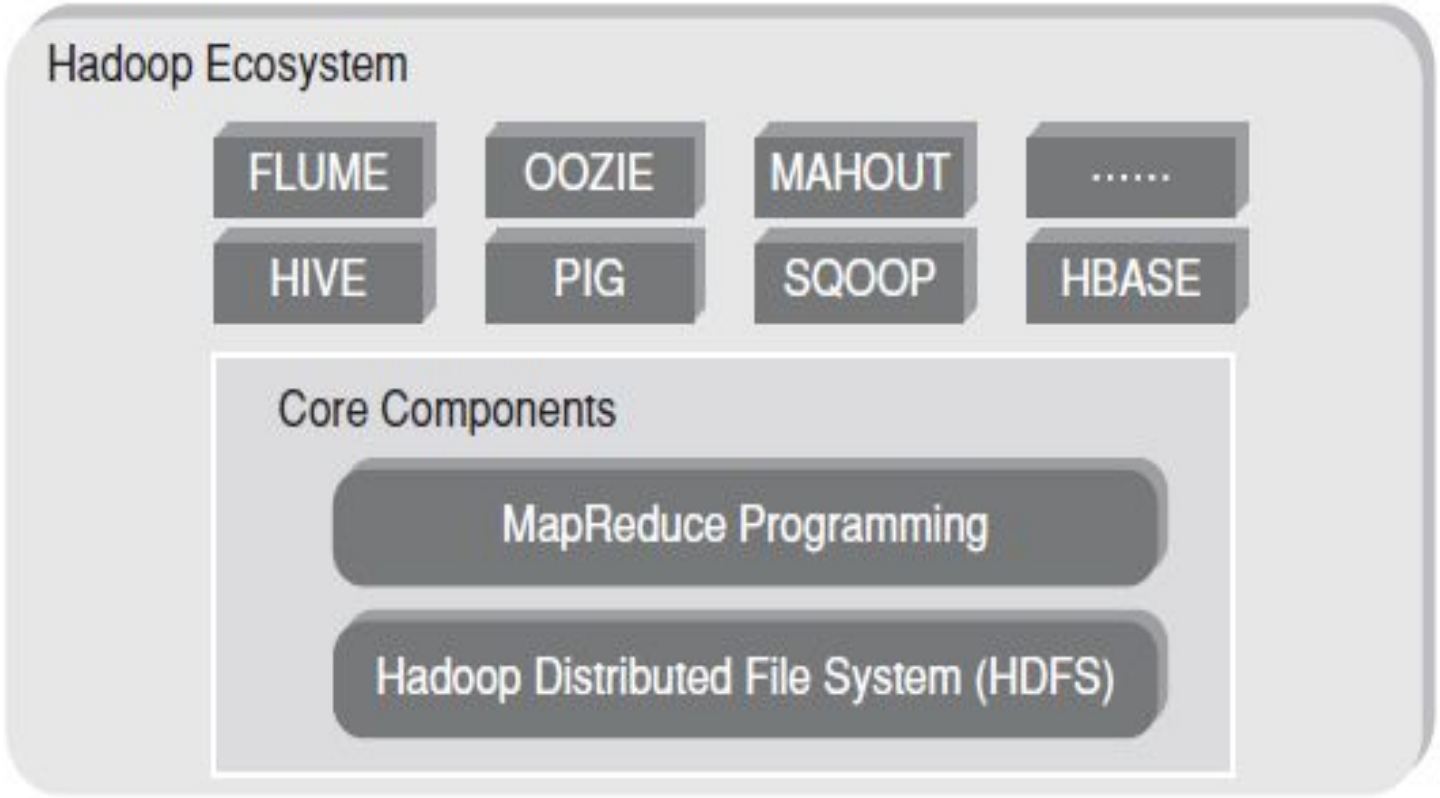


Hadoop Overview

Key Aspects of Hadoop



Hadoop Components



Hadoop Components

Hadoop Core Components:

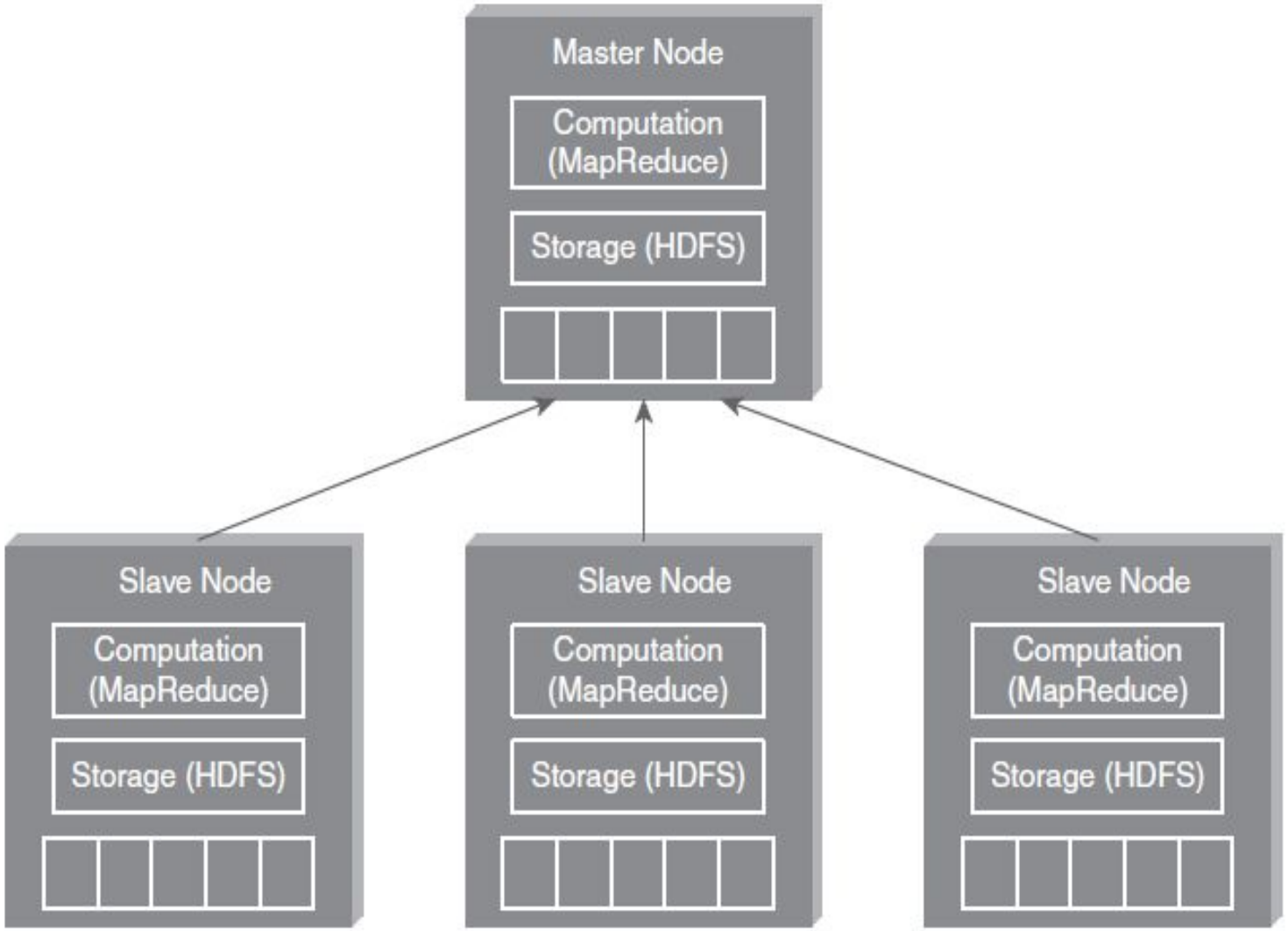
HDFS:

(a) Storage component.
(b) Distributes data across several nodes.
(c) Natively redundant.

MapReduce:

(a) Computational framework.
(b) Splits a task across multiple nodes.
(c) Processes data in parallel.

Hadoop High Level Architecture



Hadoop Architecture

- Hadoop Distributed File System follows the **master-slave architecture**.
- Each cluster comprises a **single master node** and **multiple slave nodes**.
- Internally the files get divided into one or more **blocks**, and each block is stored on different slave machines depending on the **replication factor.**
- The master node stores and manages the file system namespace, that is information about blocks of files like block locations, permissions, etc.
- The slave nodes store data blocks of files.
- The Master node is the NameNode and DataNodes are the slave nodes.

Use case for Hadoop

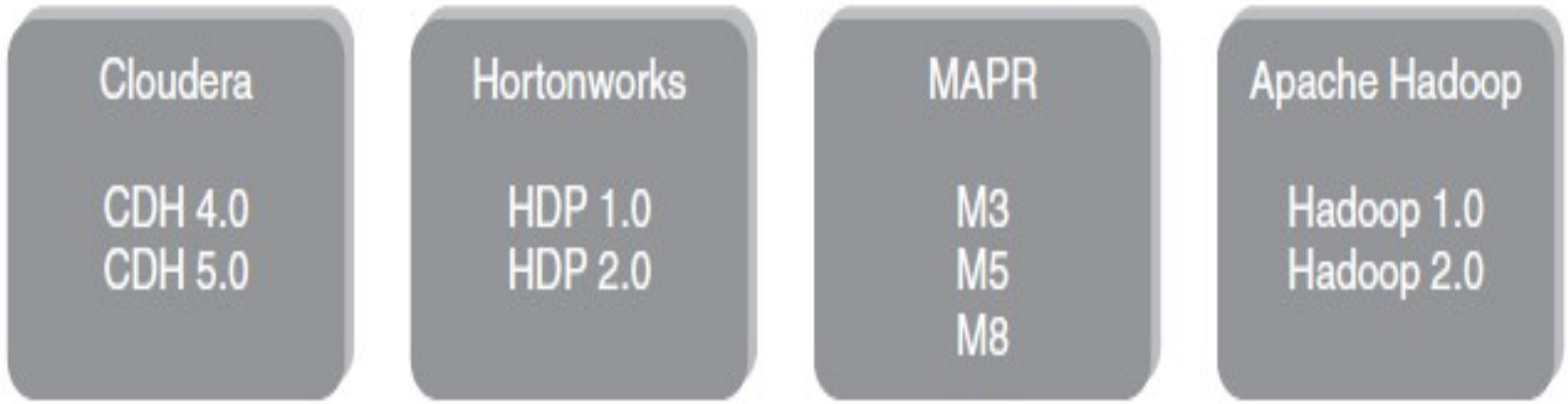
ClickStream Data Analysis

ClickStream data (mouse clicks) helps you to understand the purchasing behavior of customers. ClickStream analysis helps online marketers to optimize their product web pages, promotional content, etc. to improve their business.

ClickStream Data Analysis using Hadoop – Key Benefits		
Joins ClickStream data with CRM and sales data.	Stores years of data without much incremental cost.	Hive or Pig Script to analyze data.

Hadoop Distributors

Hadoop Distributors



HDFS
(HADOOP DISTRIBUTED FILE SYSTEM)

Hadoop Distributed File System

1. Storage component of Hadoop.

2. Distributed File System.

3. Modeled after Google File System.

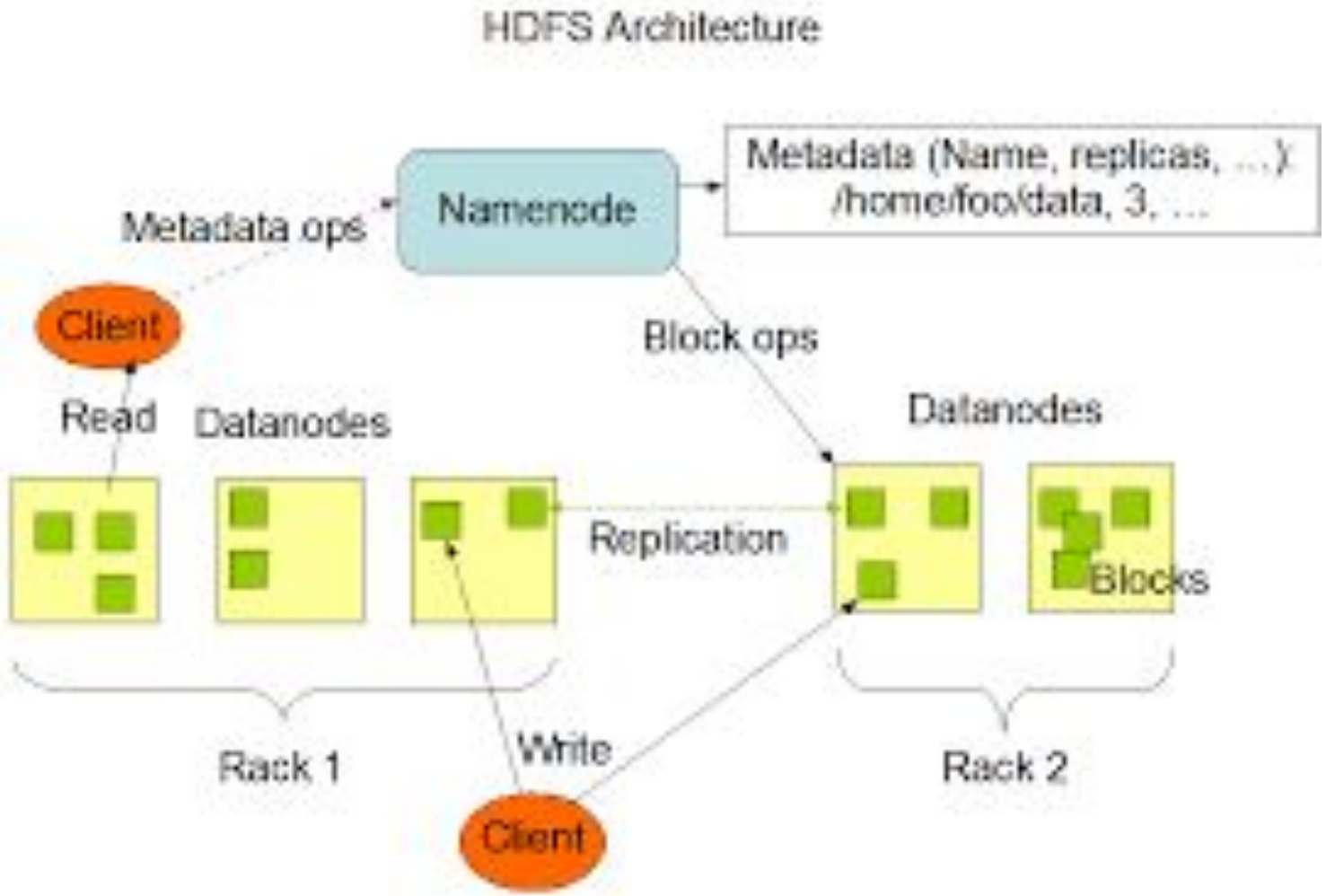
4. Optimized for high throughput (HDFS leverages large block size and moves computation where data is stored).

5. You can replicate a file for a configured number of times, which is tolerant in terms of both software and hardware.

6. Re-replicates data blocks automatically on nodes that have failed.

7. You can realize the power of HDFS when you perform read or write on large files (gigabytes and larger).

8. Sits on top of native file system such as ext3 and ext4, which is described


HDFS Architecture
Metadata (Name, replicas, ...)
/home/foo/data, 3, ...
Namenode
Metadata ops
Client
Block ops
Read
Datanodes
Datanodes
Replication
Blocks
Rack 1
Write
Rack 2
Client

HDFS Daemons

NameNode:

- Single NameNode per cluster.
- Keeps the metadata details
- Manages File related Operations
  - FSImage- File. In which entire file system is stored
  - EditLog- Records every transaction that occurs to file system metadata

DataNode:

- Multiple DataNode per cluster
- During pipeline Read/Write datanodes communicate with each other
- Sends “heartbeat” message to NameNode to ensure the connectivity between them.

SecondaryNameNode:

- Takes the snapshot of the HDFS metadata at intervals specified in the Hadoop configuration.
- In case of failure of the NameNode, the Secondary NameNode can be configured manually to bring up the cluster.

What is HDFS NameNode?

- NameNode is the centerpiece of the Hadoop Distributed File System. It maintains and manages the **file system namespace** and provides the right access permission to the clients.

- The NameNode stores information about blocks locations, permissions, etc. on the local disk in the form of two files:

  - **Fsimage:** Fsimage stands for File System image. It contains the complete namespace of the Hadoop file system since the NameNode creation.

  - **Edit log:** It contains all the recent changes performed to the file system namespace to the most recent Fsimage.

Functions of HDFS NameNode

1. It executes the file system namespace operations like opening, renaming, and closing files and directories.

2. NameNode manages and maintains the DataNodes.

3. It determines the mapping of blocks of a file to DataNodes.

4. NameNode records each change made to the file system namespace.

5. It keeps the locations of each block of a file.

6. NameNode takes care of the replication factor of all the blocks.

7. NameNode receives heartbeat and block reports from all DataNodes that ensure DataNode is alive.

8. If the DataNode fails, the NameNode chooses new DataNodes for new replicas.

What is HDFS DataNode?

- DataNodes are the slave nodes in Hadoop HDFS. DataNodes are **inexpensive commodity hardware**. They store blocks of a file.

- Functions of DataNode

1. DataNode is responsible for serving the client read/write requests.

2. Based on the instruction from the NameNode, DataNodes performs block creation, replication, and deletion.

3. DataNodes send a heartbeat to NameNode to report the health of HDFS.

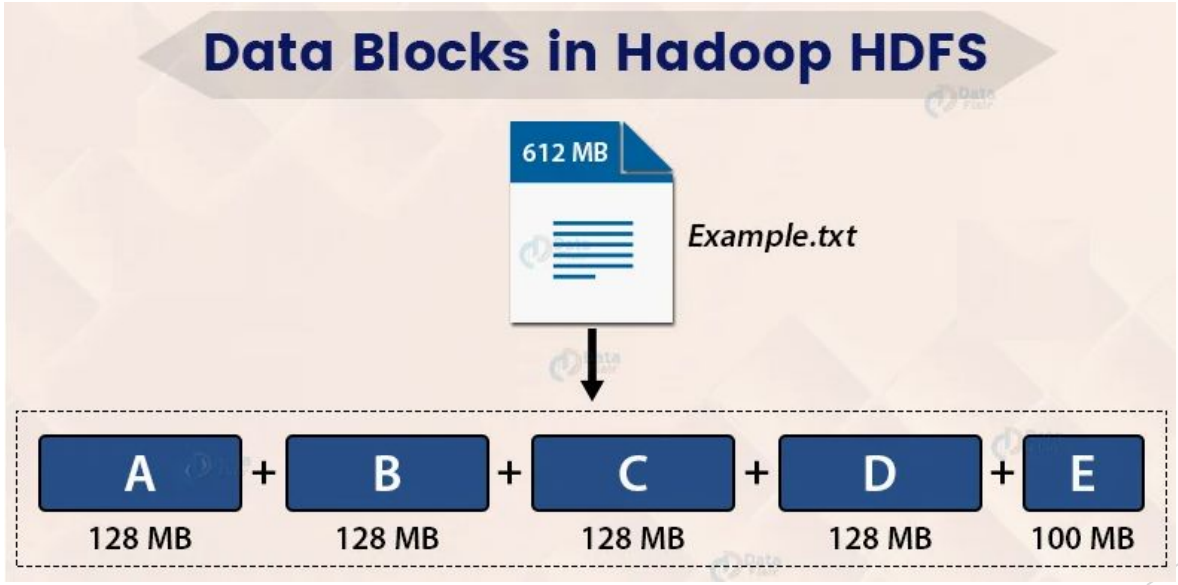
4. DataNodes also sends block reports to NameNode to report the list of blocks it contains.

What is secondary NameNode?

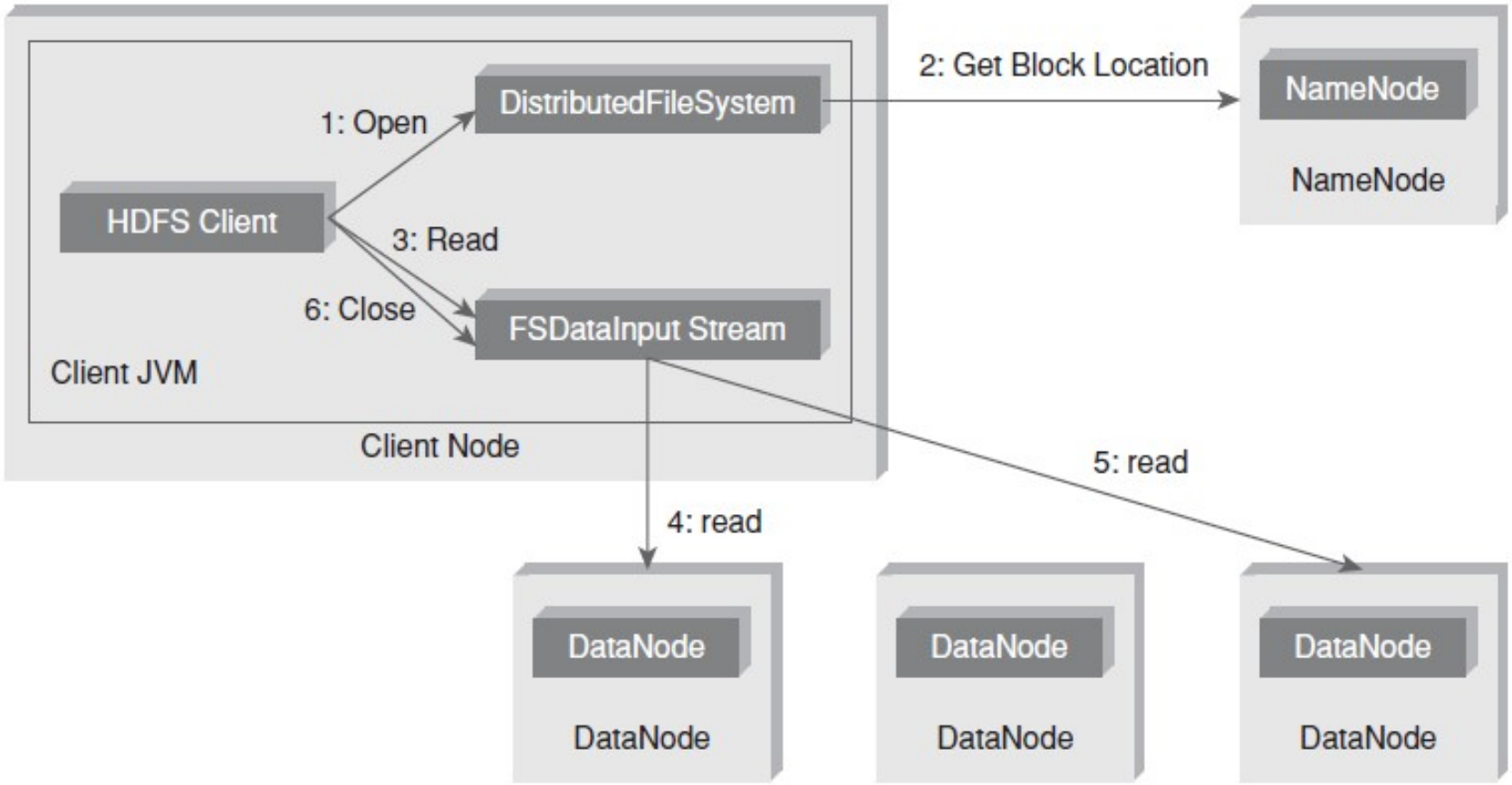
- Apart from DataNode and NameNode, there is another daemon called the **secondary NameNode**. Secondary NameNode works as a helper node to primary NameNode but doesn't replace primary NameNode.
- When the NameNode starts, the NameNode merges the Fsimage and edit logs file to restore the current file system namespace. Since the NameNode runs continuously for a long time without any restart, the size of edit logs becomes too large. This will result in a long restart time for NameNode.
- Secondary NameNode solves this issue.
- Secondary NameNode downloads the Fsimage file and edit logs file from NameNode.
- It periodically applies edit logs to Fsimage and refreshes the edit logs. The updated Fsimage is then sent to the NameNode so that NameNode doesn't have to re-apply the edit log records during its restart. This keeps the edit log size small and reduces the NameNode restart time.
- If the NameNode fails, the last save Fsimage on the secondary NameNode can be used to recover file system metadata. The secondary NameNode performs regular checkpoints in HDFS.

What are blocks in HDFS Architecture

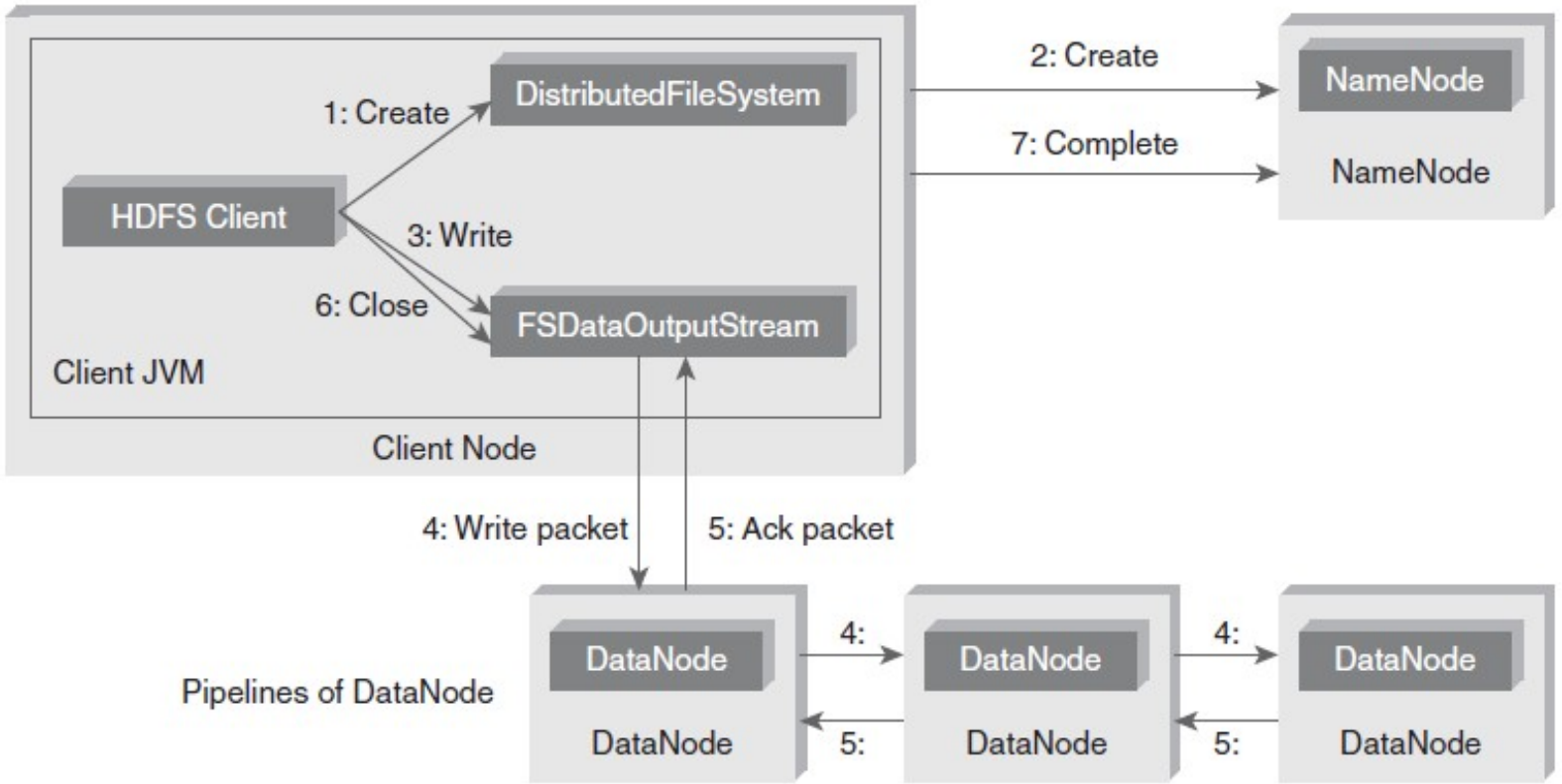
- Internally, HDFS split the file into block-sized chunks called a block. The size of the block is **128 Mb** by default. One can configure the block size as per the requirement.
- For example, if there is a file of size 612 Mb, then HDFS will create four blocks of size 128 Mb and one block of size 100 Mb.
- The file of a smaller size does not occupy the full block size space in the disk.
- For example, the file of size 2 Mb will occupy only 2 Mb space in the disk.
- The user doesn't have any control over the location of the blocks.



Anatomy of File Read

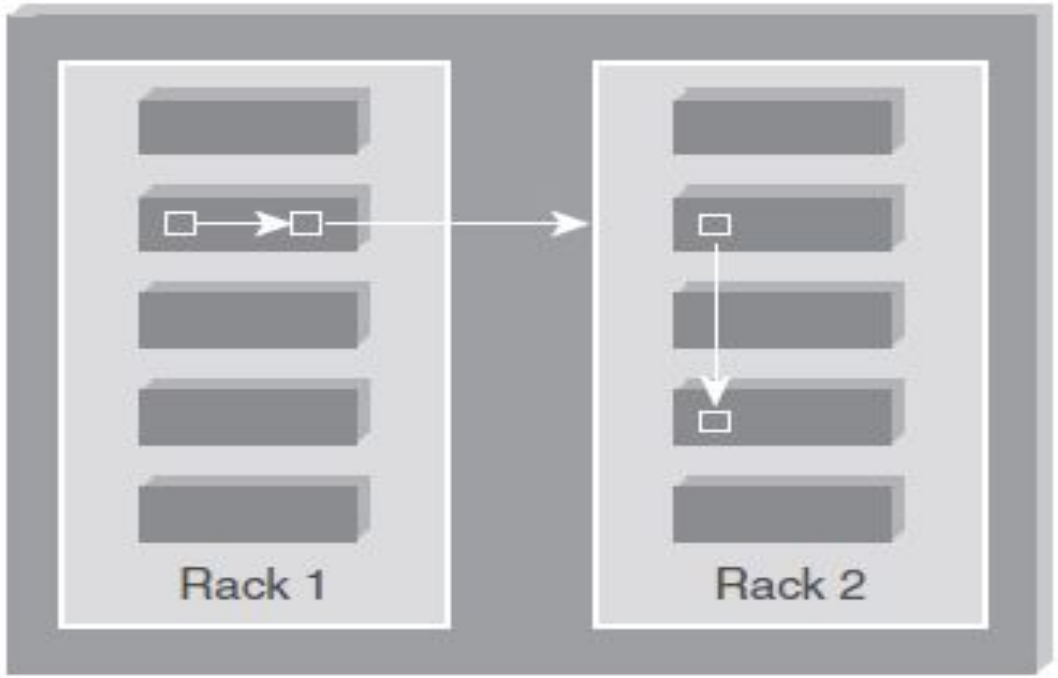


Anatomy of File Write



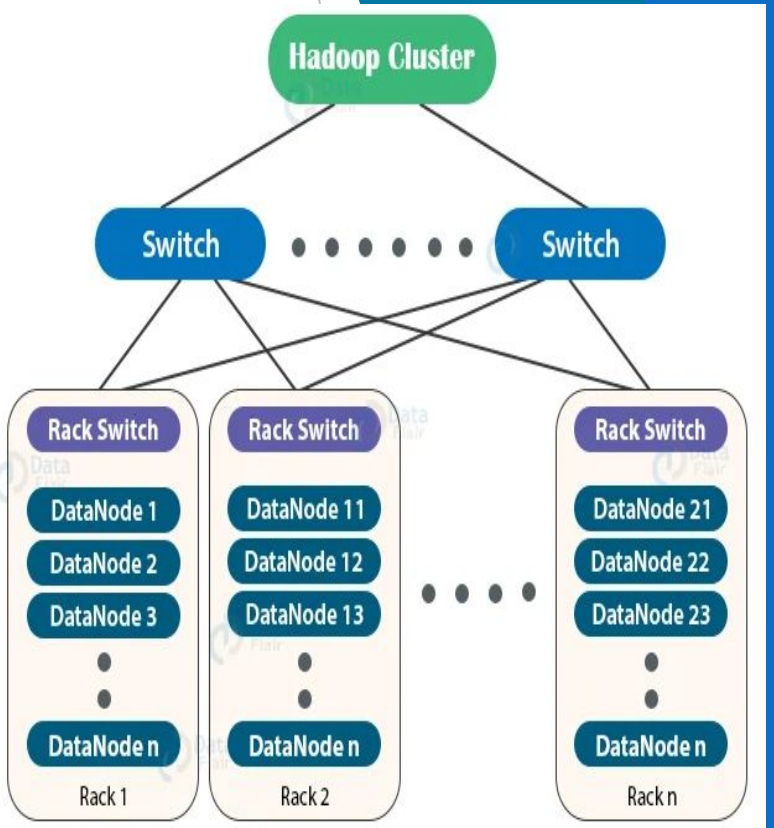
Replica Placement Strategy

As per the Hadoop Replica Placement Strategy, first replica is placed on the same node as the client. Then it places second replica on a node that is present on different rack. It places the third replica on the same rack as second, but on a different node in the rack. Once replica locations have been set, a pipeline is built. This strategy provides good reliability.



Rack awareness in HDFS

- In a large Hadoop cluster, there are multiple racks. Each rack consists of DataNodes.
- Communication between the DataNodes on the same rack is more efficient as compared to the communication between DataNodes residing on different racks.
- To reduce the network traffic during file **read/write**, NameNode chooses the closest DataNode for serving the client read/write request.
- NameNode maintains **rack ids** of each DataNode to achieve this rack information. This concept of choosing the closest DataNode based on the rack information is known as **Rack Awareness**.



Why Rackawareness?

The reasons for the Rack Awareness in Hadoop are:

1. To reduce the network traffic while file read/write, which improves the cluster performance.

2. To achieve **fault tolerance**, even when the rack goes down (discussed later in this article).

3. Achieve high availability of data so that data is available even in unfavorable conditions.

4. To reduce the latency, that is, to make the file read/write operations done with lower delay.

Working With HDFS Command

- To use HDFS commands, start the Hadoop services using the following command:

  sbin/start-all.sh OR sbin/start-dfs.sh

- To check if Hadoop is up and running:

  jps

Working with HDFS Commands

1. ***hadoop fs -ls /***

To get the list of directories and files at the root of HDFS

2. ***hadoop fs -ls -R /***

To get the list of complete directories and files of HDFS

3. ***hadoop fs -mkdir /sample***

To create a directory (say, sample) in HDFS.

4. ***hadoop fs -put /root/sample/test.txt /sample/test.txt***

To copy a file from local file system to HDFS.

5. hadoop fs -get /sample/test.txt /root/sample/testsample.txt

To copy a file from HDFS to local file system.

Working with HDFS Commands

6. hadoop fs -copyFromLocal /root/sample/test.txt /sample/test.txt

To copy a file from local file system to HDFS using copyFromLocal command.

7. hadoop fs -copyToLocal /sample/test.txt /root/sample/testsample.txt

To copy a file from HDFS to local file system using copyToLocal command

8. ***hadoop fs -cat /sample/test.txt***

To display the contents of an HDFS file on console

9. ***hadoop fs -cp /sample/test.txt /sample1***

To copy a file from one directory to another

10. ***hadoop fs -rm-r /sample1***

To remove a directory from HDFS

Command Execution Example

1. hadoop version

2. M

```
File  Edit  View  Search  Terminal  Help
dataflair@admin1-All-Series:~$ hadoop fs -mkdir /newDataFlair
dataflair@admin1-All-Series:~$ hadoop fs -ls /
Found 3 items
drwxr-xr-x   - dataflair supergroup          0 2020-01-29 10:38 /DataFlair
drwxr-xr-x   - dataflair supergroup          0 2020-01-29 10:39 /dataflair
drwxr-xr-x   - dataflair supergroup          0 2020-01-29 10:41 /newDataFlair
dataflair@admin1-All-Series:~$
```

HDFS DFS Commands

COMMAND	DESCRIPTION
`-ls`	List files with permissions and other details
`-mkdir`	Creates a directory named path in HDFS
`-rm`	To Remove File or a Directory
`-rmr`	Removes the file that identified by path / Folder and subfolders
`-rmdir`	Delete a directory
`-put`	Upload a file / Folder from the local disk to HDFS
`-cat`	Display the contents for a file
`-du`	Shows the size of the file on hdfs.
`-dus`	Directory/file of total size
`-get`	Store file / Folder from HDFS to local file
`-getmerge`	Merge Multiple Files in an HDFS
`-count`	Count number of directory, number of files and file size
`-setrep`	Changes the replication factor of a file

HDFS DFS Commands

`-mv`	HDFS Command to move files from source to destination
`-moveFromLocal`	Move file / Folder from local disk to HDFS
`-moveToLocal`	Move a File to HDFS from Local
`-cp`	Copy files from source to destination
`-tail`	Displays last kilobyte of the file
`-touch`	create, change and modify timestamps of a file
`-touchz`	Create a new file on HDFS with size 0 bytes
`-appendToFile`	Appends the content to the file which is present on HDF
`-copyFromLocal`	Copy file from local file system
`-copyToLocal`	Copy files from HDFS to local file system
`-usage`	Return the Help for Individual Command
`-checksum`	Returns the checksum information of a file
`-chgrp`	Change group association of files/change the group of a file or a path
`-chmod`	Change the permissions of a file

HDFS DFS Commands

`-chmod`	Change the permissions of a file
`-chown`	change the owner and group of a file
`-df`	Displays free space
`-head`	Displays first kilobyte of the file
`-Create Snapshots`	Create a snapshot of a snapshottable directory
`-Delete Snapshots`	Delete a snapshot of from a snapshottable directory
`-Rename Snapshots`	Rename a snapshot
`-expunge`	create new checkpoint
`-Stat`	Print statistics about the file/directory
`-truncate`	Truncate all files that match the specified file pattern to the specified length
`-find`	Find File Size in HDFS

Special Features of HDFS

Data Replication: There is absolutely no need for a client application to track all blocks. It directs the client to the nearest replica to ensure high performance.

Data Pipeline: A client application writes a block to the first DataNode in the pipeline. Then this DataNode takes over and forwards the data to the next node in the pipeline. This process continues for all the data blocks, and subsequently all the replicas are written to the disk.

MANAGING RESOURCES AND APPLICATIONS WITH HADOOP - YARN

(YET ANOTHER RESOURCE NEGOTIATOR)

Limitations of Hadoop 1.0 Architecture

1. Single NameNode is responsible for managing entire namespace for Hadoop Cluster.

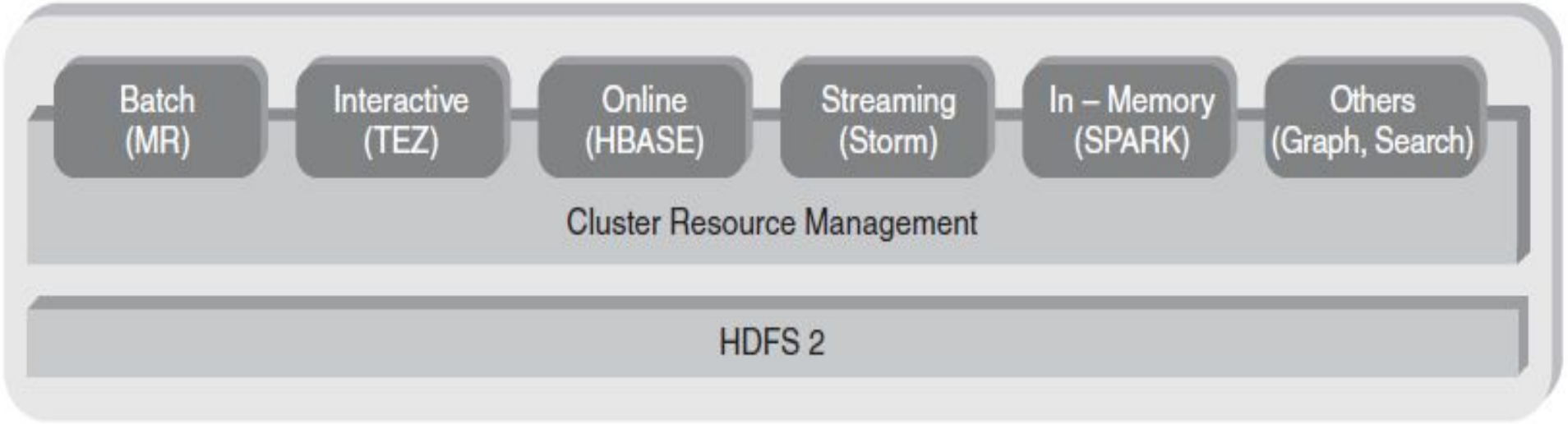
2. It has a restricted processing model which is suitable for batch-oriented MapReduce jobs.

3. Hadoop MapReduce is not suitable for interactive analysis.

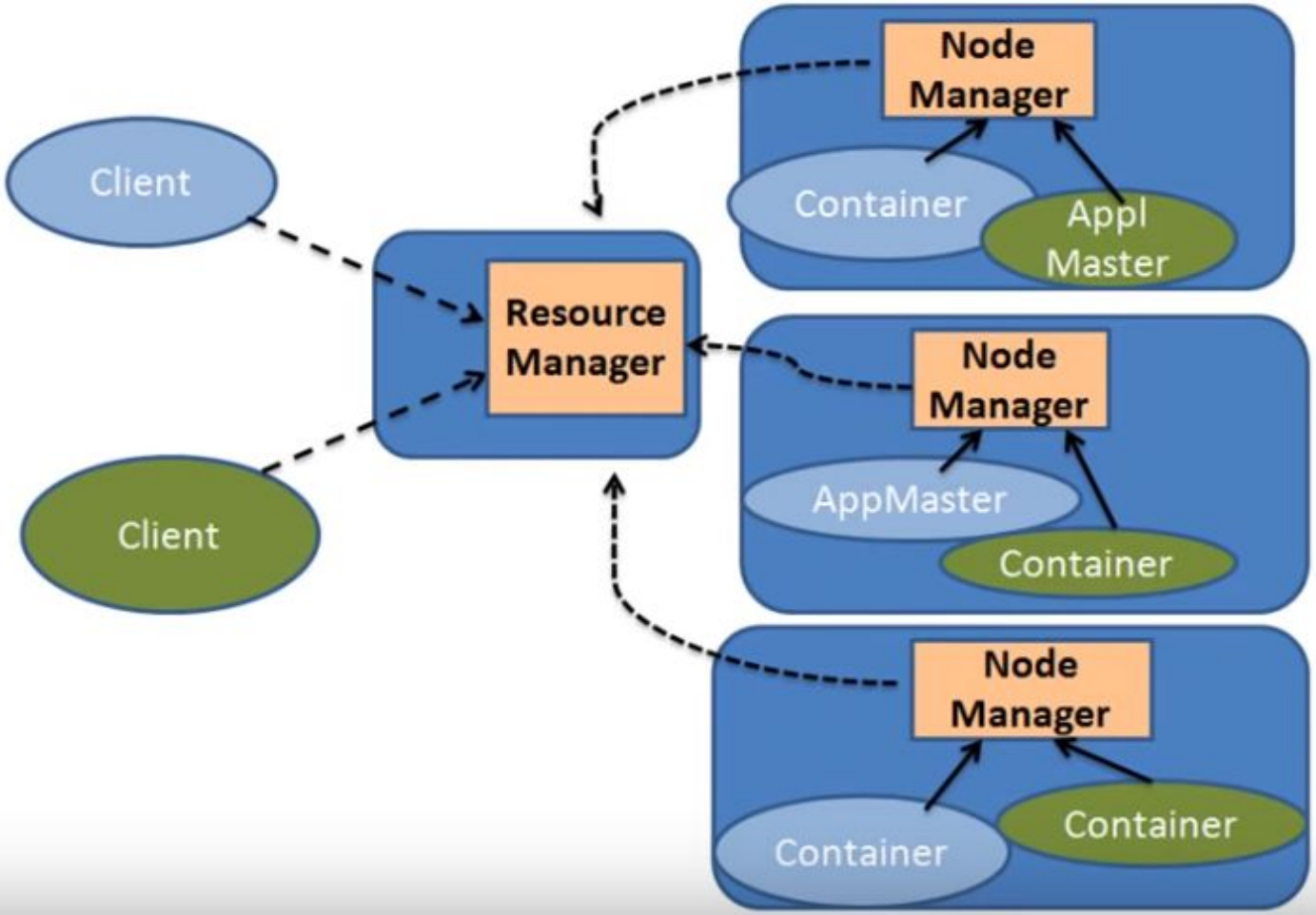
4. Hadoop 1.0 is not suitable for machine learning algorithms, graphs, and other memory intensive algorithms.

5. **MapReduce** is responsible for **cluster resource management and data processing**.

Hadoop 2 YARN: Taking Hadoop beyond Batch



- YARN stands for "Yet Another Resource Negotiator."
- YARN/MapReduce2 has been introduced in Hadoop 2.0.
- It is a layer that separates the resource management layer and the processing components layer.
- MapReduce2 moves Resource management (like infrastructure to monitor nodes, allocate resources and schedule jobs)into YARN



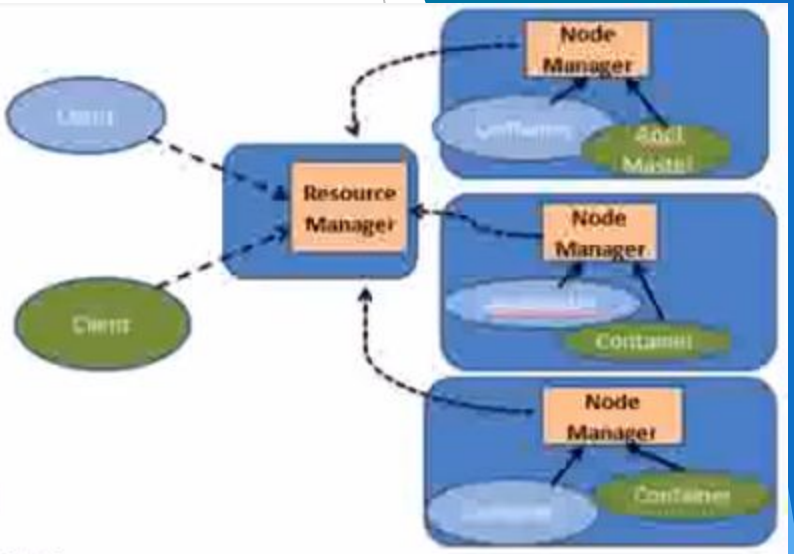
✓Job tracker 1.0 responsibility is now split

- **Resource Manager** manages the resource allocation in the cluster
- **Application master** manages resource needs of individual applications

✓Node Manager is a generalized task tracker

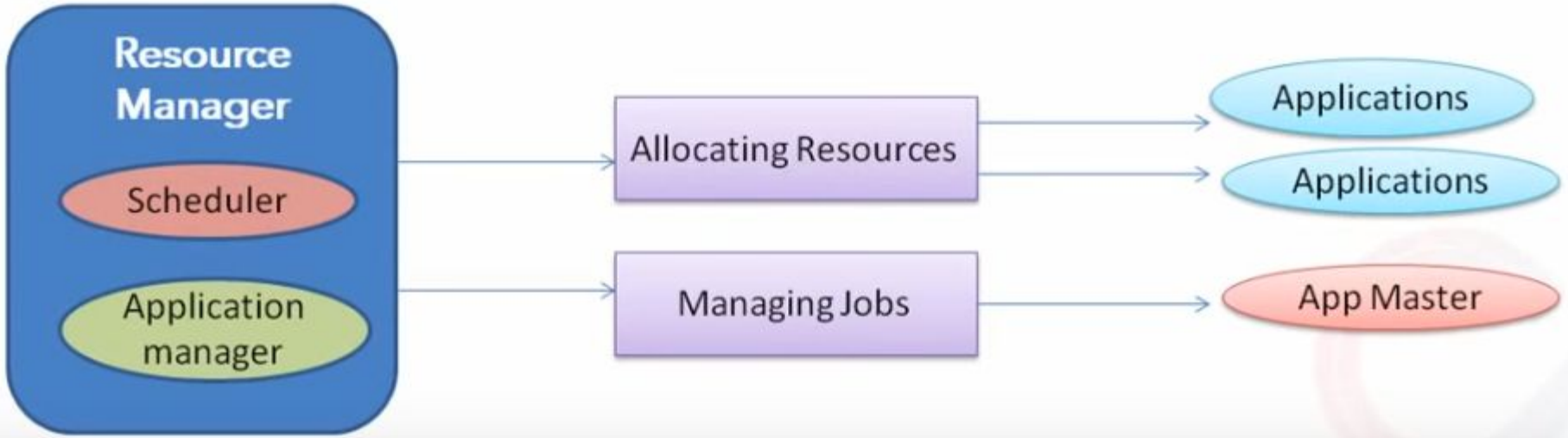
✓A container executes an application specific process

1. **Client**: To submit MapReduce jobs
2. **Resource Manager**: To manage the use of resources across the cluster.
3. **Container** : Name given to a package of resources including RAM, CPU, Network, HDD etc.
4. **Node Manager**: to oversee the containers running on the cluster nodes.
5. **Application Master**: which negotiates with the Resource Manager for resources and runs the application-specific process (Map or Reduce tasks) in those clusters.



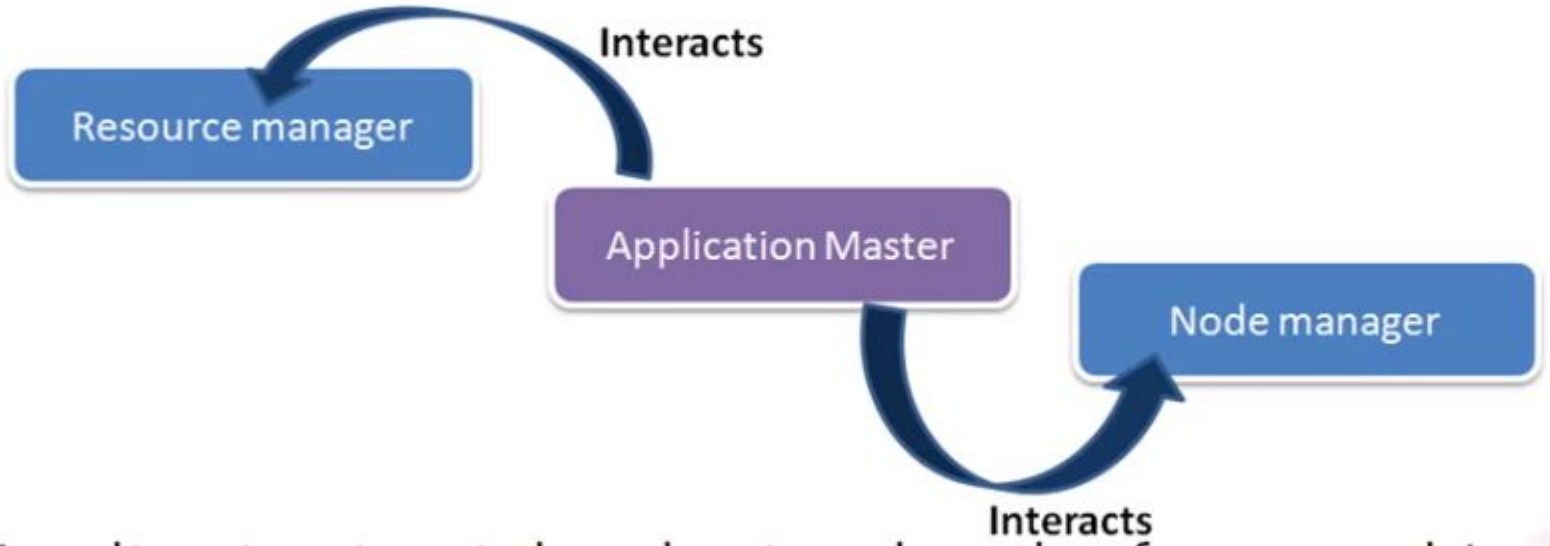
Resource manager

- Two components-scheduler and application manager
- It is a global resource scheduler
- Manages and allocates cluster resources



Application Master

- Manages application life cycle and task scheduling



- Application is a job submitted to the framework(ex.MapReduce job)

- **Node Manager**

Manages single node resource allocations

Per-node agent

- **Container/Slot**

Basic unit of allocation

Ex:Container X= 2GB,1 CPU

Fine grained resource

Hadoop 2 YARN: Taking Hadoop beyond Batch

The fundamental idea behind this architecture is splitting the JobTracker responsibility of resource management and Job Scheduling/Monitoring into separate daemons. Daemons that are part of YARN Architecture are described below.

A Global ResourceManager: Its main responsibility is to distribute resources among various applications in the system. It has two main components:

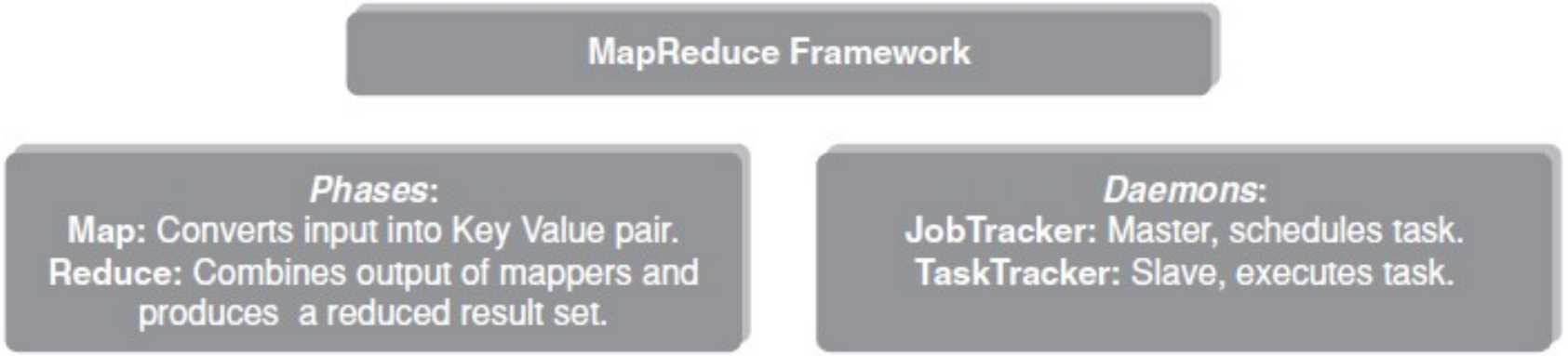
NodeManager: This is a per-machine slave daemon. NodeManager responsibility is launching the application containers for application execution. NodeManager monitors the resource usage such as memory, CPU, disk, network, etc. It then reports the usage of resources to the global ResourceManager.

Per-application ApplicationMaster: This is an application-specific entity. Its responsibility is to negotiate required resources for execution from the ResourceManager. It works along with the NodeManager for executing and monitoring component tasks.

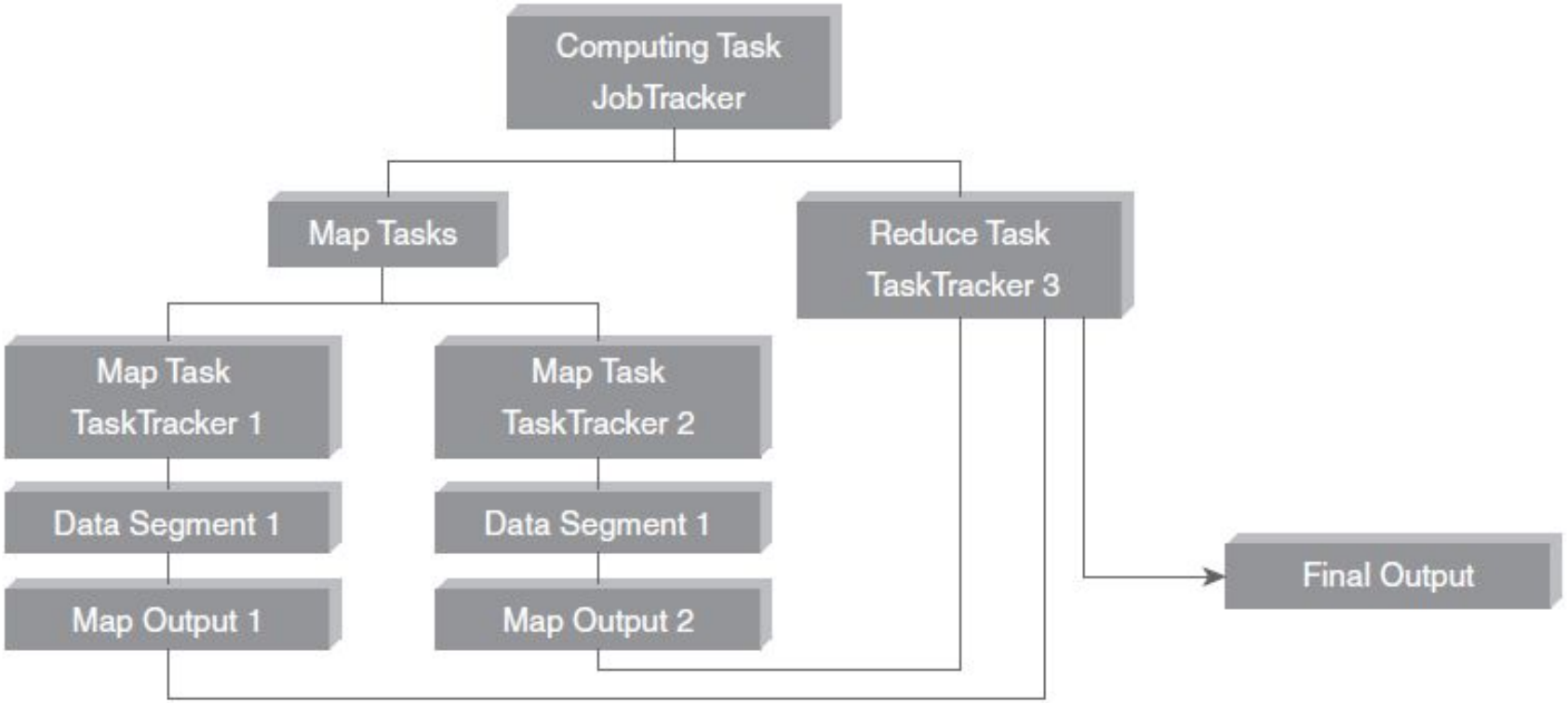
Processing with Hadoop

What is MapReduce Programming?

MapReduce Programming is a software framework. MapReduce Programming helps you to process massive amounts of data in parallel.



How MapReduce Programming Works



Introduction

In MapReduce Programming, Jobs (Applications) are split into a set of map tasks and reduce tasks. Then these tasks are executed in a distributed fashion on Hadoop cluster.

Each task processes small subset of data that has been assigned to it. This way, Hadoop distributes the load across the cluster.

MapReduce job takes a set of files that is stored in HDFS (Hadoop Distributed File System) as input.

Mapper

Mapper

A mapper maps the input key–value pairs into a set of intermediate key–value pairs. Maps are individual tasks that have the responsibility of transforming input records into intermediate key–value pairs.

Mapper Consists of following phases:

- **RecordReader**
- **Map**
- **Combiner**
- **Partitioner**

Reducer

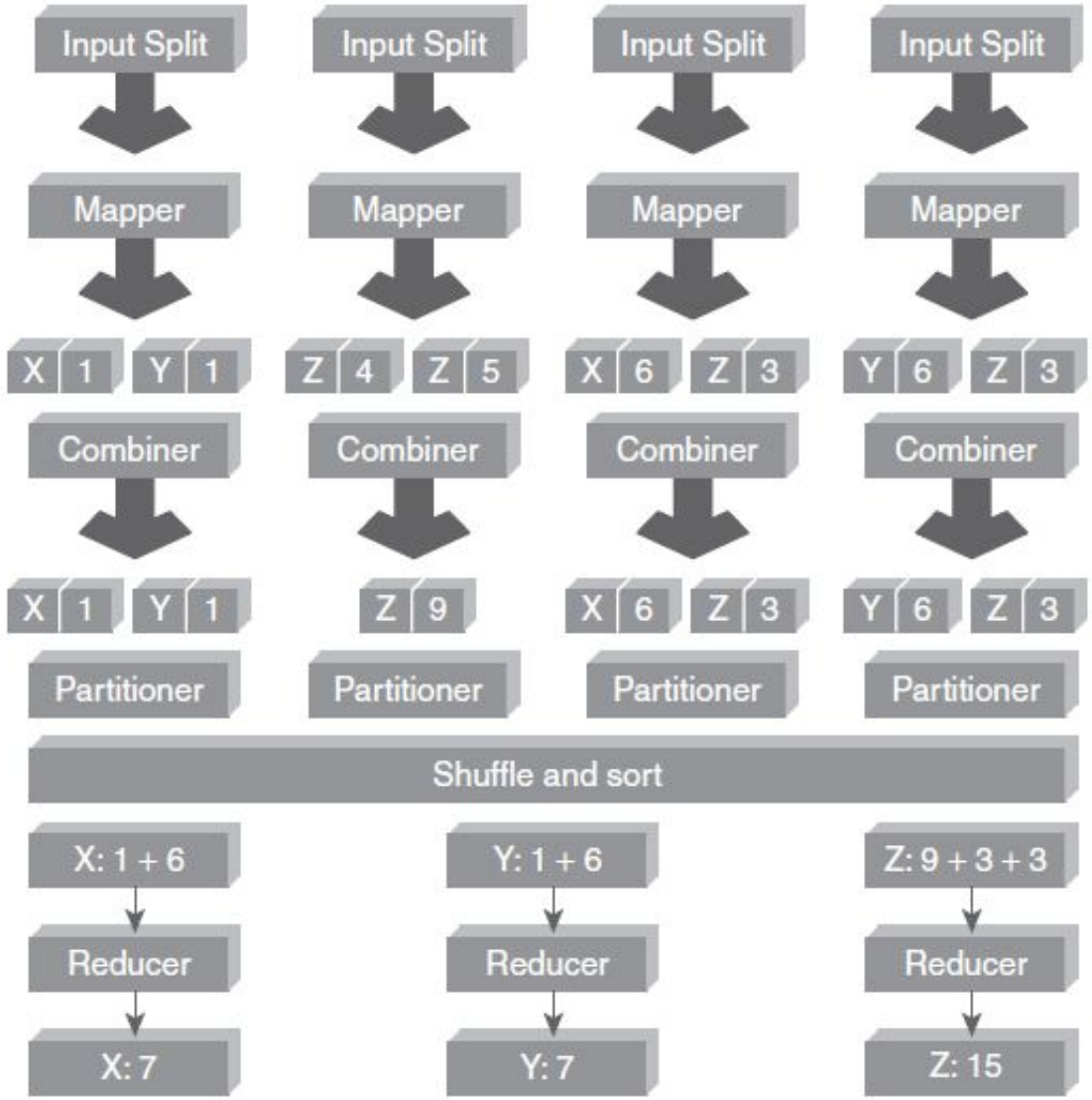
Reducer

The primary chore of the Reducer is to reduce a set of intermediate values (the ones that share a common key) to a smaller set of values.

The Reducer has three primary phases:
Shuffle and Sort,
Reduce, and
Output Format.

The chores of Mapper, Combiner, Partitioner, and Reducer

The chores of Mapper, Combiner, Partitioner, and Reducer



Combiner

Combiner

It is an optimization technique for MapReduce Job. Generally, the reducer class is set to be the combiner class. The difference between combiner class and reducer class is as follows:

- Output generated by combiner is intermediate data and it is passed to the reducer.
- Output of the reducer is passed to the output file on disk.

Advantages of Combiner

- Hadoop Combiner reduces the time taken for data transfer between mapper and reducer.
- It decreases the amount of data that needed to be processed by the reducer.
- MapReduce Combiner plays a key role in reducing network congestion.
- MapReduce combiner improves the overall performance of the reducer by summarizing the output of Mapper.

In the driver program, set the partitioner class as:

conf.setCombinerClass(WordCounterRed.class);

Partitioner

Partitioner

The partitioning phase happens after map phase and before reduce phase. Usually the number of partitions are equal to the number of reducers. The default partitioner is hash partitioner.

```
Public class WordCountPartitioner extends Partitioner<Text, IntWritable>
{
Public int getPartition(Text key, IntWritable value, int numPartitions)
{
  String word= key.toString()
  char alphabet = word.toUpperCase().charAt(0);
  int partitionNumber=0;
  switch(alphabet)
{
  case 'A' : partitionNumber=1;
             break;
   case 'B' :partitionNumber=2;
             break;
:
:
:
:
case 'Z': parttionNumber=0;
          break;
}
return partitionNumber;
}
```

In the driver program, set the partitioner class as shown below

```
conf.setNumReduceTasks(27);
conf. setPartitionerClass(WordCountPartitioner.class);
```

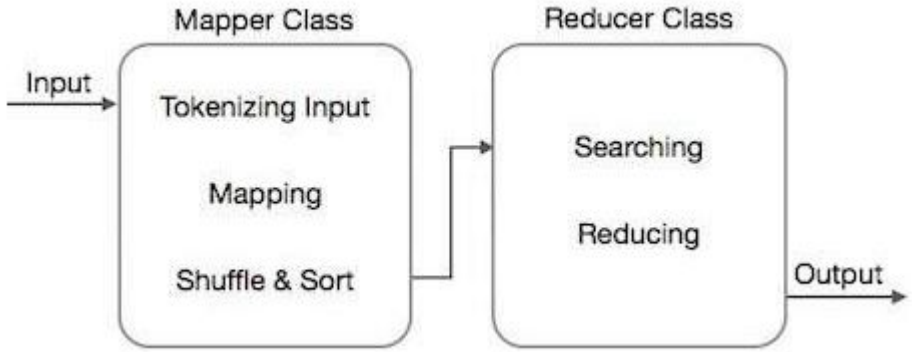
Conclusion

- Hadoop Partitioner allows even distribution of the map output over the reducer.
- In Partitioner, partitioning of map output take place on the basis of the key and sorted

Searching and Sorting Demo

MapReduce Algorithms

Mapper class takes the input, tokenizes it, maps and sorts it. The output of Mapper class is used as input by Reducer class, which in turn searches matching pairs and reduces them.



MapReduce implements various mathematical algorithms to divide a task into small parts and assign them to multiple systems. In technical terms, MapReduce algorithm helps in sending the Map & Reduce tasks to appropriate servers in a cluster.

These mathematical algorithms may include the following −

- Sorting
- Searching
- Indexing
- TF-IDF

Sorting

Sorting is one of the basic MapReduce algorithms to process and analyze data. MapReduce implements sorting algorithm to automatically sort the output key-value pairs from the mapper by their keys.

- Sorting methods are implemented in the mapper class itself.
- In the Shuffle and Sort phase, after tokenizing the values in the mapper class, the **Context** class (user-defined class) collects the matching valued keys as a collection.
- To collect similar key-value pairs (intermediate keys), the Mapper class takes the help of **RawComparator** class to sort the key-value pairs.
- The set of intermediate key-value pairs for a given Reducer is automatically sorted by Hadoop to form key-values (K2, {V2, V2, …}) before they are presented to the Reducer.

Searching

Searching plays an important role in MapReduce algorithm. It helps in the combiner phase (optional) and in the Reducer phase. Let us try to understand how Searching works with the help of an example.

Example

The following example shows how MapReduce employs Searching algorithm to find out the details of the employee who draws the highest salary in a given employee dataset.

- Let us assume we have employee data in four different files – A, B, C, and D. Let us also assume there are duplicate employee records in all four files because of importing the employee data from all database tables repeatedly. See the following illustration.

name, salary	name, salary	name, salary	name, salary
satish, 26000	gopal, 50000	satish, 26000	satish, 26000
Krishna, 25000	Krishna, 25000	kiran, 45000	Krishna, 25000
Satishk, 15000	Satishk, 15000	Satishk, 15000	manisha, 45000
Raju, 10000	Raju, 10000	Raju, 10000	Raju, 10000

- **The Map phase** processes each input file and provides the employee data in key-value pairs (<k, v> : <emp name, salary>). See the following illustration.

<satish, 26000>	<gopal, 50000>	<satish, 26000>	<satish, 26000>
<Krishna, 25000>	<Krishna, 25000>	<kiran, 45000>	<Krishna, 25000>
<Satishk, 15000>	<Satishk, 15000>	<Satishk, 15000>	<manisha, 45000>
<Raju, 10000>	<Raju, 10000>	<Raju, 10000>	<Raju, 10000>

Searching

```
<k: employee name, v: salary>
Max= the salary of an first employee. Treated as max salary

if(v(second employee).salary > Max){
   Max = v(salary);
}

else{
   Continue checking;
}
```

The expected result is as follows –

<satish, 26000>	<gopal, 50000>	<kiran, 45000>	<manisha, 45000>

- **Reducer phase** – Form each file, you will find the highest salaried employee. To avoid redundancy, check all the <k, v> pairs and eliminate duplicate entries, if any. The same algorithm is used in between the four <k, v> pairs, which are coming from four input files. The final output should be as follows –

```
<gopal, 50000>
```

Indexing

Normally indexing is used to point to a particular data and its address. It performs batch indexing on the input files for a particular Mapper.

The indexing technique that is normally used in MapReduce is known as **inverted index.** Search engines like Google and Bing use inverted indexing technique. Let us try to understand how Indexing works with the help of a simple example.

Example

The following text is the input for inverted indexing. Here T[0], T[1], and t[2] are the file names and their content are in double quotes.

```
T[0] = "it is what it is"
T[1] = "what is it"
T[2] = "it is a banana"
```

After applying the Indexing algorithm, we get the following output –

```
"a": {2}
"banana": {2}
"is": {0, 1, 2}
"it": {0, 1, 2}
"what": {0, 1}
```

Here "a": {2} implies the term "a" appears in the T[2] file. Similarly, "is": {0, 1, 2} implies the term "is" appears in the files T[0], T[1], and T[2].

TF-IDF

TF-IDF is a text processing algorithm which is short for Term Frequency − Inverse Document Frequency.

It is one of the common web analysis algorithms. Here, the term 'frequency' refers to the number of times a term appears in a document.

Term Frequency (TF)

It measures how frequently a particular term occurs in a document. It is calculated by the number of times a word appears in a document divided by the total number of words in that document.

```
TF(the) = (Number of times term the ‘the’ appears in a document) / (Total number of terms in the document)
```

Inverse Document Frequency (IDF)

It measures the importance of a term. It is calculated by the number of documents in the text database divided by the number of documents where a specific term appears.

While computing TF, all the terms are considered equally important. That means, TF counts the term frequency for normal words like “is”, “a”, “what”, etc. Thus we need to know the frequent terms while scaling up the rare ones, by computing the following –

```
IDF(the) = log_e(Total number of documents / Number of documents with term ‘the’ in it).
```

TF-IDF

Example

Consider a document containing 1000 words, wherein the word **hive** appears 50 times. The TF for **hive** is then (50 / 1000) = 0.05.

Now, assume we have 10 million documents and the word **hive** appears in 1000 of these. Then, the IDF is calculated as log(10,000,000 / 1,000) = 4.

The TF-IDF weight is the product of these quantities – 0.05 × 4 = 0.20.

Compression

In MapReduce programming, you can compress the MapReduce output file. Compression provides two benefits as follows:

1. Reduces the space to store files.
2. Speeds up data transfer across the network.

You can specify compression format in the Driver Program as shown below:

```
conf.setBoolean("mapred.output.compress",true);
conf.setClass("mapred.output.compression.codec",
GzipCodec.class,CompressionCodec.class);
```

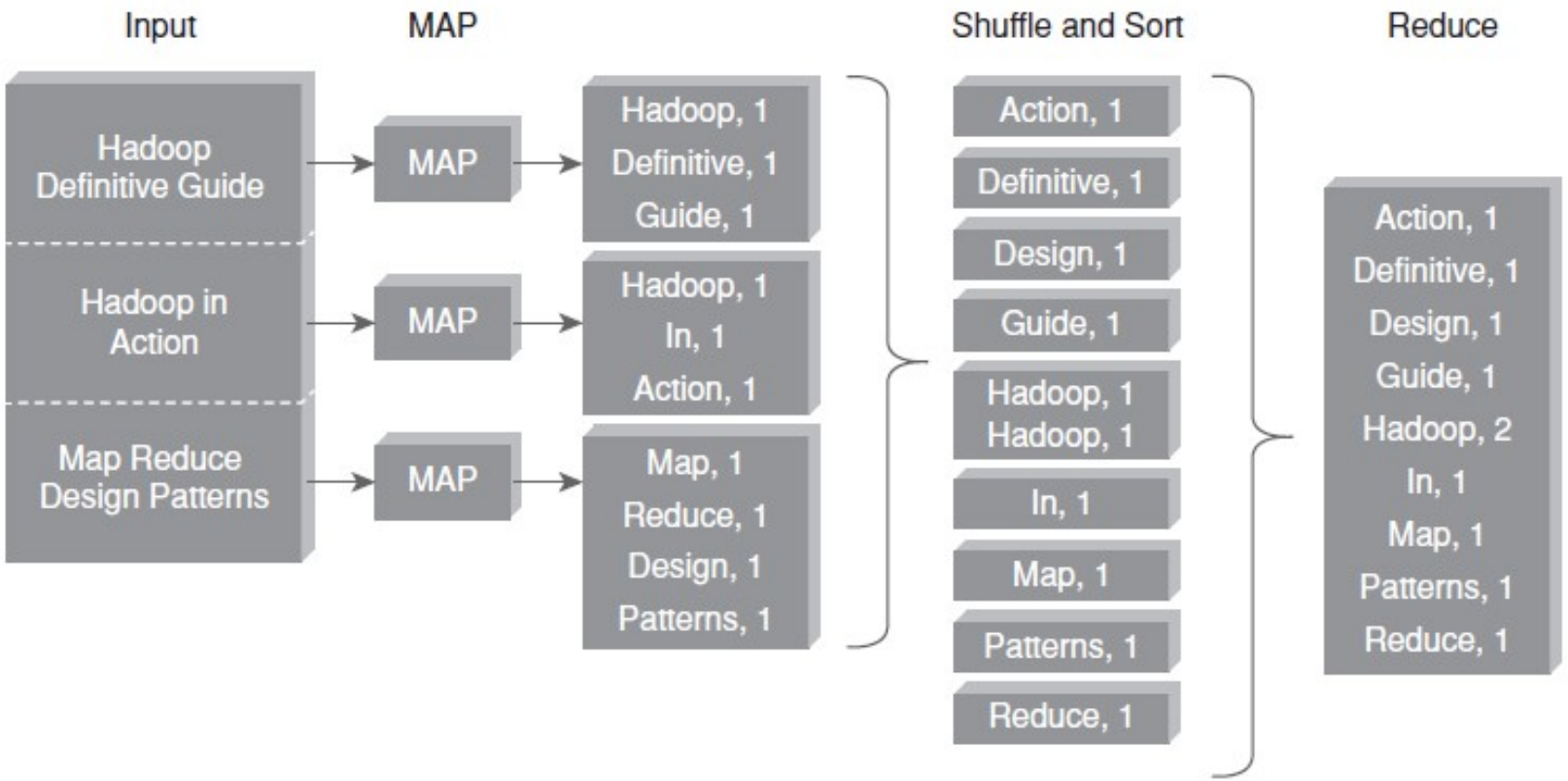
Here, codec is the implementation of a compression and decompression algorithm. GzipCodec is the compression algorithm for gzip. This compresses the output file.

Answer a few questions…

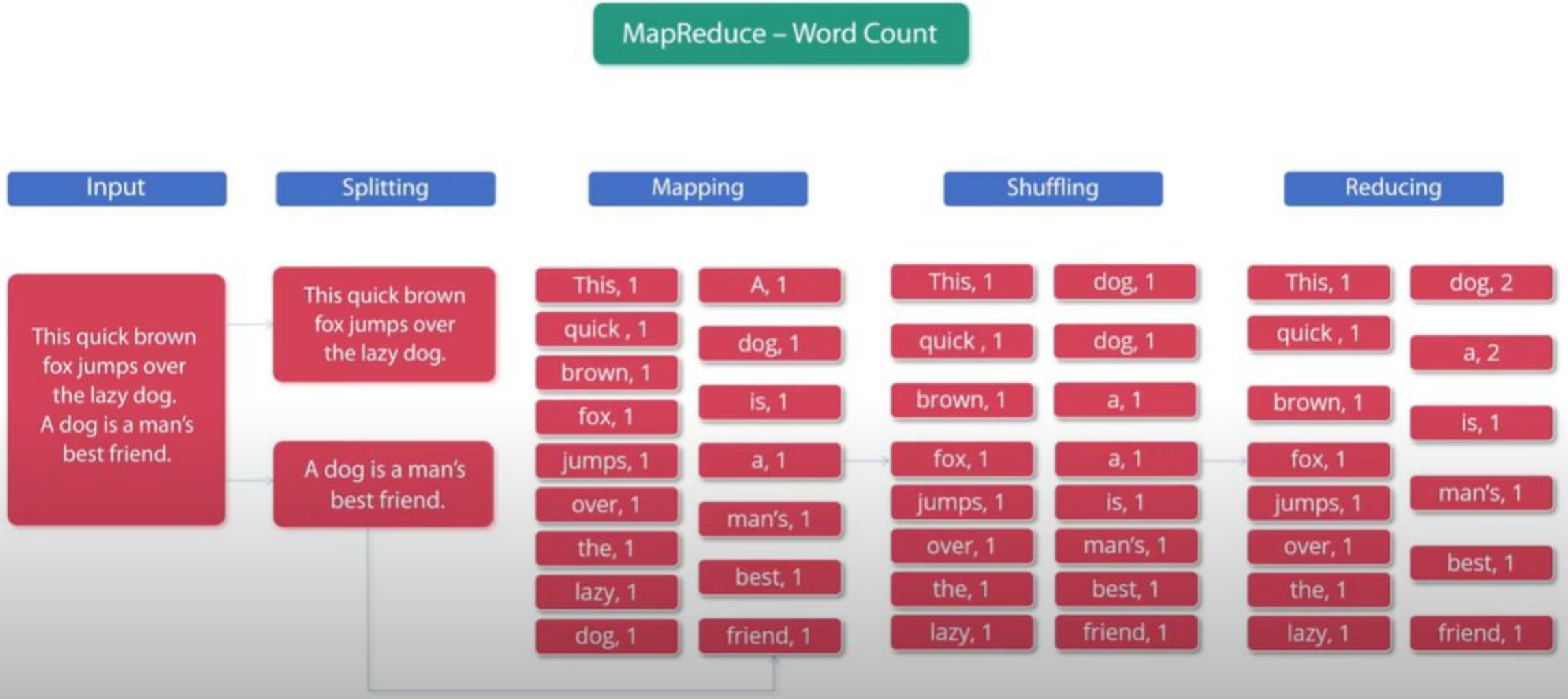
Fill in the blanks

1. Partitioner phase belongs ---------- to task.
2. Combiner is also known ----------.
3. RecordReader converts byte-oriented view into ---------- view.
4. MapReduce sorts the intermediate value based on ---------- .
5. In MapReduce Programming, reduce function is applied ---------- group at a time.

MapReduce – Word Count Example



Word Count Example



Map Execution Phases

Map phase

- Reads assigned input split from HDFS
- Parses input into records as key-value pairs
- Applies map function to each record
- Informs master node of its completion

Partition phase

- Each mapper must determine which reducer will receive each of the outputs
- For any key, the destination partition is the same
- Number of partitions = Number of reducers

Shuffle phase

- Fetches input data from all map tasks for the portion corresponding to the reduce tasks bucket

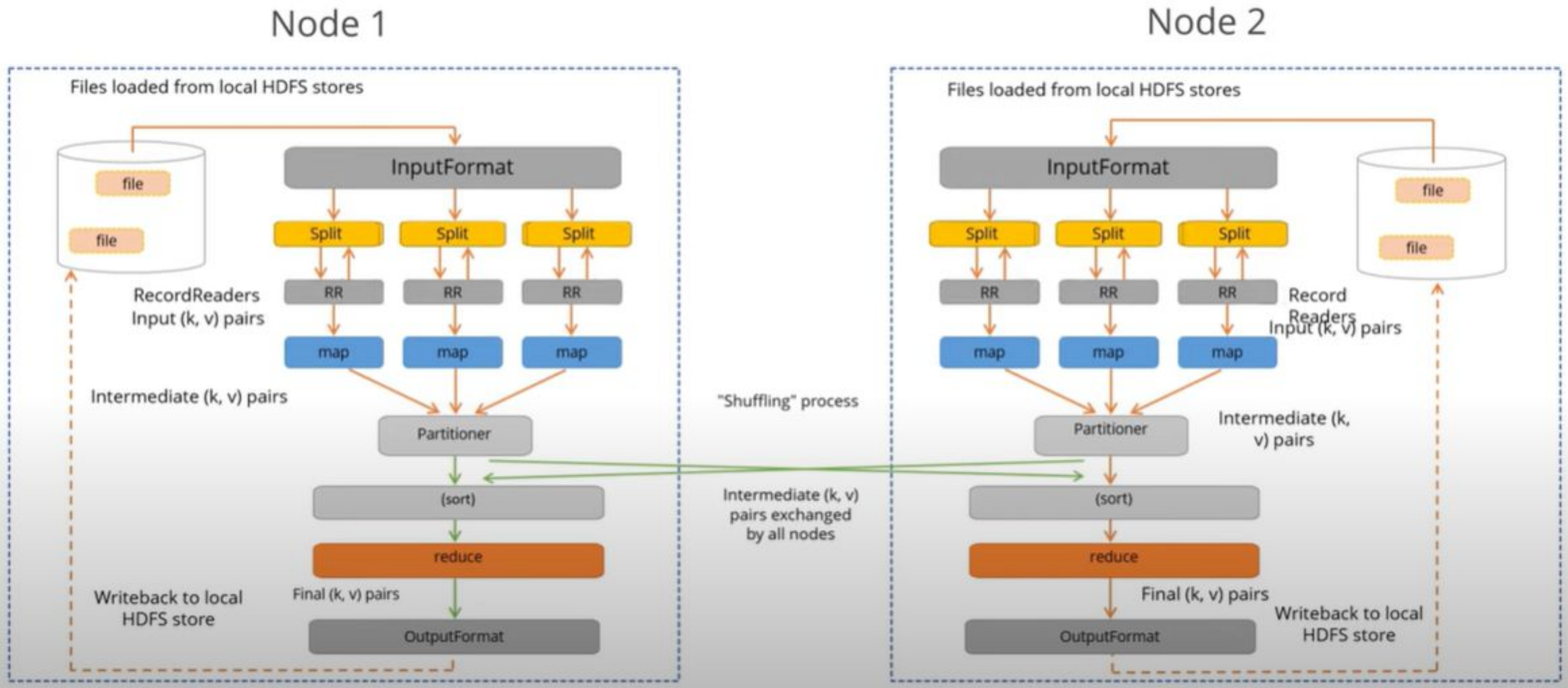
Sort phase

- Merge sorts all map outputs into a single run

Reduce phase

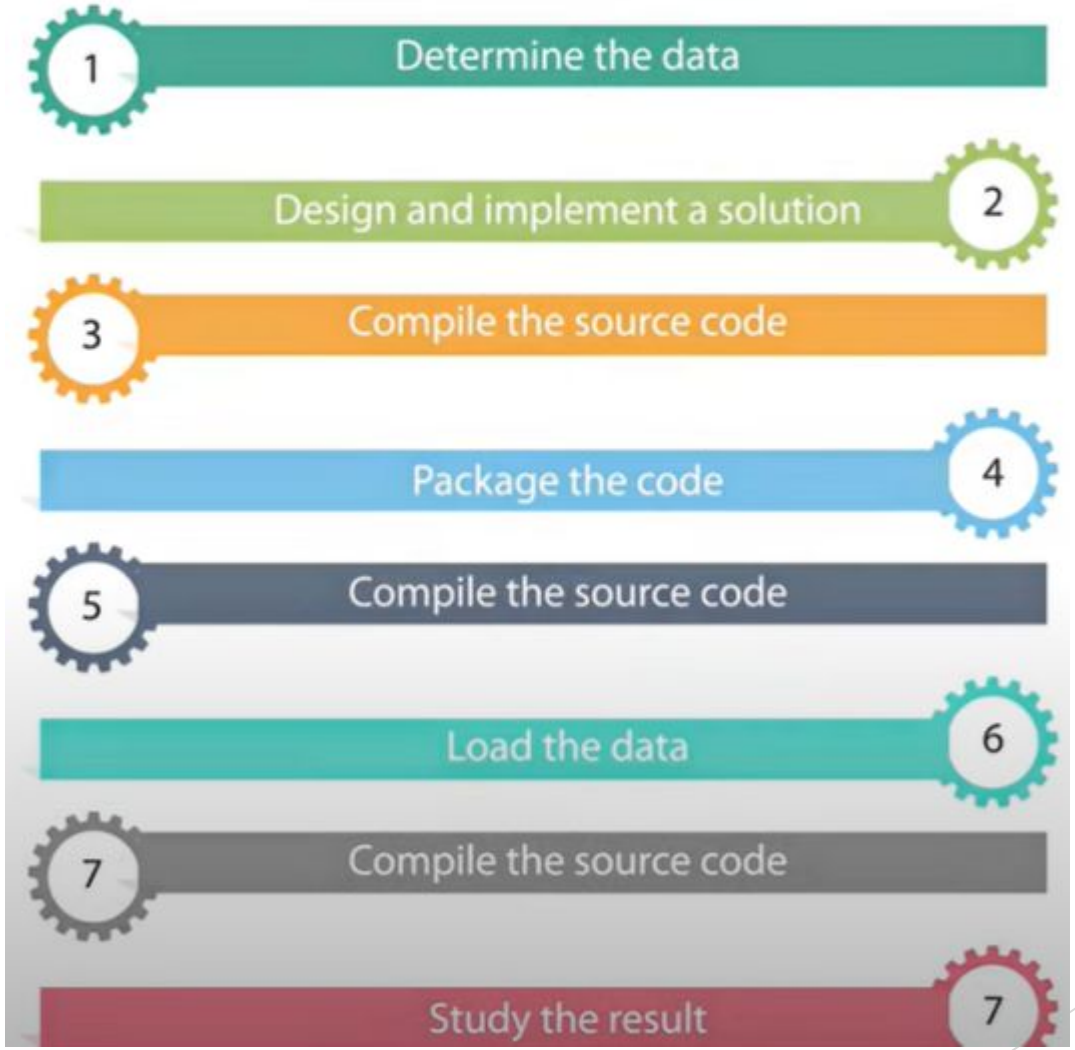
- Applies user-defined reduce function to the merged run

Map Execution: Distributed Two Node Environment

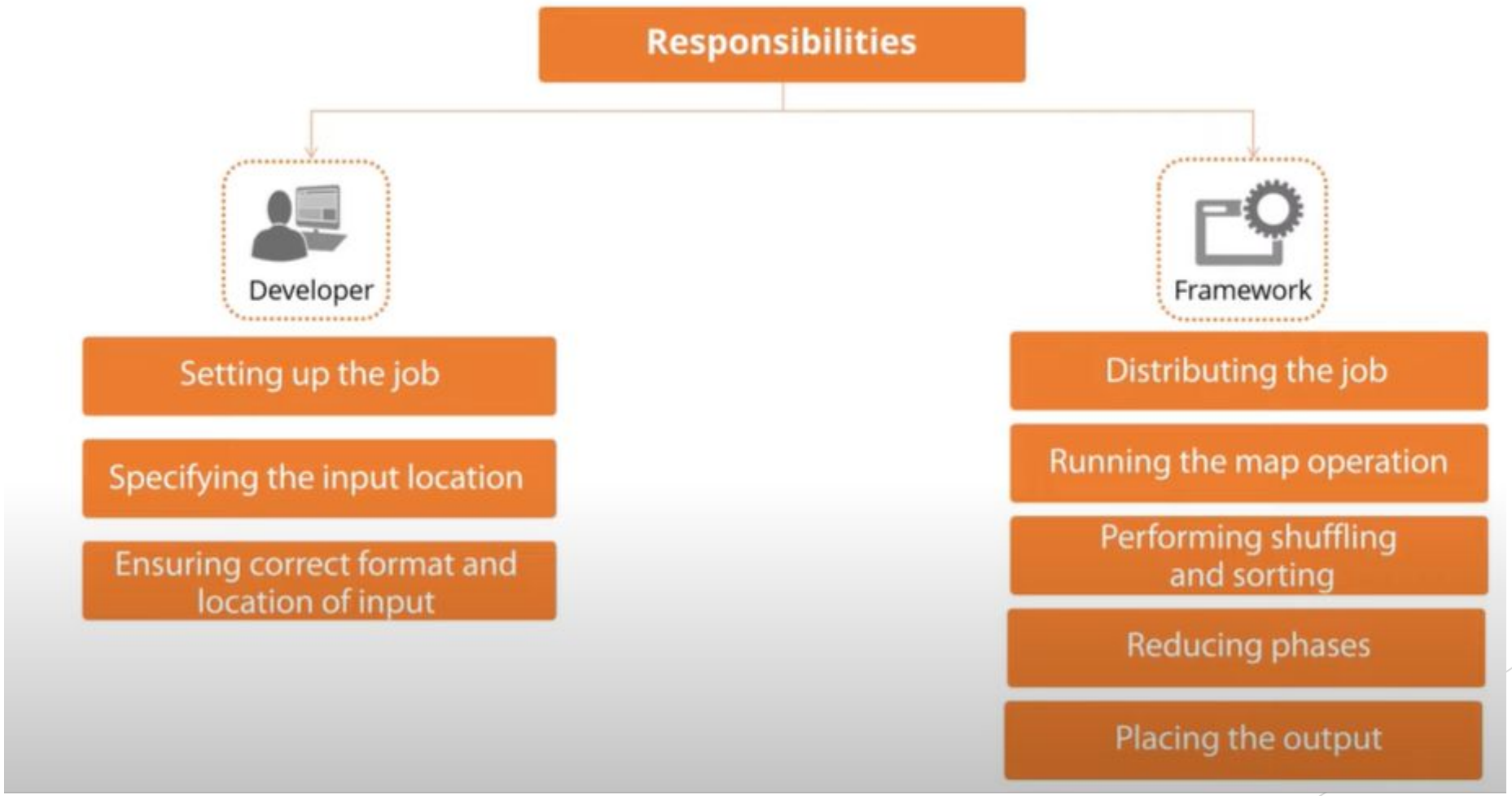


Building a MapReduce Program

The steps to build a MapReduce program are as follows:



MapReduce - Responsibilities



InputFormats in MapReduce

MapReduce can specify how its input is to be read by defining an InputFormat.
The table lists some of the classes of InputFormats provided by the Hadoop framework:

InputFormat classes	Description
Key Value Text Input Format	One key-value pair per line
Text Input Format	Key is the line number, and value is the line
N Line Input Format	Similar to TextInputFormat, but the difference is that there are N number of lines that make an input split
Multi File Input Format	Input format that aggregates multiple files into one split
Sequence File Input Format	The input file is a Hadoop sequence file which contains a serialized key-value pair.

OutputFormat classes	Description
TextOutputFormat	It is the default Output Format and writes records as lines of text. Each key- value pair is separated by a TAB character. This can be customized by using the mapred.textoutputformat.separator property. The corresponding InputFormat is KeyValueTextInputFormat.
SequenceFileOutputFormat	It writes sequence files to save the output. It is compact and compressed.
SequenceFileAsBinaryOutputFormat	It writes key and value in raw binary format into a sequential file container.
MapFileOutputFormat	It writes MapFiles as the output. The keys in a MapFile must be added in an order, and the reducer will emit keys in the sorted order.
MultipleTextOutputFormat	It writes data to multiple files whose names are derived from output keys and values.
MultipleSequenceFileOutputFormat	It creates output in multiple files in a compressed form.

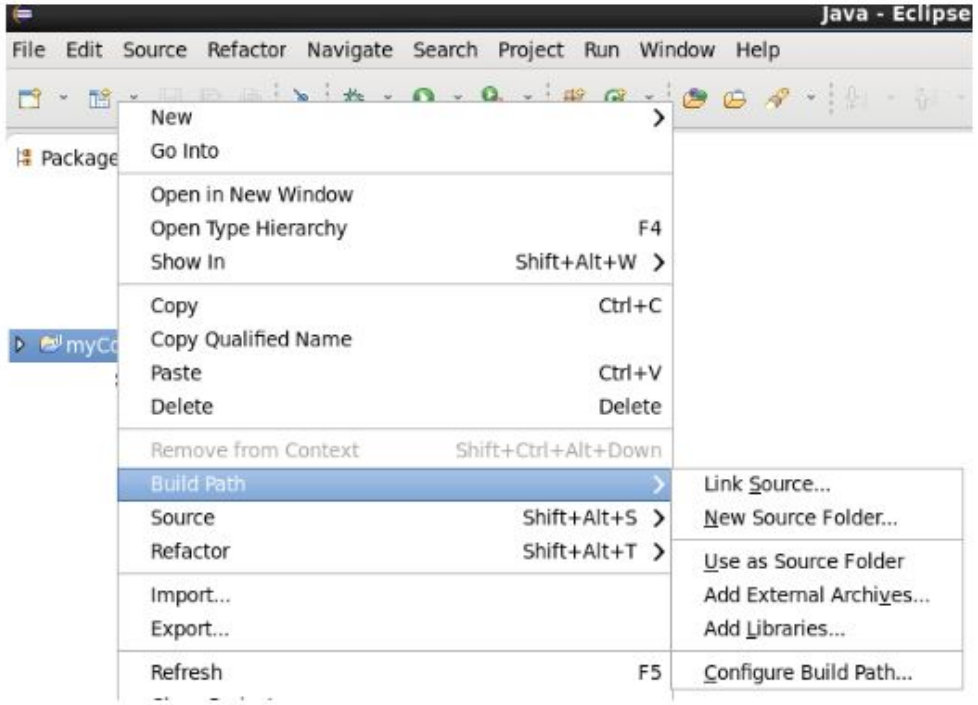
Example-1 – Word Count Program

*NOTE: create a directory and give access permission

```
sudo mkdir MapReduceTutorial
sudo chmod -R 777 MapReduceTutorial
```

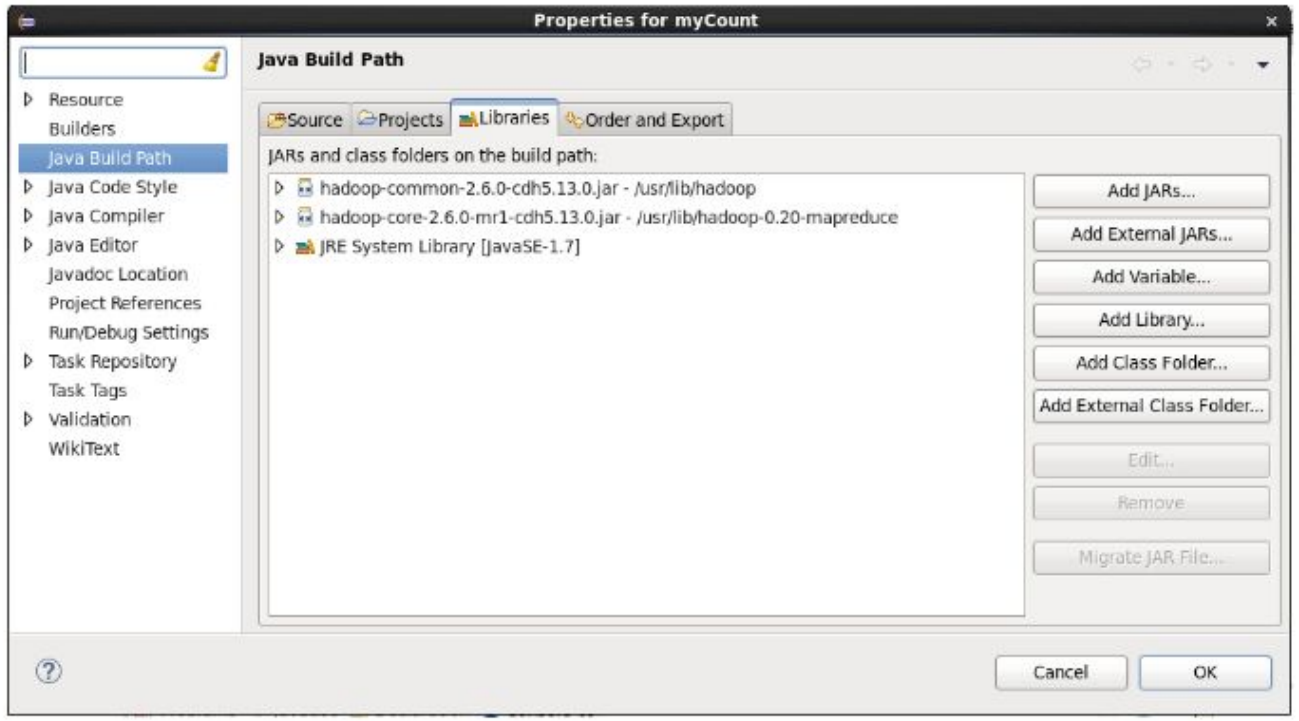
Example-1 - Word Count Program

- Create Three Java Classes into the project. Name them **WCDriver**(having the main function), **WCMapper**, **WCReducer**.
- You have to include two Reference Libraries for that:
  Right Click on **Project** -> then select **Build Path**-> Click on **Configure Build Path**



Example-1 - Word Count Program

- In the above figure, you can see the Add External JARs option on the Right Hand Side. Click on it and add the below mention files. You can find these files in */usr/lib/*
  1. /usr/lib/hadoop-0.20-mapreduce/hadoop-core-2.6.0-mr1-cdh5.13.0.jar
  2. /usr/lib/hadoop/hadoop-common-2.6.0-cdh5.13.0.jar



WCMapper Java Class file

```
// Importing libraries
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;

public class WCMapper extends MapReduceBase implements Mapper<LongWritable,
                                                Text, Text, IntWritable> {

    // Map function
    public void map(LongWritable key, Text value, OutputCollector<Text,
                 IntWritable> output, Reporter rep) throws IOException
    {

        String line = value.toString();

        // Splitting the line on spaces
        for (String word : line.split(" "))
        {
            if (word.length() > 0)
            {
                output.collect(new Text(word), new IntWritable(1));
            }
        }
    }
}
```

WCReducer Java Class file

```
// Importing libraries
import java.io.IOException;
import java.util.Iterator;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;

public class WCReducer extends MapReduceBase implements Reducer<Text,
                                    IntWritable, Text, IntWritable> {

    // Reduce function
    public void reduce(Text key, Iterator<IntWritable> value,
                   OutputCollector<Text, IntWritable> output,
                            Reporter rep) throws IOException
    {

        int count = 0;

        // Counting the frequency of each words
        while (value.hasNext())
        {
            IntWritable i = value.next();
            count += i.get();
        }

        output.collect(key, new IntWritable(count));
    }
}
```

WCDriver Code

```java
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class WCDriver extends Configured implements Tool {

    public int run(String args[]) throws IOException
    {
        if (args.length < 2)
        {
            System.out.println("Please give valid inputs");
            return -1;
        }

        JobConf conf = new JobConf(WCDriver.class);
        FileInputFormat.setInputPaths(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));
        conf.setMapperClass(WCMapper.class);
        conf.setReducerClass(WCReducer.class);
        conf.setMapOutputKeyClass(Text.class);
        conf.setMapOutputValueClass(IntWritable.class);
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);
        JobClient.runJob(conf);
        return 0;
    }

    // Main Method
    public static void main(String args[]) throws Exception
    {
        int exitCode = ToolRunner.run(new WCDriver(), args);
        System.out.println(exitCode);
    }
}
```

Execution

Now you have to make a jar file. Right Click on **Project**-> **Click on Export**-> **Select export destination as Jar File**-> **Name the jar File**(WordCount.jar) -> **Click on next** -> at last **Click on Finish**.

hadoop fs -put WCFile.txt WCFile.txt //Run this command to copy input file into the HDFS

hadoop jar wordcount.jar WCDriver WCFile.txt WCOutput //Run the jar file

hadoop fs -cat WCOutput

Example-2 Sum of Even and odd numbers

Input:

```
1 2 3 4 5 6 7 8 9
```

Output:

```
Even    20   // sum of even numbers
Even    4    // count of even numbers
Odd     25   // sum of odd numbers
Odd     5    // count of odd numbers
```

EOMapper Java Class File

```java
// Importing libraries
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;

public class EOMapper extends MapReduceBase implements Mapper<LongWritable,
                                                    Text, Text, IntWritable> {

    @Override
    // Map function
    public void map(LongWritable key, Text value, OutputCollector<Text,
                                    IntWritable> output, Reporter rep)

    throws IOException
    {
        // Splitting the line into spaces
        String data[] = value.toString().split(" ");

        for (String num : data)
        {

            int number = Integer.parseInt(num);

            if (number % 2 == 1)
            {
                // For Odd Numbers
                output.collect(new Text("ODD"), new IntWritable(number));
            }

            else
            {
                // For Even Numbers
                output.collect(new Text("EVEN"),
                        new IntWritable(number));
            }
        }
    }
```

EORecucer Class File

```
public class EOReducer extends MapReduceBase implements Reducer<Text,
                                IntWritable, Text, IntWritable> {

    @Override
    // Reduce Function
    public void reduce(Text key, Iterator<IntWritable> value,
     OutputCollector<Text, IntWritable> output, Reporter rep)

    throws IOException
    {

        // For finding sum and count of even and odd
        // you don't have to take different variables
        int sum = 0, count = 0;
        if (key.equals("ODD"))
        {
            while (value.hasNext())
            {
                IntWritable i = value.next();

                // Finding sum and count of ODD Numbers
                sum += i.get();
                count++;
            }
        }

        else
        {
            while (value.hasNext())
            {
                IntWritable i = value.next();

                // Finding sum and count of EVEN Numbers
                sum += i.get();
                count++;
            }
        }

        // First sum then count is printed
        output.collect(key, new IntWritable(sum));
        output.collect(key, new IntWritable(count));
    }
}
```

EODriver Class File

```java
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class EODriver extends Configured implements Tool {

    @Override
    public int run(String[] args) throws Exception
    {
        if (args.length < 2)
        {
            System.out.println("Please enter valid arguments");
            return -1;
        }

        JobConf conf = new JobConf(EODriver.class);
        FileInputFormat.setInputPaths(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));
        conf.setMapperClass(EOMapper.class);
        conf.setReducerClass(EOReducer.class);
        conf.setMapOutputKeyClass(Text.class);
        conf.setMapOutputValueClass(IntWritable.class);
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);

        JobClient.runJob(conf);
        return 0;
    }

    // Main Method
    public static void main(String args[]) throws Exception
    {
        int exitcode = ToolRunner.run(new EODriver(), args);
        System.out.println(exitcode);
    }
}
```

Example-3 Sales Data

SalesMapper.Java

```
package SalesCountry;

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.*;

public class SalesMapper extends MapReduceBase implements Mapper <LongWritable, Text, Text, IntWritable> {
	private final static IntWritable one = new IntWritable(1);

	public void map(LongWritable key, Text value, OutputCollector <Text, IntWritable> output, Reporter
reporter) throws IOException {

		String valueString = value.toString();
		String[] SingleCountryData = valueString.split(",");
		output.collect(new Text(SingleCountryData[7]), one);
	}
}
```

SalesReducer Java

```
package SalesCountry;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.*;

public class SalesCountryReducer extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {

	public void reduce(Text t_key, Iterator<IntWritable> values, OutputCollector<Text,IntWritable> output, Reporter reporter)
throws IOException {
		Text key = t_key;
		int frequencyForCountry = 0;
		while (values.hasNext()) {
			// replace type of value with the actual type of our value
			IntWritable value = (IntWritable) values.next();
			frequencyForCountry += value.get();

		}
		output.collect(key, new IntWritable(frequencyForCountry));
	}
}
```

Driver code

```
public class SalesCountryDriver {
   public static void main(String[] args) {
      JobClient my_client = new JobClient();
      // Create a configuration object for the job
      JobConf job_conf = new JobConf(SalesCountryDriver.class);

      // Set a name of the Job
      job_conf.setJobName("SalePerCountry");

      // Specify data type of output key and value
      job_conf.setOutputKeyClass(Text.class);
      job_conf.setOutputValueClass(IntWritable.class);

      // Specify names of Mapper and Reducer Class
      job_conf.setMapperClass(SalesCountry.SalesMapper.class);
      job_conf.setReducerClass(SalesCountry.SalesCountryReducer.class);

      // Specify formats of the data type of Input and output
      job_conf.setInputFormat(TextInputFormat.class);
      job_conf.setOutputFormat(TextOutputFormat.class);

      // Set input and output directories using command line arguments,
      //arg[0] = name of input directory on HDFS, and arg[1] =  name of output directory to be created to store the output file.

      FileInputFormat.setInputPaths(job_conf, new Path(args[0]));
      FileOutputFormat.setOutputPath(job_conf, new Path(args[1]));

      my_client.setConf(job_conf);
      try {
         // Run the job
         JobClient.runJob(job_conf);
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Answer a few quick questions…

Match the columns

Column A	Column B
HDFS	DataNode
MapReduce	NameNode
Programming Master node	Processing
Slave node	Data
Hadoop Implementation	Google File System and MapReduce
	Storage

Match the columns

Column A	Column B
JobTracker	Executes Task
MapReduce	Schedules Task
TaskTracker	Programming Model
Job Configuration	Converts input into Key Value pair Job Parameters
Map	

Thank you