UNIT – III

Introduction to Hadoop and MapReduce Programming

Hadoop Overview, **HDFS** (Hadoop Distributed File System), Processing–Data with Hadoop, Managing Resources and Applications with Hadoop **YARN** (Yet another Resource Negotiator).

Introduction to MAPREDUCE Programming: Introduction, Mapper, Reducer, Combiner, Partitioner, Searching, Sorting, Compression.

Q) Explain the differences between Hadoop and RDBMS

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 formats(xml, json)
Processing	OLTP	Analytical, big data processing Hadoop clusters, node require any consistent relationships between data
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 commodity hardware less configure hardware.
Cost	Cost around $10,000 to $14,000 per terabytes of storage	Cost around $4000 per terabytes of storage.

Q) What is Hadoop? Explain features of hadoop.

- Hadoop is an open source framework that is meant for storage and processing of big data in a distributed manner.
- It is the best solution for handling big data challenges.

Some important **features** of Hadoop are –

- **Open Source** – Hadoop is an open source framework which means it is available free of cost. Also, the users are allowed to change the source code as per their requirements.
- **Distributed Processing** – Hadoop supports distributed processing of data i.e. faster processing. The data in Hadoop HDFS is stored in a distributed manner and MapReduce is responsible for the parallel processing of data.
- **Fault Tolerance** – Hadoop is highly fault-tolerant. It creates three replicas for each block (default) at different nodes.
- **Reliability** – Hadoop stores data on the cluster in a reliable manner that is independent of machine. So, the data stored in Hadoop environment is not affected by the failure of the machine.

- **Scalability** – It is compatible with the other hardware and we can easily add/remove the new hardware to the nodes.
- **High Availability** – The data stored in Hadoop is available to access even after the hardware failure. In case of hardware failure, the data can be accessed from another node.

The **core components** of Hadoop are –

1. **HDFS:** (Hadoop Distributed File System) – HDFS is the basic storage system of Hadoop. The large data files running on a cluster of commodity hardware are stored in HDFS. It can store data in a reliable manner even when hardware fails. The key aspects of HDFS are:
    a. Storage component
    b. Distributes data across several nodes
    c. Natively redundant.
2. **Map Reduce:** MapReduce is the Hadoop layer that is responsible for data processing. It writes an application to process unstructured and structured data stored in HDFS.
    It is responsible for the parallel processing of high volume of data by dividing data into independent tasks. The processing is done in two phases Map and Reduce.
    The **Map** is the first phase of processing that specifies complex logic code and the
    Reduce is the second phase of processing that specifies light-weight operations.
    The key aspects of Map Reduce are:
    a. Computational frame work
    b. Splits a task across multiple nodes
    c. Processes data in parallel

Q) Explain Hadoop Architecture with a neat sketch.

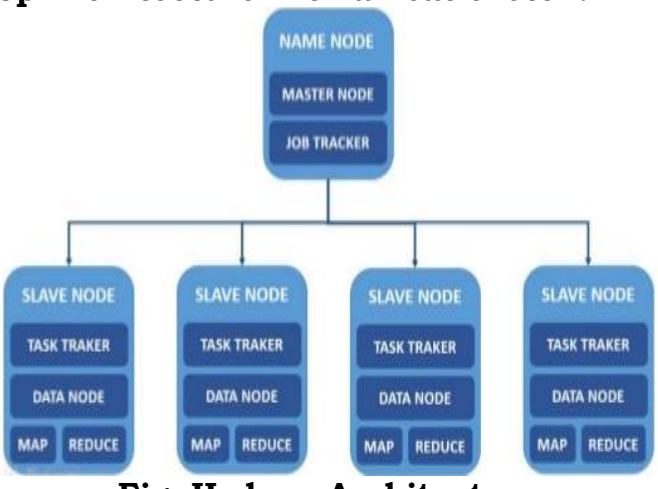


Fig. Hadoop Architecture

Hadoop Architecture **is a distributed Master-slave architecture.**

Master HDFS: Its main responsibility is partitioning the data storage across the slave nodes. It also keep track of locations of data on Datanodes.

Master Map Reduce: It decides and schedules computation task on slave nodes.

NOTE: Based on marks for the question explain hdfs daemons and mapreduce daemons.

Q) Explain the following

a) Modules of Apache Hadoop framework

There are four basic or core components:

Hadoop Common: It is a set of common utilities and libraries which handle other Hadoop modules. It makes sure that the hardware failures are managed by Hadoop cluster automatically.

Hadoop YARN: It allocates resources which in turn allow different users to execute various applications without worrying about the increased workloads.

HDFS: It is a Hadoop Distributed File System that stores data in the form of small memory blocks and distributes them across the cluster. Each data is replicated multiple times to ensure data availability.

Hadoop MapReduce: It executes tasks in a parallel fashion by distributing the data as small blocks.

b) Hadoop Modes of Installations

i. **Standalone**, or local mode: which is one of the least commonly used environments, which only for **running and debugging** of MapReduce programs. This mode does not use HDFS nor it launches any of the hadoop daemon.

ii. **Pseudo-distributed mode(Cluster of One)**, which runs all daemons on single machine. It is most commonly used in development environments.

iii. **Fully distributed mode**, which is most commonly used in production environments. This mode runs all daemons on a cluster of machines rather than single one.

c) XML File configrations in Hadoop.

core-site.xml – This configuration file contains Hadoop core configuration settings, for example, I/O settings, very common for MapReduce and HDFS.

mapred-site.xml – This configuration file specifies a framework name for MapReduce by setting mapreduce.framework.name

hdfs-site.xml – This configuration file contains HDFS daemons configuration settings. It also specifies default block permission and replication checking on HDFS.

yarn-site.xml – This configuration file specifies configuration settings for ResourceManager and NodeManager.

Q) Explain features of HDFS. Discuss the design of Hadoop distributed file system and concept in detail.

HDFS: (Hadoop Distributed File System) – HDFS is the basic storage system of Hadoop. The large data files running on a cluster of commodity hardware are stored in HDFS. It can store data in a reliable manner even when hardware fails. The key aspects of HDFS are:

- HDFS is developed by the inspiration of Google File System(GFS).
- Storage component: Stores data in hadoop
- Distributes data across several nodes: divides large file into blocks and stores in various data nodes.
- Natively redundant: replicates the blocks in various data nodes.
- High Throughput Access: Provides access to data blocks which are nearer to the client.
- Re-replicates the nodes when nodes are failed.

Fig. Features of HDFS

HDFS Daemons:

(i) NameNode

- The NameNode is the master of HDFS that directs the slave DataNodes to perform I/O tasks.
- **Blocks**: HDFS breaks large file into smaller pieces called blocks.
- **rackID**: NameNode uses rackID to identify data nodes in the rack. (rack is a collection of datanodes with in the cluster) NameNode keep track of blocks of a file.
- File System **Namespace:** NameNode is the book keeper of HDFS. It keeps track of how files are broken down into blocks and which DataNode stores these blocks. It is a collection of files in the cluster.
- **FsImage:** file system namespace includes mapping of blocks of a file, file properties and is stored in a file called FsImage.
- **EditLog:** namenode uses an EditLog (transaction log) to record every transaction that happens to the file system metadata.
- NameNode is **single point of failure** of Hadoop cluster.

HDFS KEY POINTS		
BLOCK STRUCTURED FILE	DEFAULT REPLICATION FACTOR: 3	DEFAULT BLOCK SIZE: 64MB/128MB

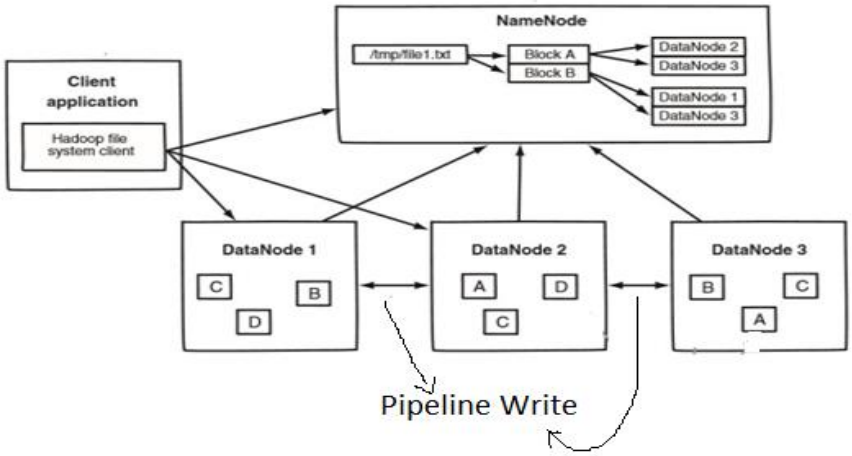


Fig. HDFS Architecture

(ii) DataNode

- Multiple data nodes per cluster. Each slave machine in the cluster have DataNode daemon for reading and writing HDFS blocks of actual file on local file system.
- During pipeline read and write DataNodes communicate with each other.
- It also continuously Sends **"heartbeat"** message to NameNode to ensure the connectivity between the Name node and the data node.
- If no heartbeat is received for a period of time NameNode assumes that the DataNode had failed and it is re-replicated.

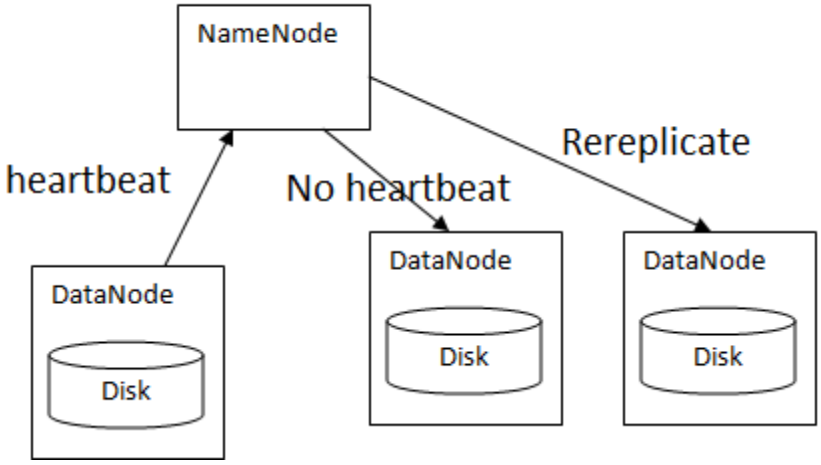


Fig. Interaction between NameNode and DataNode.

(iii)Secondary name node

- Takes snapshot of HDFS meta data at intervals specified in the hadoop configuration.
- Memory is same for secondary node as NameNode.
- But secondary node will run on a different machine.
- In case of **name node failure** secondary name node can be configured manually to bring up the cluster i.e; we make secondary namenode as name node.

File Read operation:

The steps involved in the File Read are as follows:

1. The client opens the file that it wishes to read from by calling open() on the DFS.
2. The DFS communicates with the NameNode to get the location of data blocks. NameNode returns with the addresses of the DataNodes that the data blocks are stored on.
   Subsequent to this, the DFS returns an FSD to client to read from the file.
3. Client then calls read() on the stream DFSInputStream, which has addresses of DataNodes for the first few block of the file.
4. Client calls read() repeatedly to stream the data from the DataNode.
5. When the end of the block is reached, DFSInputStream closes the connection with the DataNode. It repeats the steps to find the best DataNode for the next block and subsequent blocks.
6. When the client completes the reading of the file, it calls close() on the FSInputStream to the connection.

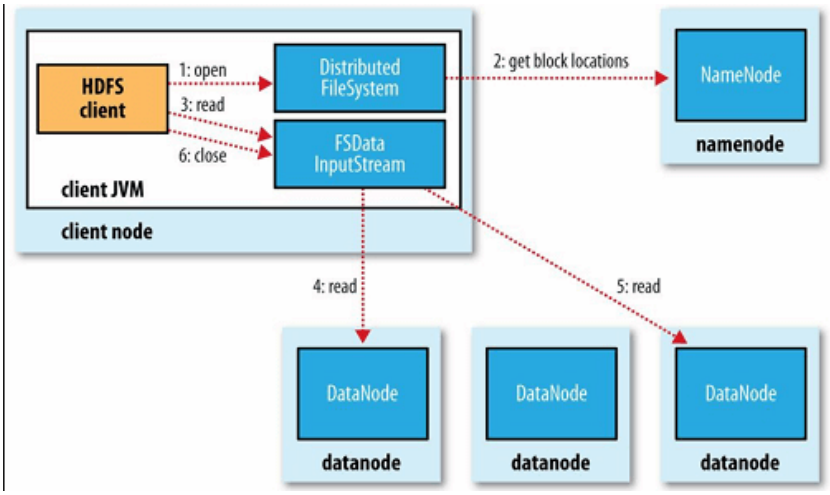


Fig. File Read Anatomy

File Write operation:

1. The client calls create() on DistributedFileSystem to create a file.
2. An RPC call to the namenode happens through the DFS to create a new file.
3. As the client writes data, data is split into packets by DFSOutputStream, which is then writes to an internal queue, called data queue. Datastreamer consumes the data queue.

4. Data streamer streams the packets to the first DataNode in the pipeline. It stores packet and forwards it to the second DataNode in the pipeline.
5. In addition to the internal queue, DFSOutputStream also manages on "Ackqueue" of the packets that are waiting for acknowledged by DataNodes.
6. When the client finishes writing the file, it calls close() on the stream.

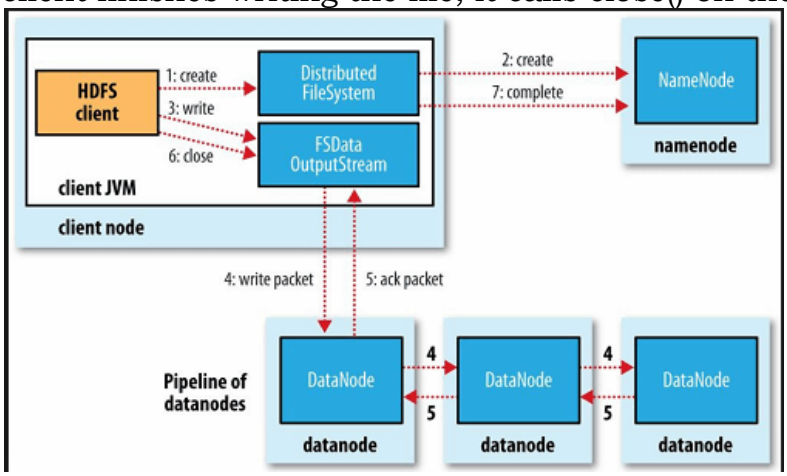


Fig. File Write Anatomy

Special features of HDFS:

1. **Data Replication:** There is absolutely no need for a client application to track all blocks. It directs client to the nearest replica to ensure high performance.
2. **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.

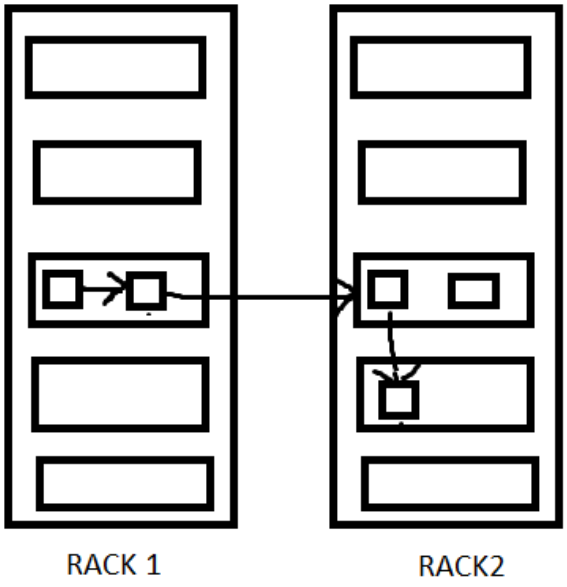


Fig. File Replacement Strategy

Q) Explain basic HDFS File operations with an example.

1. Creating a directory:
   Syntax: hdfs dfs –mkdir <path>
   Eg. hdfs dfs –mkdir /chp

2. Remove a file in specified path:
   Syntax: hdfs dfs –rm <src>
   Eg. hdfs dfs –rm /chp/abc.txt

3. Copy file from local file system to hdfs:
   Syntax: hdfs dfs –copyFromLocal <src> <dst>
   Eg. hdfs dfs –copyFromLocal /home/hadoop/sample.txt /chp/abc1.txt

4. To display list of contents in a directory:
   Syntax: hdfs dfs –ls <path>
   Eg. hdfs dfs –ls /chp

5. To display contents in a file:
   Syntax: hdfs dfs –cat <path>
   Eg. hdfs dfs –cat /chp/abc1.txt
6. Copy file from hdfs to local file system:
   Syntax: hdfs dfs –copyToLocal <src <dst>
   Eg. hdfs dfs –copyToLocal /chp/abc1.txt /home/hadoop/Desktop/sample.txt

7. To display last few lines of a file:
   Syntax: hdfs dfs –tail <path>
   Eg. hdfs dfs –tail /chp/abc1.txt

8. Display aggregate length of file in bytes:
   Syntax: hdfs dfs –du <path>
   Eg. hdfs dfs –du /chp

9. To count no.of directories, files and bytes under given path:
   Syntax: hdfs dfs –count <path>
   Eg. hdfs dfs –count /chp
   o/p: 1 1 60

10. Remove a directory from hdfs
    Syntax: hdfs dfs –rmr <path>
    Eg. hdfs dfs rmr /chp

Q) Explain the importance of MapReduce in Hadoop environment for processing data.

- MapReduce programming helps to process massive amounts of data in parallel.
- Input data set splits into independent chunks. Map tasks process these independent chunks completely in a parallel manner.
- Reduce task-provides reduced output by combining the output of various mapers. There are two **daemons** associated with MapReduce Programming: **JobTracker** and **TaskTracer**.

JobTracker:

JobTracker is a master daemon responsible for executing over MapReduce job.
It provides connectivity between Hadoop and application.

Whenever code submitted to a cluster, JobTracker creates the execution plan by deciding which task to assign to which node.

It also monitors all the running tasks. When task fails it automatically re-schedules the task to a different node after a predefined number of retires.

There will be one job Tracker process running on a single Hadoop cluster. Job Tracker processes run on their own Java Virtual machine process.

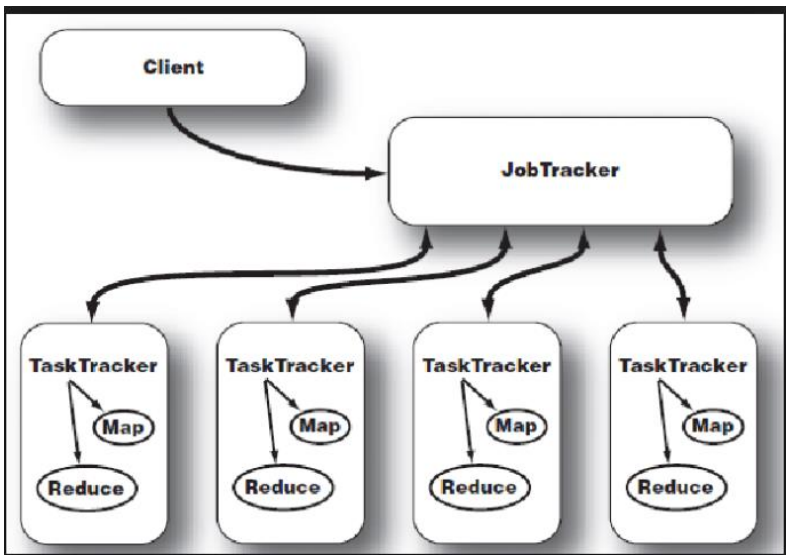


Fig. Job Tracker and Task Tracker interaction

TaskTracker:

This daemon is responsible for executing individual tasks that is assigned by the Job Tracker.

Task Tracker continuously sends **heartbeat** message to job tracker. When a job tracker fails to receive a heartbeat message from a

TaskTracker, the JobTracker assumes that the TaskTracker has failed and resubmits the task to another available node in the cluster.

Map Reduce Framework	
Phases: **Map:** Converts input into key-value pairs. **Reduce:** Combines output of mappers and produces a reduced result set.	**Daemons:** **JobTracker:** Master, Schedules Task **TaskTracker:** Slave, Execute task

MapReduce working:

MapReduce divides a data analysis task into two parts – Map and Reduce. In the example given below: there two mappers and one reduce.

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

Steps:

1. First, the input dataset is split into multiple pieces of data.
2. Next, the framework creates a master and several slave processes and executes the worker processes remotely.
3. Several map tasks work simultaneously and read pieces of data that were assigned to each map task.
4. Map worker uses partitioner function to divide the data into regions.
5. When the map slaves complete their work, the master instructs the reduce slaves to begin their work.
6. When all the reduce slaves complete their work, the master transfers the control to the user program.

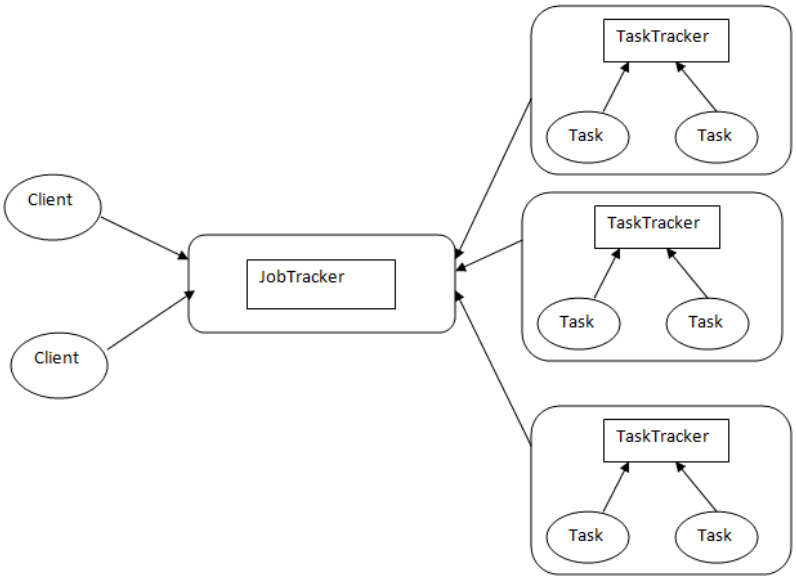


Fig. MapReduce Programming Architecture

A MapReduce programming using Java requires three classes:

1. Driver Class: This class specifies Job configuration details.
2. MapperClass: this class overrides the MapFunction based on the problem statement.
3. Reducer Class: This class overrides the Reduce function based on the problem statement.

NOTE: Based on marks given write MapReduce example if necessary with program.

Q) Explain difference between Hadoop1X and Hadoop2X

Limitations of Hadoop 1.0: HDFS and MapReduce are core components, while other components are built around the core.

1. Single namenode is responsible for entire namespace.
2. It is Restricted processing model which is suitable for batch-oriented mapreduce jobs.
3. Not supported for interactive analysis.
4. Not suitable for Machine learning algorithms, graphs, and other memory intensive algorithms
5. MapReduce is responsible for cluster resource management and data Processing.

HDFS Limitation: The NameNode can quickly become overwhelmed with load on the system increasing. In Hadoop 2.x this problem is resolved.

Hadoop 2: Hadoop 2.x is YARN based architecture. It is general processing platform. YARN is not constrained to MapReduce only. One can run multiple applications in Hadoop 2.x in which all applications share common resource management.

Hadoop 2.x can be used for various types of processing such as Batch, Interactive, Online, Streaming, Graph and others.

HDFS 2 consists of two major **components**

a) **NameSpace:** Takes care of file related operations such as creating files, modifying files and directories

b) **Block storage service:** It handles data node cluster management and replication.

HDFS 2 Features:

Horizontal scalability: HDFS Federation uses multiple independent NameNodes for horizontal scalability. The DataNodes are common storage for blocks and shared by all NameNodes. All DataNodes in the cluster registers with each NameNode in the cluster.

High availability: High availability of NameNode is obtained with the help of Passive Standby NameNode.

Active-Passive NameNode handles failover automatically. All namespace edits are recorded to a shared **NFS**(Network File Storage) Storage and there is a single writer at an point of time.

Passive NameNode reads edits from shared storage and keeps updated metadata information.
Incase of Active NameNode failure, Passive NameNode becomes an Active NameNode automatically. Then it starts writing to the shared storage.

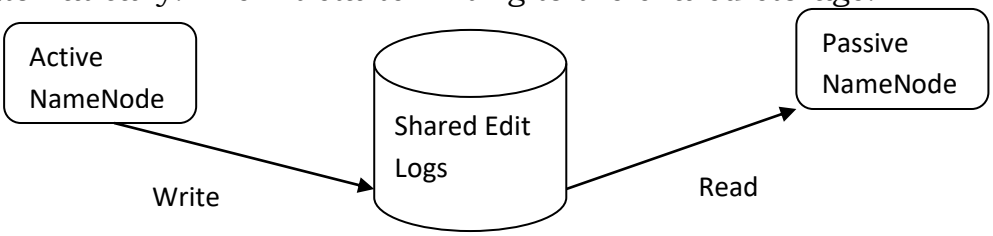


Fig. Active and Passive NameNode Interaction

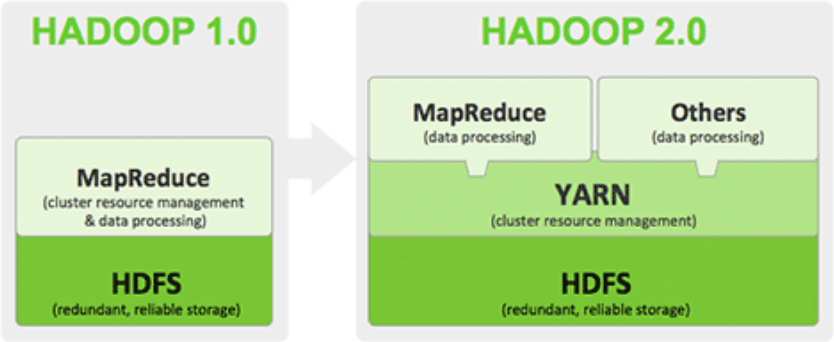


Fig. Comparing Hadoop1.0 and Hadoop 2.0

	Hadoop1X	**Hadoop2X**
1	Supports MapReduce (MR) processing model only. Does not support non-MR tools	Allows working in MR as well as other distributed computing models like Spark, & HBase coprocessors.
2	MR does both processing and cluster-resource management.	YARN does cluster resource management and processing is done using different processing models.
3	Has limited scaling of nodes. Limited to 4000 nodes per cluster	Has better scalability. Scalable up to 10000 nodes per cluster
4	Works on concepts of slots – slots can run either a Map task or a Reduce task only.	Works on concepts of containers. Using containers can run generic tasks.
5	A single Namenode to manage the entire namespace.	Multiple Namenode servers manage multiple namespaces.
6	Has Single-Point-of-Failure (SPOF) – because of single Namenode.	Has to feature to overcome SPOF with a standby Namenode and in the case of Namenode failure, it is configured for automatic recovery.
7	MR API is compatible with Hadoop1x. A program written in Hadoop1 executes in Hadoop1x without any additional files.	MR API requires additional files for a program written in Hadoop1x to execute in Hadoop2x.

8	Has a limitation to serve as a platform for event processing, streaming and real-time operations.	Can serve as a platform for a wide variety of data analytics-possible to run event processing, streaming and real-time operations.
9	Does not support Microsoft Windows	Added support for Microsoft windows

Q) Explain in detail about YARN?

The fundamental idea behind the YARN(Yet Another Resource Negotiator) architecture is to splitting the JobTracker reponsibility of resource management and job scheduling/monitoring into separate daemons.

Basic concepts of YARN are Application and Container.
Application is a job submitted to system.
Ex: MapReduce job.
Container: Basic unit of allocation. Replaces fixed map/reduce slots. Fine-grained resource allocation across multiple resource type
Eg. Container_0: 2GB, 1CPU
Container_1: 1GB, 6CPU

Daemons that are part of YARN architecture are:
1. **Global Resource Manager:** The main responsibility of Global Resource Manager is to distribute resources among various applications.
It has two main components:
Scheduler: The pluggable scheduler of ResourceManager decides allocation of resources to various running applications. The scheduler is just that, a pure scheduler, meaning it does NOT monitor or track the status of the application.
Application Manager: It does:
- Accepting job submissions.
- Negotiating resources(container) for executing the application specific ApplicationMaster
- Restarting the ApplicationMaster in case of failure

2. **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.

3. **Per-Application Application Master:** Per-application Application master is an application specific entity. It's responsibility is to negotiate required resources for execution from the ResourceManager.

It works along with the NodeManager for executing and monitoring component tasks.

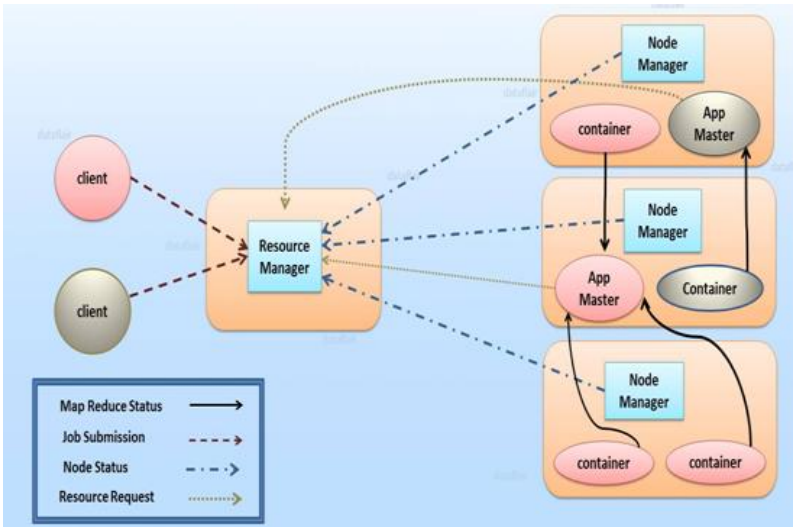


Fig. YARN Architecture

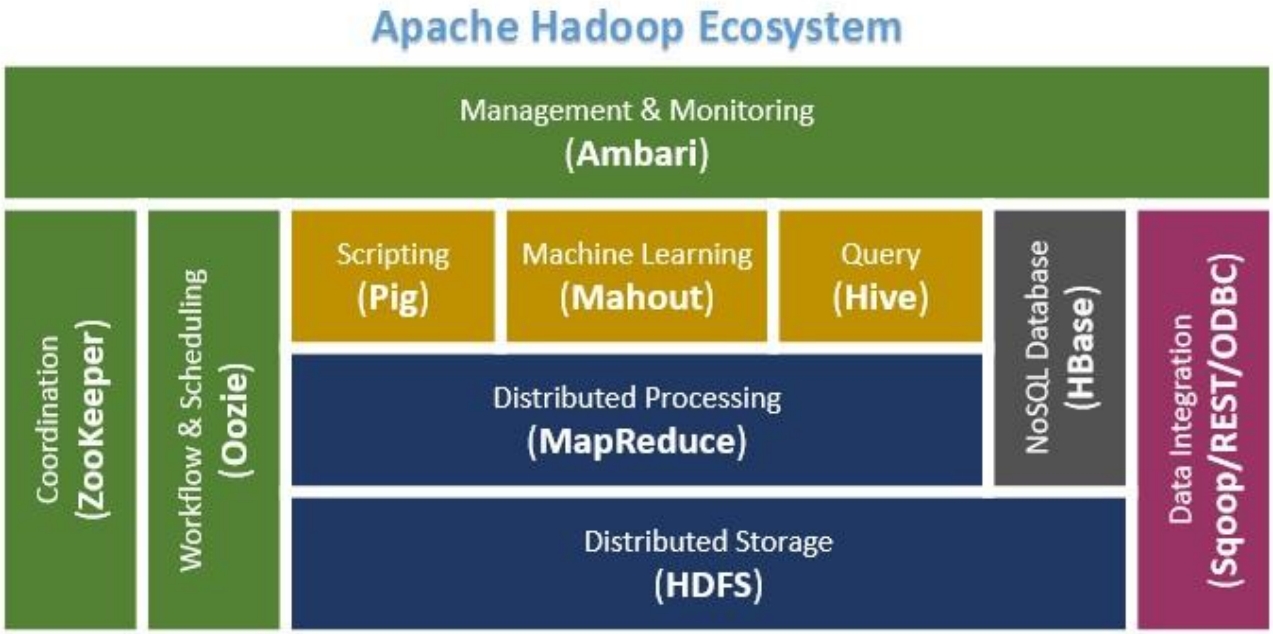
The steps involved in YARN architecture are:

1. The client program submits an application.
2. The Resource Manager launches the Application Master by assigning some container.
3. The Application Master registers with the Resource manager.
4. On successful container allocations, the application master launches the container by providing the container launch specification to the NodeManager.
5. The NodeManager executes the application code.
6. During the application execution, the client that submitted the job directly communicates with the Application Master to get status, progress updates.
7. Once the application has been processed completely, the application master deregisters with the ResourceManager and shuts down allowing its own container to be repurposed.

Q) Explain Hadoop Ecosystem in detail.

The following are the components of Hadoop ecosystem:

1. **HDFS:** Hadoop Distributed File System. It simply stores data files as close to the original form as possible.
2. **HBase:** It is Hadoop's distributed column based database. It supports structured data storage for large tables.
3. **Hive:** It is a Hadoop's data warehouse, enables analysis of large data sets using a language very similar to SQL. So, one can access data stored in hadoop cluster by using Hive.
4. **Pig:** Pig is an easy to understand data flow language. It helps with the analysis of large data sets which is quite the order with Hadoop without writing codes in MapReduce paradigm.



5. **ZooKeeper:** It is an open source application that configures synchronizes the distributed systems.
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.

Q) Describe differences between SQL and MapReduce

Characteristic	**SQL**	**MapReduce(Hadoop1X)**
Access	Interactive and Batch	Batch
Structure	Static	Dynamic
Updates	Read and Write many times	Write once, Read many times
Integrity	High	Low
Scalability	Nonlinear	Linear

Q) What is MapReduce. Explain indetail different phases in MapReduce. (or) Explain MapReduce anatomy.

MapReduce is a programming model for data processing. Hadoop can run MapReduce programs written in Java, Ruby and Python.
MapReduce programs are inherently parallel, thus very large scale data analysis can be done fastly.

In MapReduce programming, Jobs(applications) are split into a set of **map tasks and reduce tasks**.
Map task takes care of **loading, parsing, transforming and filtering.**
The responsibility of reduce task is **grouping and aggregating** data that is produced by map tasks to generate final output.
Each **map** task is broken down into the following phases:

1. Record Reader 2. Mapper
3. Combiner 4.Partitioner.

The output produced by the map task is known as intermediate <keys, value> pairs. These intermediate <keys, value> pairs are sent to reducer.
The **reduce** tasks are broken down into the following phases:

1. Shuffle 2. Sort
3. Reducer 4. Output format.

Hadoop assigns map tasks to the DataNode where the actual data to be processed resides. This way, Hadoop ensures data locality. **Data locality** means that data is not moved over network; only computational code moved to process data which saves network bandwidth.

Mapper Phases:
Mapper maps the input <keys, value> pairs into a set of intermediate <keys, value> pairs.
Each **map** task is broken into following phases:

1. **RecordReader**: converts byte oriented view of input in to Record oriented view and presents it to the Mapper tasks. It presents the tasks with keys and values.
   i) InputFormat: It reads the given input file and splits using the method **getsplits()**.
   ii) Then it defines RecordReader using **createRecordReader()** which is responsible for generating <keys, value> pairs.

2. **Mapper**: Map function works on the <keys, value> pairs produced by RecordReader and generates intermediate (key, value) pairs.
   Methods:
   - protected void cleanup(Context context): called once at tend of task.
   - **protected void map(KEYIN key, VALUEIN value, Context context): called once for each key-value pair in input split.**
   - void run(Context context): user can override this method for complete control over execution of Mapper.
   - protected void setup(Context context): called once at beginning of task to perform required activities to initiate map() method.

3. **Combiner**: It takes intermediate <keys, value> pairs provided by mapper and applies user specific aggregate function to only one mapper. It is also known as local Reducer.

We can optionally specify a combiner using **Job.setCombinerClass(ReducerClass)** to perform local aggregation on intermediate outputs.

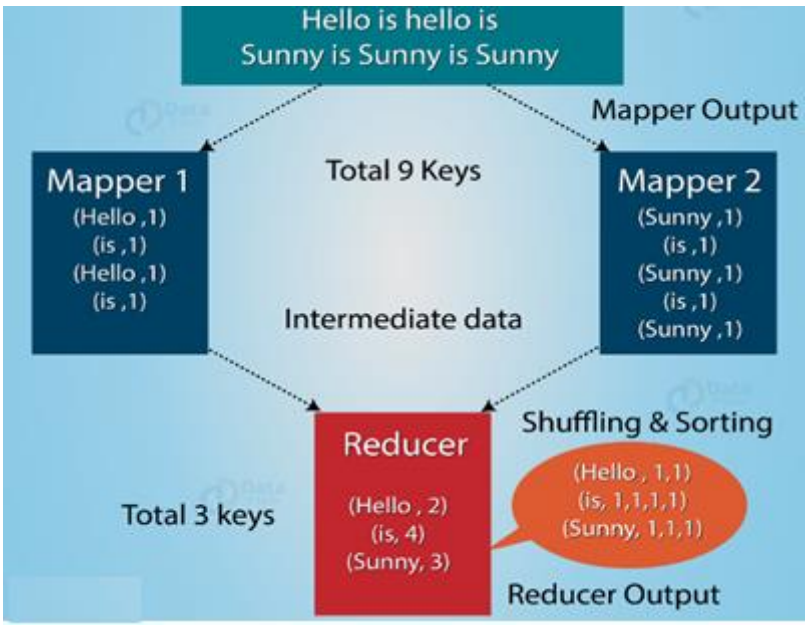


Fig. MapReduce without Combiner class

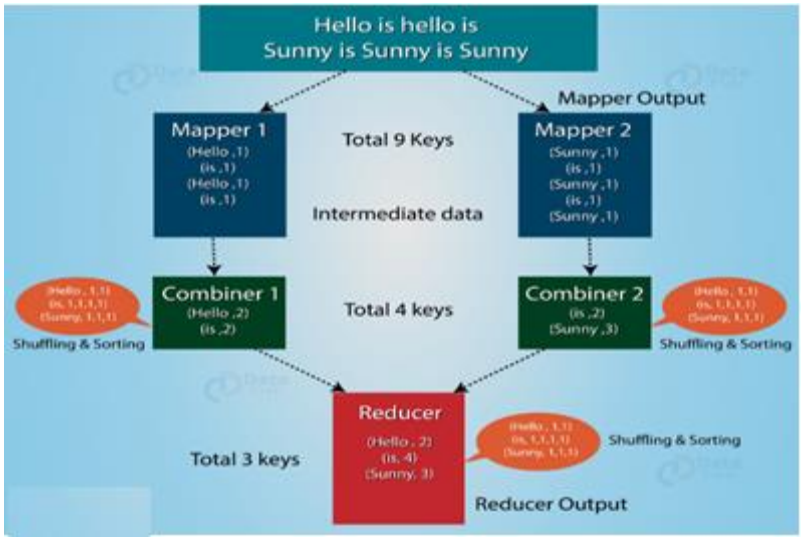


Fig. MapReduce with Combiner class

4. **Partitioner**: Take intermediate <keys, value> pairs produced by the mapper, splits them into partitions the data using a user-defined condition.

The default behavior is to hash the key to determine the reducer.User can control by using the method:

int getPartition(KEY key, VALUE value, int numPartitions)

Reducer Phases:

1. **Shuffle & Sort**:
   - Downloads the grouped key-value pairs onto the local machine, where the Reducer is running.
   - The individual <keys, value> pairs are sorted by **key** into a larger data list.
   - The data list groups the equivalent keys together so that their values can be iterated easily in the Reducer task.
2. **Reducer**:
   - The Reducer takes the grouped key-value paired data as input and runs a Reducer function on each one of them.
   - Here, the data can be aggregated, filtered, and combined in a number of ways, and it requires a wide range of processing.
   - Once the execution is over, it gives zero or more key-value pairs to the final step.

Methods:

- protected void cleanup(Context context): called once at tend of task.

- **protected void reduce(KEYIN key, VALUEIN value, Context context): called once for each key-value pair.**

- void run(Context context): user can override this method for complete control over execution of Reducer.

- protected void setup(Context context): called once at beginning of task to perform required activities to initiate reduce() method.

3. **Output format**:
   - In the output phase, we have an output formatter that translates the final key-value pairs from the Reducer function and writes them onto a file using a record writer.

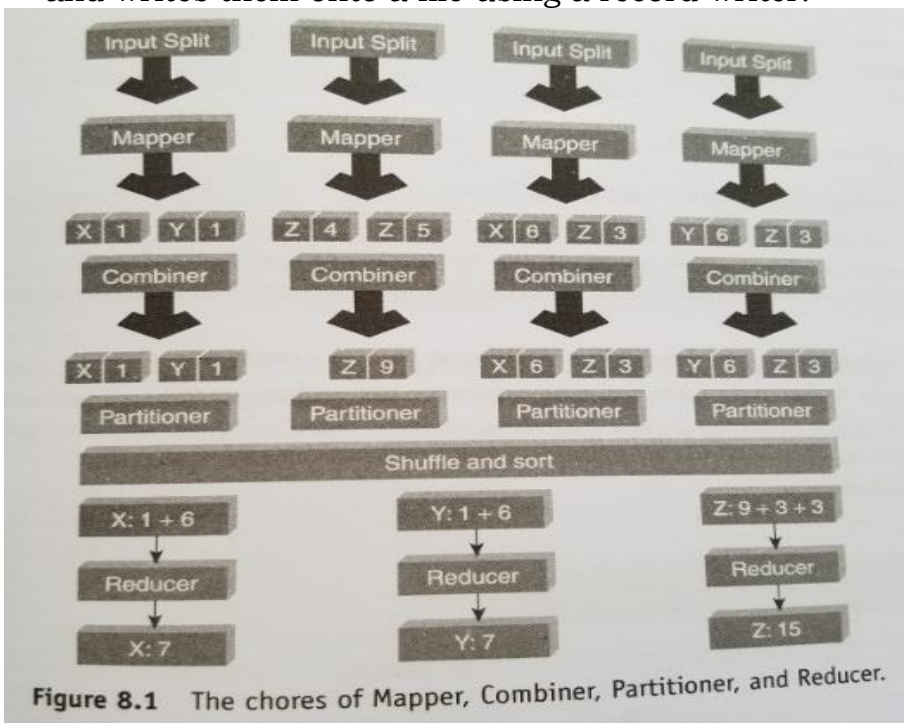


Figure 8.1 The chores of Mapper, Combiner, Partitioner, and Reducer.

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

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

We can specify compression format in the Driver program as 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.

Q) Write a MapReuduce program for WordCount problem.

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount
{
        public static class WCMapper extends  Mapper <Object, Text, Text, IntWritable>

        {
                final static IntWritable one = new IntWritable(1);
                Text word = new Text();
               public void map(Object key, Text value, Context context) throws IOException, InterruptedException {

                        StringTokenizer itr = new tringTokenizer(value.toString());
                        while (itr.hasMoreTokens()) {
                                word.set(itr.nextToken());
                                context.write(word, one);
                        }
                }
        }
```

```
public static class WCReducer extends Reducer<Text, IntWritable, Text,
IntWritable>  {
            private IntWritable result = new IntWritable();
            public void reduce(Text key, Iterable<IntWritable> values,
Context context ) throws IOException, InterruptedException {

                  int sum = 0;
                    for (IntWritable val : values) {
                          sum += val.get();
                      }
                  result.set(sum);
                  context.write(key, result);
            }
      }

public static void main(String[] args) throws Exception {
Configuration conf = new Configuration();
Job job = Job.getInstance(conf, "word count");
job.setJarByClass(WordCount.class);
job.setMapperClass(WCMapper.class);
job.setReducerClass(WCReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

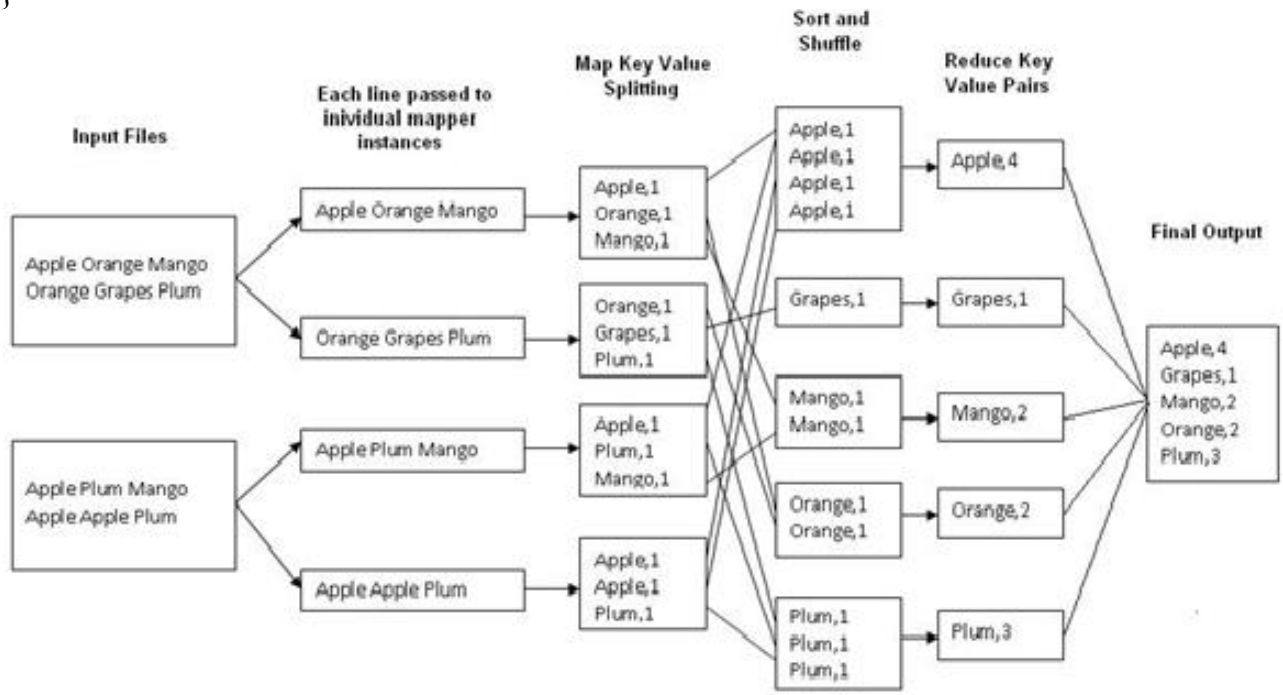


Fig. MapReduce paradigm for WordCount

Q) Write a MapReduce program to calculate employee salary of each department in the university.
I/P:
001,it,10000
002,cse,20000
003,it,30000

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class Salary
{

      public static class SalaryMapper  extends Mapper <LongWritable,
Text, Text,
IntWritable>
  {

   public void map(LongWritable key, Text value, Context context) throws
IOException,
    InterruptedException
   {

        String[] token = value.toString().split(",");
        int s = Integer.parseInt(token[2]);
        IntWritable sal = new IntWritable();
        sal.set(s);
        context.write(new Text(token[1]),sal);

       }
  }
```

```
public static class SalaryReducer extends Reducer<Text, IntWritable, Text,
IntWritable>
  {
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values, Context
context ) throws
IOException, InterruptedException
            {

           int sum = 0;
           for (IntWritable val : values) {
                 sum += val.get();
            }
        result.set(sum);
         context.write(key,result);
        }
}

public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "Salary");

        job.setJarByClass(Salary.class);
        job.setMapperClass(SalaryMapper.class);
        job.setReducerClass(SalaryReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        System.exit(job.waitForCompletion(true) ? 0 : 1);
        }

}
```

Q) Write a user define partitioner class for WordCount problem.

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.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 = 1;break;
                        case 'C': partitionNumber = 1;break;
                        case 'D': partitionNumber = 1;break;
                        case 'E': partitionNumber = 1;break;
                        case 'F': partitionNumber = 1;break;
                        case 'G': partitionNumber = 1;break;
                        case 'H': partitionNumber = 1;break;
                        case 'I': partitionNumber = 1;break;
                        case 'J': partitionNumber = 1;break;
                        case 'K': partitionNumber = 1;break;
                        case 'L': partitionNumber = 1;break;
                        case 'M': partitionNumber = 1;break;
                        case 'N': partitionNumber = 1;break;
                        case 'O': partitionNumber = 1;break;
                        case 'P': partitionNumber = 1;break;
                        case 'Q': partitionNumber = 1;break;
                        case 'R': partitionNumber = 1;break;
                        case 'S': partitionNumber = 1;break;
                        case 'T': partitionNumber = 1;break;
                        case 'U': partitionNumber = 1;break;
                        case 'V': partitionNumber = 1;break;
                        case 'W': partitionNumber = 1;break;
                        case 'X': partitionNumber = 1;break;
                        case 'Y': partitionNumber = 1;break;
                        case 'Z': partitionNumber = 1;break;
                default: partitionNumber = 0;break;
                }
        return partionNumber;
        }
}
```

In the drive program set the partioner class as shown below:

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

Q) Write a MapReuduce program for sorting following data according to name.

Input:

001,chp
002,vr
003,pnr
004,prp

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
```

```
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class Sort{
	public static class SortMapper extends Mapper
<LongWritable,Text,Text,Text> {
		protected void map(LongWritable key, Text value, Context
context) throws IOException,InterruptedException{

	String[] token = value.toString().split(",");
context.write(new Text(token[1]),new Text(token[0]+"-"+token[1]));
		}
	}

	public static class SortReducer extends
Reducer<Text,Text,NullWritable,Text>{
		public void reduce(Text key, Iterable<Text> values, Context
context) throws IOException,InterruptedException{

			for(Text details:values){
				context.write(NullWritable.get(),details);

			}

		}

	}

public static void main(String args[]) throws
IOException,InterruptedException,ClassNotFoundException{

	Configuration conf = new Configuration();
	Job job = new Job(conf);
	job.setJarByClass(Sort.class);
	job.setMapperClass(SortMapper.class);
	job.setReducerClass(SortReducer.class);
	job.setOutputKeyClass(Text.class);
	job.setOutputValueClass(Text.class);
	FileInputFormat.addInputPath(job, new Path(args[0]));
	FileOutputFormat.setOutputPath(job, new Path(args[1]));
	System.exit(job.waitForCompletion(true)?0:1);
  }
}
```

Q) Write a MapReduce program to arrange the data on user-id, then within the user id sort them in increasing order of the page count.

Input:

001,3,www.turorialspoint.com
001,4,www.javapoint.com
002,5,www.javapoint.com
003,2,www.analyticsvidhya.com

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
public class Sort{
        public static class SortMapper extends
Mapper<LongWritable,Text,Text,Text>{
        Text comp_key =  new Text();
                protected void map(LongWritable key, Text value, Context
context) throws IOException,InterruptedException{

        String[] token = value.toString().split(",");
        comp_key.set(token[0]);
        comp_key.set(token[1]);
context.write(comp_key,new Text(token[0]+"-"+token[1]+"-"+token[2]));
                }
        }

        public static class SortReducer extends
Reducer<Text,Text,NullWritable,Text>{
                public void reduce(Text key, Iterable<Text> values, Context
context) throws IOException,InterruptedException{

                for(Text details:values){
                        context.write(NullWritable.get(),details);
                }
           }
 }

public static void main(String args[]) throws
IOException,InterruptedException,ClassNotFoundException{
        Configuration conf = new Configuration();
        Job job = new Job(conf);
        job.setJarByClass(Sort.class);
```

```
        job.setMapperClass(SortMapper.class);
        job.setReducerClass(SortReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

o/p:

```
003-2-www.analyticsvidhya.com
001-3-www.turorialspoint.com
001-4-www.javapoint.com
002-5-www.javapoint.com
```

Q) Write a MapReduce program to search an employee name in the following data:

Input:

001,chp
002,vr
003,pnr
004,prp

```
        import java.io.IOException;
        import org.apache.hadoop.conf.Configuration;
        import org.apache.hadoop.fs.Path;
        import org.apache.hadoop.io.IntWritable;
        import org.apache.hadoop.io.LongWritable;
        import org.apache.hadoop.io.NullWritable;
        import org.apache.hadoop.io.Text;
        import org.apache.hadoop.mapreduce.Job;
        import org.apache.hadoop.mapreduce.Mapper;
        import org.apache.hadoop.mapreduce.Reducer;
        import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
        import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
        import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
        import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
        import org.apache.hadoop.mapreduce.InputSplit;
        import org.apache.hadoop.mapreduce.lib.input.FileSplit;

        public class Search{
        public static class SearchMapper extends Mapper<LongWritable,
Text, Text, Text>{
                static String keyword;
                static int pos=0;
protected void setup(Context context) throws  IOException,
InterruptedException {
                Configuration config = context.getConfiguration();
                keyword = config.get("keyword");
                }
```

```
protected void map(LongWritable key, Text value, Context  context) throws
IOException,InterruptedException{
            InputSplit in = context.getInputSplit();
            FileSplit f = (FileSplit)in;
                  String fileName = f.getPath().getName();
            Integer wordPos;
            pos++;
            if(value.toString().contains(keyword)){
               wordPos = value.find(keyword);
               context.write(value,  new  Text(fileName  +  ","+new
IntWritable(pos).toString()+","+wordPos.toString()));
          }
         }
      }

      public static class SearchReducer extends Reducer
<Text,Text,Text,Text>{
            public void reduce(Text key, Text value, Context context) throws
IOException,InterruptedException{
            context.write(key,value);
            }
         }
      public  static  void  main(String  args[])  throws
IOException,InterruptedException,ClassNotFoundException{
          Configuration conf = new Configuration();
          Job job = new Job(conf);
          job.setJarByClass(Search.class);
          job.setMapperClass(SearchMapper.class);
          job.setReducerClass(SearchReducer.class);
          job.setOutputKeyClass(Text.class);
          job.setOutputValueClass(Text.class);
          job.setInputFormatClass(TextInputFormat.class);
          job.setOutputFormatClass(TextOutputFormat.class);
          job.setNumReduceTasks(1);
          job.getConfiguration().set("keyword","chp");
          FileInputFormat.addInputPath(job, new Path(args[0]));
          FileOutputFormat.setOutputPath(job, new Path(args[1]));
          System.exit(job.waitForCompletion(true)?0:1);
        }
      }
```

Q) What are the Real time applications using MapReduce Programming?

- Social networks
- Media and Entertainment
- Health Care
- Business
- Banking
- Stock Market
- Weather Forecasting