Big Data Analytics CSF443
UNIT – I (Part – II)
INTRODUCTION TO BIG DATA AND HADOOP

HADOOP

History of Hadoop:

- Hadoop is an **open-source software** framework for storing and processing large datasets ranging in size from **gigabytes** to **petabytes**.
- Hadoop was developed at the Apache Software Foundation in 2005.
- It is written in Java.
- The traditional approach like RDBMS is not sufficient due to the heterogeneity of the data.
- So Hadoop comes as the solution to the problem of big data i.e. storing and processing the big data with some extra capabilities.
- Its co-founder Doug Cutting named it on his son's toy elephant.
- There are mainly two components of Hadoop which are :
    - **Hadoop Distributed File System (HDFS)**
    - **Yet Another Resource Negotiator(YARN).**

In April 2006 Hadoop 0.1.0 was released.

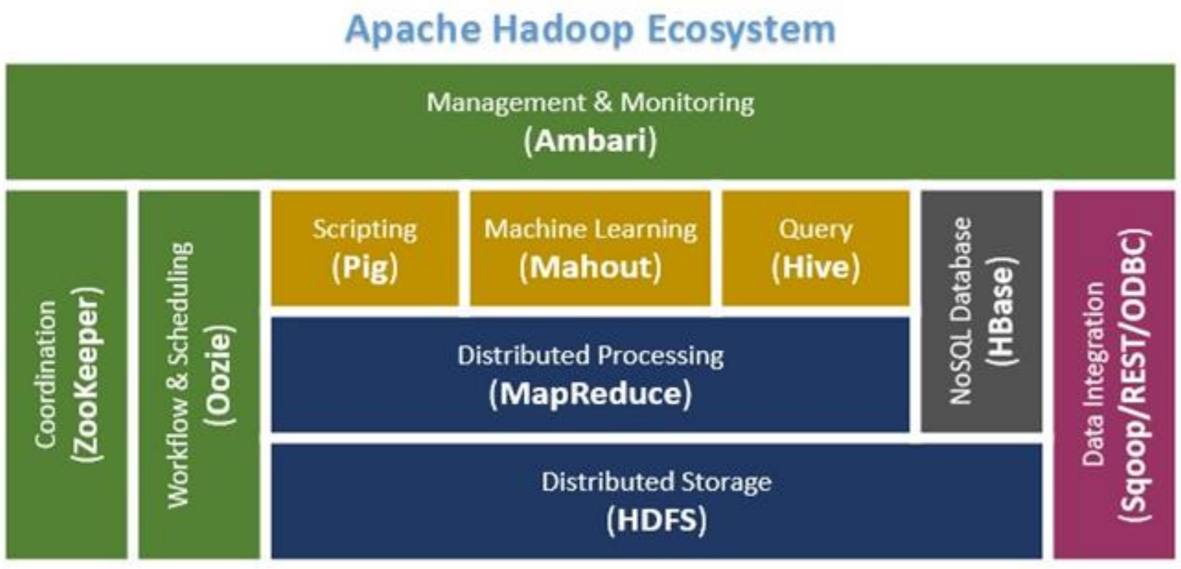
Apache Hadoop:

- Hadoop is an **open-source software** framework for storing and processing large datasets ranging in size from **gigabytes** to **petabytes**.
- Hadoop was developed at the Apache Software Foundation in 2005.
- It is written in Java.
- Hadoop is designed to scale up from a single server to thousands of machines, each offering local computation and storage.
- Applications built using HADOOP are run on large data sets distributed across clusters of commodity computers.
- Commodity computers are cheap and widely available, these are useful for achieving greater computational power at a low cost.
- In Hadoop data resides in a distributed file system which is called a Hadoop Distributed File system.

Hadoop Distributed File System :

- In Hadoop data resides in a distributed file system which is called a **Hadoop Distributed File system**.
- HDFS splits files into blocks and sends them across various nodes in form of large clusters.
- The Hadoop Distributed File System (HDFS) is based on the Google File System (GFS) and provides a distributed file system that is designed to run on commodity hardware.
- Commodity hardware is cheap and widely available, these are useful for achieving greater computational power at a low cost.
- It is highly fault-tolerant and is designed to be deployed on low-cost hardware.
- It provides high throughput access to application data and is suitable for applications having large datasets.
- Hadoop framework includes the following two modules:
    - Hadoop Common: These are Java libraries and utilities required by other Hadoop modules.
    - Hadoop YARN: This is a framework for job scheduling and cluster resource management.

Hadoop Ecosystem And Components:


Apache Hadoop Ecosystem
Management & Monitoring (Ambari)
Coordination (ZooKeeper)
Workflow & Scheduling (Oozie)
Scripting (Pig)
Machine Learning (Mahout)
Query (Hive)
NoSQL Database (HBase)
Data Integration (Sqoop/REST/ODBC)
Distributed Processing (MapReduce)
Distributed Storage (HDFS)

There are three components of Hadoop :

Hadoop HDFS -

- Hadoop Distributed File System (HDFS) is the storage unit of Hadoop.
- HDFS splits files into blocks and sends them across various nodes in form of large clusters.
- It is highly fault-tolerant and is designed to be deployed on low-cost hardware.
- It provides high throughput access to application data and is suitable for applications having large datasets

Hadoop MapReduce -

- Hadoop MapReduce is the processing unit of Hadoop.
- MapReduce is a computational model and software framework for writing applications that are run on Hadoop.
- These MapReduce programs are capable of processing enormous data in parallel on large clusters of computation nodes.

Hadoop YARN –

- Hadoop YARN is a resource management unit of Hadoop.
- This is a framework for job scheduling and cluster resource management.
- YARN helps to open up Hadoop by allowing to process and run data for batch processing, stream processing, interactive processing and graph processing which are stored in HDFS.
- It helps to run different types of distributed applications other than MapReduce.

<u>DATA FORMAT</u>:

- A data/file format defines how information is stored in HDFS.
- Hadoop does not have a default file format and the choice of a format depends on its use.
- The big problem in the performance of applications that use HDFS is the information search time and the writing time.
- Managing the processing and storage of large volumes of information is very complex that's why a certain data format is required.

- The choice of an appropriate file format can produce the following benefits:

- Optimum writing time
- Optimum reading time
- File divisibility
- Adaptive scheme and compression support

Some of the most commonly used formats of the Hadoop ecosystem are :

● **Text/CSV:** A plain text file or CSV is the most common format both outside and within the Hadoop ecosystem.
● **SequenceFile:** The SequenceFile format stores the data in binary format, this format accepts compression but does not store metadata.
● **Avro:** Avro is a row-based storage format. This format includes the definition of the scheme of your data in JSON format. Avro allows block compression along with its divisibility, making it a good choice for most cases when using Hadoop.
● **Parquet:** Parquet is a column-based binary storage format that can store nested data structures. This format is very efficient in terms of disk input/output operations when the necessary columns to be used are specified.
● **RCFile (Record Columnar File):** RCFile is a columnar format that divides data into groups of rows, and inside it, data is stored in columns.
● **ORC (Optimized Row Columnar):** ORC is considered an evolution of the RCFile format and has all its benefits alongside some improvements such as better compression, allowing faster queries.

Analysing Data with Hadoop:

- While the MapReduce programming model is at the heart of Hadoop, it is low-level and as such becomes an unproductive way for developers to write complex analysis jobs.
- To increase developer productivity, several higher-level languages and APIs have been created that abstract away the low-level details of the MapReduce programming model.
- There are several choices available for writing data analysis jobs.

- The Hive and Pig projects are popular choices that provide SQL-like and procedural data flow-like languages, respectively.
- HBase is also a popular way to store and analyze data in HDFS. It is a column-oriented database, and unlike MapReduce, provides random read and write access to data with low latency.
- MapReduce jobs can read and write data in HBase's table format, but data processing is often done via HBase's own client API.

Scaling In Vs Scaling Out:

Once a decision has been made for data scaling, the specific scaling approach must be chosen.
There are two commonly used types of data scaling:

1. Up
2. Out

Scaling up, or vertical scaling:

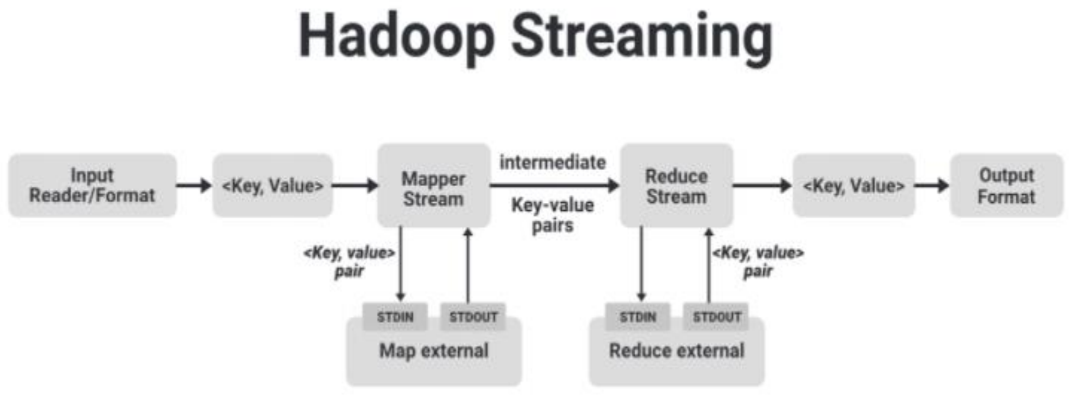
- It involves obtaining a faster server with more powerful processors and more memory.
- This solution uses less network hardware and consumes less power; but ultimately.
- For many platforms, it may only provide a short-term fix, especially if continued growth is expected.

Scaling out, or horizontal scaling:

- It involves adding servers for parallel computing.
- The scale-out technique is a long-term solution, as more and more servers may be added when needed.
- But going from one monolithic system to this type of cluster may be difficult, although extremely effective solution.

Hadoop Streaming:

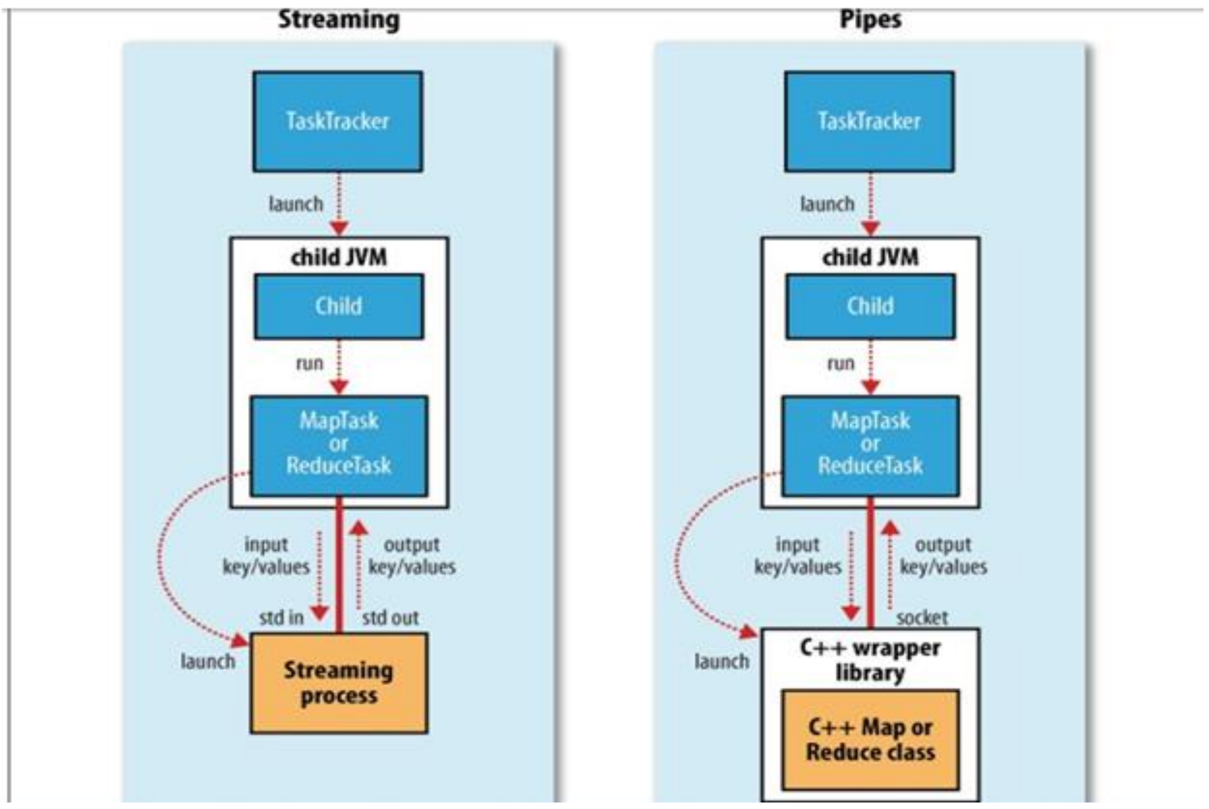
- It is a feature that comes with a Hadoop distribution that allows developers or programmers to write the Map-Reduce program using different programming languages like Ruby, Perl, Python, C++, etc.
- We can use any language that can read from the standard input (STDIN) like keyboard input and all and write using standard output (STDOUT).
- Although Hadoop Framework is completely written in java programs for Hadoop do not necessarily need to code in Java programming language.
- In the diagram,



- We have an **Input Reader** which is responsible for reading the input data and produces the list of key-value pairs. We can read data in .csv format, in delimiter format, from a database table, image data(.jpg, .png), audio data etc.
- This list of key-value pairs is fed to the **Map phase** and Mapper will work on each of these key-value pair of each pixel and generate some intermediate key-value pairs.
- After shuffling and sorting, the intermediate key-value pairs are fed to the **Reducer**: then the final output produced by the reducer will be written to the HDFS. These are how a simple Map-Reduce job works.

Hadoop Pipes:

- Hadoop Pipes is the name of the C++ interface to Hadoop MapReduce.
- Unlike Streaming, this uses standard input and output to communicate with the map and reduce code.
- Pipes uses sockets as the channel over which the task tracker communicates with the process running the C++ map or reduce function.

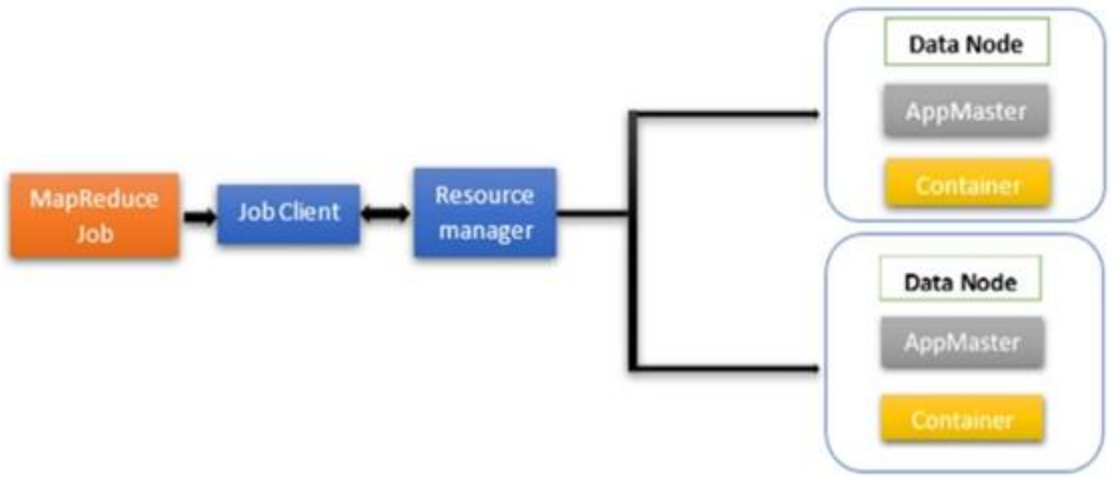


<u>Map Reduce Framework and Basics</u>:

- MapReduce is a software framework for processing data sets in a distributed fashion over several machines.
- Prior to Hadoop 2.0, MapReduce was the only way to process data in Hadoop.
- A MapReduce job usually splits the input data set into independent chunks, which are processed by the map tasks in a completely parallel manner.
- The core idea behind MapReduce is mapping your data set into a collection of < key, value> pairs, and then *reducing* overall pairs with the same key.
- The framework sorts the outputs of the maps, which are then inputted to the reduced tasks.
- Both the input and the output of the job are stored in a file system.
- The framework takes care of scheduling tasks, monitors them, and re-executes the failed tasks.
- The overall concept is simple :

1. Almost all data can be mapped into pairs somehow, and
2. Your keys and values may be of any type: strings, integers, dummy types and, of course, pairs themselves.

<u>How Map Reduce Works</u>:



Map Reduce contains two core components:

1. Mapper component
2. Reducer component

- Uses master-slave architecture.
- Storing data in HDFS is low cost, fault-tolerant, and easily scalable.
- MapReduce integrates with HDFS to provide the exact same benefits for parallel data processing.
- Sends computations where the data is stored on local disks.
- Programming model or framework for distributed computing.
- It hides complex "housekeeping" tasks from you as a developer.

<u>Developing A Map-Reduce Application</u>:

- Writing a program in MapReduce follows a certain pattern.
- You start by writing your map and reduce functions, ideally with unit tests to make sure they do what you expect.
- Then you write a driver program to run a job, which can run from your IDE using a small subset of the data to check that it is working.

- If it fails, you can use your IDE's debugger to find the source of the problem.
- When the program runs as expected against the small dataset, you are ready to unleash it on a cluster.
- Running against the full dataset is likely to expose some more issues, which you can fix by expanding your tests and altering your mapper or reducer to handle the new cases.

After the program is working, you may wish to do some tuning :

  - First by running through some standard checks for making MapReduce programs faster
  - Second by doing task profiling.

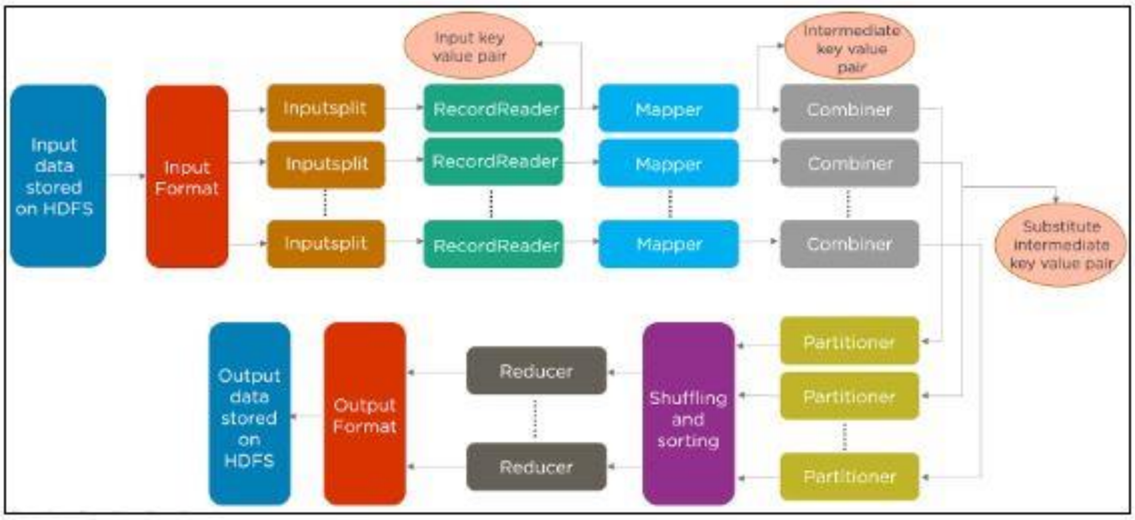
- Profiling distributed programs are not easy, but Hadoop has hooks to aid in the process.
- Before we start writing a MapReduce program, we need to set up and configure the development environment.
- Components in Hadoop are configured using Hadoop's own configuration API.
- An instance of the Configuration class represents a collection of configuration properties and their values.
- Each property is named by a String, and the type of a value may be one of several, including Java primitives such as boolean, int, long, and float and other useful types such as String, Class, and java.io.File; and collections of Strings.

<u>How Does the Hadoop MapReduce Algorithm Work?</u>

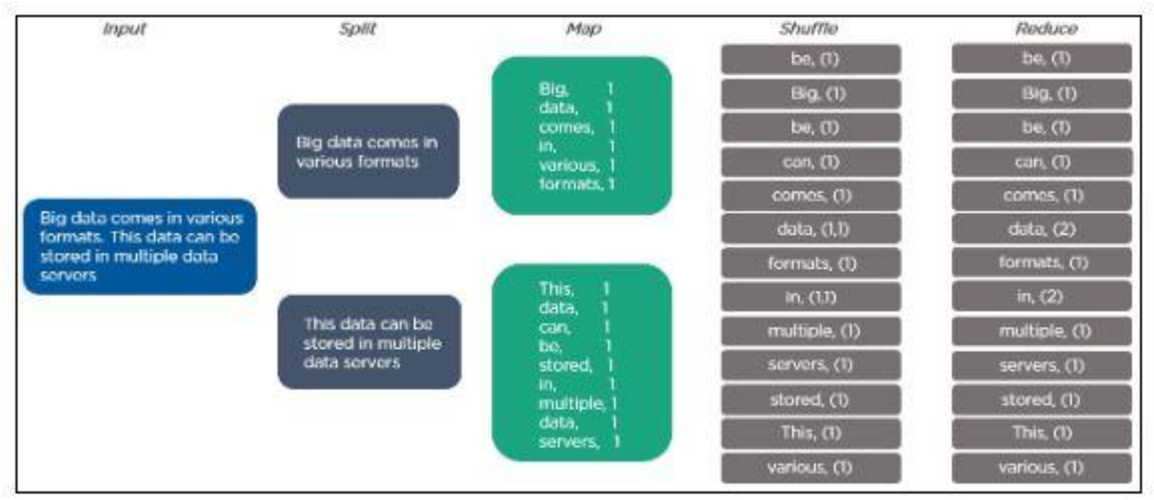
- Let's understand how the MapReduce algorithm works by understanding the job execution flow in detail.

- The input data to process using the MapReduce task is stored in input files that reside on HDFS.
- The input format defines the input specification and how the input files are split and read.

- The input split logically represents the data to be processed by an individual Mapper.
- The record reader communicates with the input split and converts the data into key-value pairs suitable for reading by the mapper (k, v).
- The mapper class processes input records from RecordReader and generates intermediate key-value pairs (k’, v’). Conditional logic is applied to ‘n’ number of data blocks present across various data nodes.
- The combiner is a mini reducer. For every combiner, there is one mapper. It is used to optimize the performance of MapReduce jobs.
- The partitioner decides how outputs from the combiner are sent to the reducers.
- The output of the partitioner is shuffled and sorted. All the duplicate values are removed, and different values are grouped based on similar keys. This output is fed as input to the reducer. All the intermediate values for the intermediate keys are combined into a list by the reducer called tuples.
- The record writer writes these output key-value pairs from the reducer to the output files. The output data is stored on the HDFS.

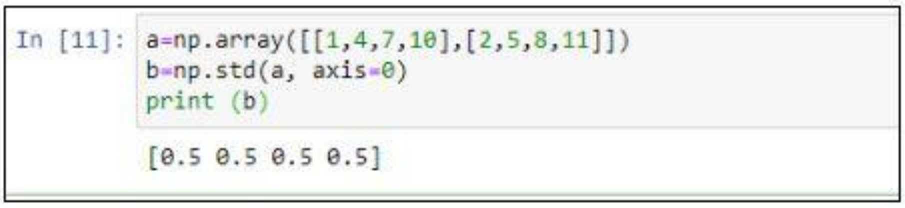


- Shown below is a MapReduce example to count the frequency of each word in a given input text. Our input text is, “Big data comes in various formats. This data can be stored in multiple data servers.”



MapReduce Example to Analyze Call Data Records

- Shown below is a sample data of call records. It has the information regarding phone numbers from which the call was made, and to which phone number it was made. The data also gives information about the total duration of each call. It also tells you if the call made was a local (0) or an STD call (1).

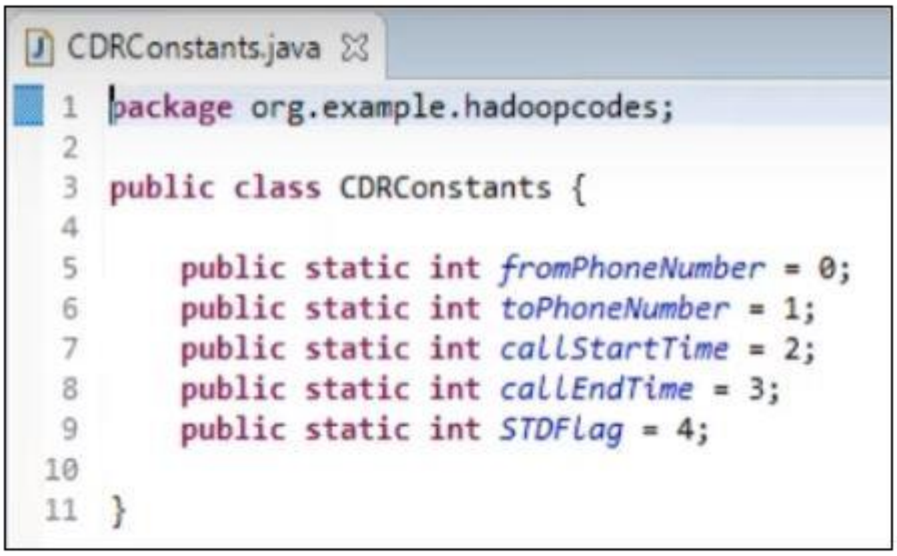


```
In [11]: a=np.array([[1,4,7,10],[2,5,8,11]])
         b=np.std(a, axis=0)
         print (b)

         [0.5 0.5 0.5 0.5]
```

- We'll use this data to perform certain operations with the help of a MapReduce algorithm. One of the operations you can perform is to find all the phone numbers that made more than 60 minutes of STD calls.

- We'll use Java programming language to do this task.

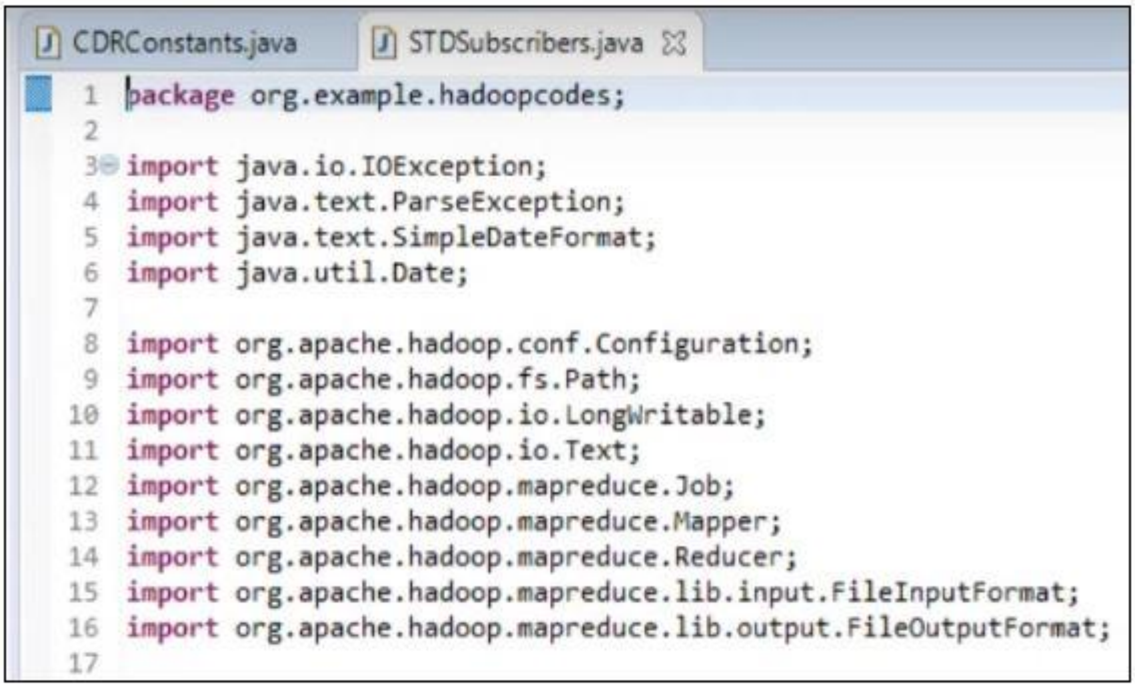
1. Let’s first declare our constants for the fields.



CDRConstants.java

```java
package org.example.hadoopcodes;

public class CDRConstants {

    public static int fromPhoneNumber = 0;
    public static int toPhoneNumber = 1;
    public static int callStartTime = 2;
    public static int callEndTime = 3;
    public static int STDFlag = 4;

}
```

2. Import all the necessary packages to make sure we use the classes in the right way.



CDRConstants.java STDSubscribers.java

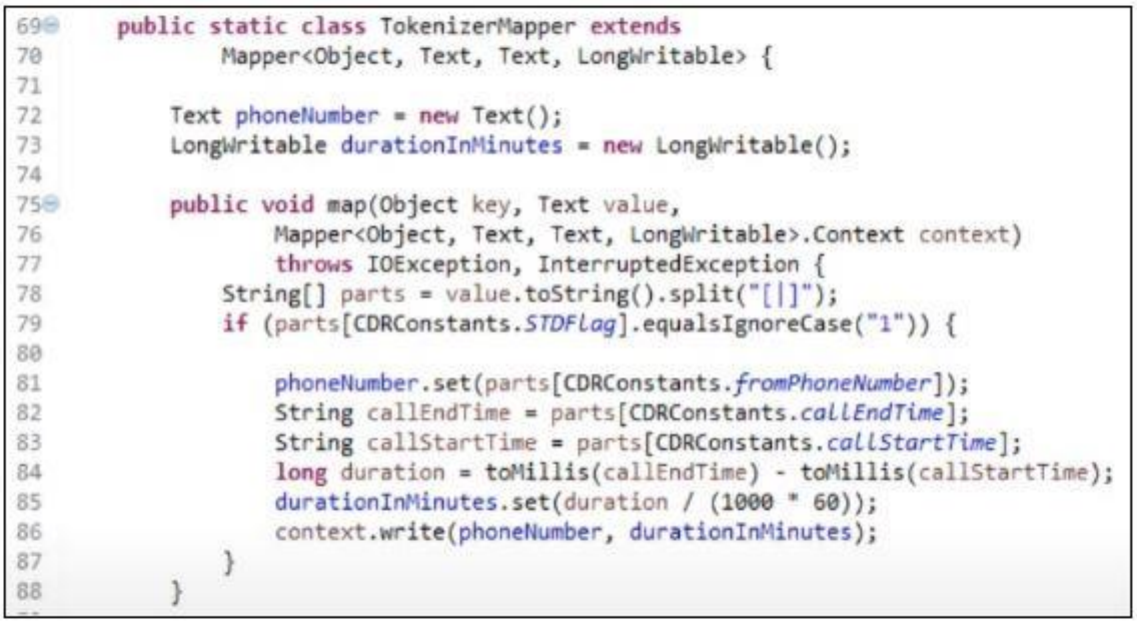
```java
package org.example.hadoopcodes;

import java.io.IOException;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
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;

```

3. The order of the driver, mapper, and reducer class does not matter. So, let’s create a mapper that will do the map task.

- We will create a TokenizerMapper that will extend our Mapper class. It accepts the desired data types (line 69-70).

- We'll assign phone numbers and the duration of the calls in minutes (line 72-73).
- The map task works on a string, and it breaks it into individual elements based on a delimiter (line 75-78).
- Then, we'll check if the string that we are looking for has an STD flag (line 79).
- We will then set the phone numbers using the constant class and find the duration (line 81-83).
- Finally, we'll extract the phone numbers and the duration of the call made by a particular phone number (line 84-86).

This mapper class will return an intermediate output, which would then be sorted and shuffled and passed on to the reducer.

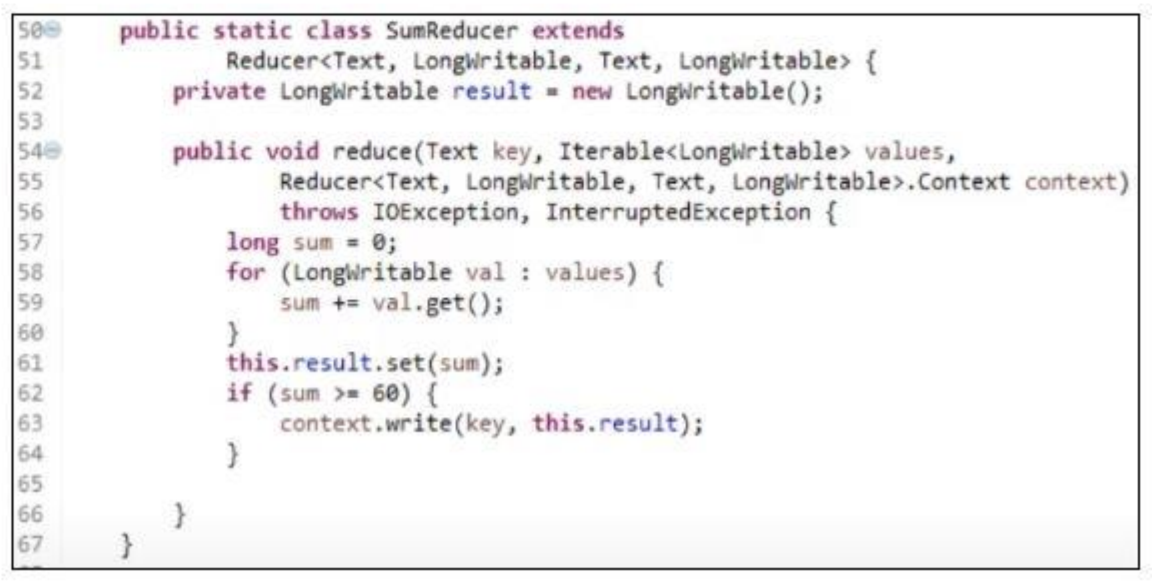


```
69      public static class TokenizerMapper extends
70              Mapper<Object, Text, Text, LongWritable> {
71
72          Text phoneNumber = new Text();
73          LongWritable durationInMinutes = new LongWritable();
74
75          public void map(Object key, Text value,
76                  Mapper<Object, Text, Text, LongWritable>.Context context)
77                  throws IOException, InterruptedException {
78              String[] parts = value.toString().split("[|]");
79              if (parts[CDRConstants.STDFlag].equalsIgnoreCase("1")) {
80
81                  phoneNumber.set(parts[CDRConstants.fromPhoneNumber]);
82                  String callEndTime = parts[CDRConstants.callEndTime];
83                  String callStartTime = parts[CDRConstants.callStartTime];
84                  long duration = toMillis(callEndTime) - toMillis(callStartTime);
85                  durationInMinutes.set(duration / (1000 * 60));
86                  context.write(phoneNumber, durationInMinutes);
87              }
88          }
```

4. Next, we define our reducer class.

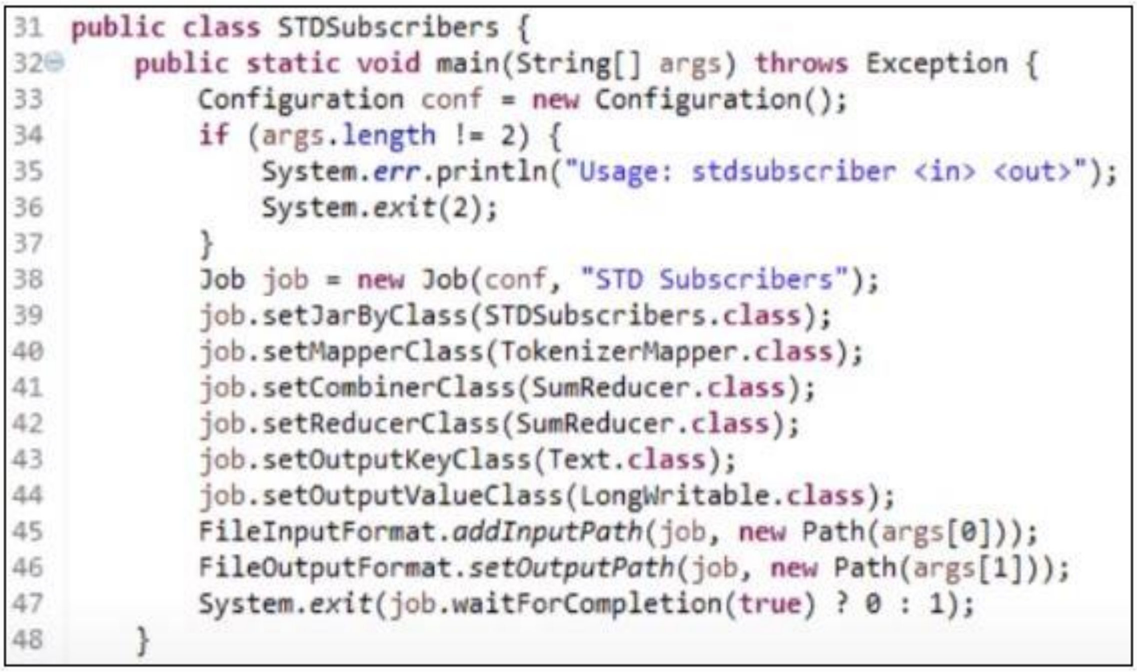
- So, we define our reducer class called SumReducer. The reducer uses the right data types specific to Hadoop MapReduce (line 50-52).

- The reduce (Object, Iterable, Context) method is called for each <key, (collection of values)> in the sorted inputs. The output of the reduce task is written to a RecordWriter via TaskInputOutputContext.write(Object, Object) (line 54-56).
- It looks into all the keys and values. Wherever it finds that the keys that are repeating and the duration is more than 60 minutes, it would return an aggregated result (line 57-63).



```
50      public static class SumReducer extends
51              Reducer<Text, LongWritable, Text, LongWritable> {
52          private LongWritable result = new LongWritable();
53
54          public void reduce(Text key, Iterable<LongWritable> values,
55                  Reducer<Text, LongWritable, Text, LongWritable>.Context context)
56                  throws IOException, InterruptedException {
57              long sum = 0;
58              for (LongWritable val : values) {
59                  sum += val.get();
60              }
61              this.result.set(sum);
62              if (sum >= 60) {
63                  context.write(key, this.result);
64              }
65
66          }
67      }
```

5. The driver class has all the job configurations, mapper, reducer, and also a combiner class. It is responsible for setting up a MapReduce job to run in the Hadoop cluster. You can specify the names of Mapper and Reducer Classes long with data types and their respective job names.



```java
public class STDSubscribers {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        if (args.length != 2) {
            System.err.println("Usage: stdsubscriber <in> <out>");
            System.exit(2);
        }
        Job job = new Job(conf, "STD Subscribers");
        job.setJarByClass(STDSubscribers.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(SumReducer.class);
        job.setReducerClass(SumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(LongWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
```

6. Now, package the files as .jar and transfer it to the Hadoop cluster and run it on top of YARN.

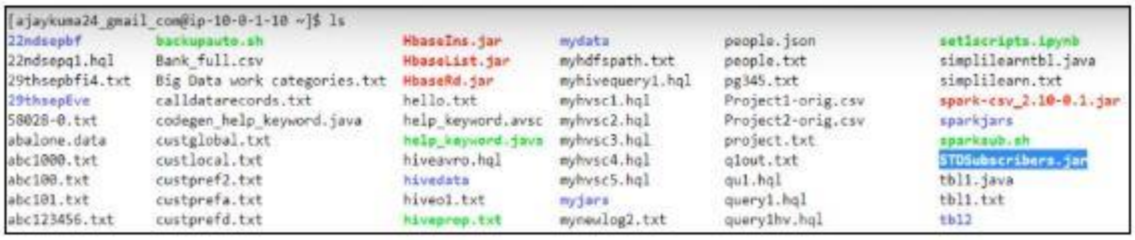
You can locate your call records file using hdfs dfs -ls “Location of the file”

7. Now, we’ll input the call records file for processing. Use the command below to locate the file and give the class name, along with another file location to save the output.

hadoop jar STDSubscribers.jar
org.example.hadoopcodes.STDSubscribers
sampleMRIn/calldatarecords.txt sampleMROutput-2

8. Once you run the above command successfully, you can see the output by checking the directory.

```
hdfs dfs -cat sampleMROutput-2/part-r-00000
```



```
[ajaykuma24_gmail_com@ip-10-0-1-10 ~]$ ls
22ndsepbf         backupauto.sh                 HbaseIns.jar       mydata              people.json               set1scripts.ipynb
22ndsepq1.hql     Bank_full.csv                 HbaseList.jar      myhdfspath.txt      people.txt                simplilearntbl.java
29thsepbfi4.txt   Big Data work categories.txt  HbaseRd.jar        myhivequery1.hql    pg345.txt                 simplilearn.txt
29thsepEve        calldatarecords.txt           hello.txt          myhvsc1.hql         Project1-orig.csv         spark-csv_2.10-0.1.jar
58028-0.txt       codegen_help_keyword.java     help_keyword.avsc  myhvsc2.hql         Project2-orig.csv         sparkjars
abalone.data      custglobal.txt                help_keyword.java  myhvsc3.hql         project.txt               sparksub.sh
abc1000.txt       custlocal.txt                 hiveavro.hql       myhvsc4.hql         q1out.txt                 STDSubscribers.jar
abc100.txt        custpref2.txt                 hivedata           myhvsc5.hql         qu1.hql                   tbl1.java
abc101.txt        custprefa.txt                 hiveo1.txt         myjars              query1.hql                tbl1.txt
abc123456.txt     custprefd.txt                 hiveprep.txt       mynewlog2.txt       query1hv.hql              tbl2
```