UNIT-II

ADOOP FEATURES:

'eliability:

- ➢ When machines are working in one after other, if one of the machines fails, another machine will take over the responsibility and work in a reliable and fault tolerant fashion.
- ➢ Hadoop infrastructure has inbuilt fault tolerance features and hence, Hadoop is highly reliable

Lonomical:

- ➢ Hadoop uses commodity hardware (like your PC, laptop). For example, in a small Hadoop cluster, all your DataNodes can have normal configurations like 8-16 GB RAM with 5-10 TB hard disk and Xeon processors.I
- ➢ If I would have used hardware-based RAID with Oracle for the same purpose, I would end up spending 5 times more at least.
- ➢ So, the cost of ownership of a Hadoop-based project is pretty minimized. It is easier to maintain the Hadoop environment and is economical as well.
- ➢ Also, Hadoop is an open source software and hence there is no licensing cost.

Scalability:

- ➢ Hadoop has the inbuilt capability of integrating seamlessly with cloud-based services.
- ➢ If you are installing Hadoop on a cloud, you don't need to worry about the scalability factor because you can go ahead and procure more hardware and expand your setup within minutes whenever required.

Flexibility:

- Hadoop is very flexible in terms of ability to deal with all kinds of data. Data can be of any kind and Hadoop can store and process them all, whether it is structured, semi-structured or unstructured data

These 4 characteristics make Hadoop a front-runner as a solution to Big Data challenges.

HADOOP HDFS COMMANDS:

Hadoop file system shell commands are used to perform various Hadoop HDFS Operations and in order to manage the fiels present on HDFS clusters. All the Hadoop shell commands are invoked by the bin/hdfs script.

1. version: This command is used to print the hadoop version.
   Syntax: hdfs dfs version
2. mkdir: This command takes path URI as an argument and creates directories.
   Syntax: hdfs dfs –mkdir URL
3. ls: This command displays a list of the contents of a directory specified by the path provided by the user. It displays names, permissions, owner, size and modification date for each entry.
   Syntax: hdfs dfs –ls URL
4. put: This command copies the file or directory from the local file system to the destination with in the DFS.
   Syntax: hdfs dfs –put <local-source> <destination>
5. copyFromLocal: This command is similar to put command, but the source is restricted to a local file reference.
   Syntax: hdfs dfs –copyFromLocal <local source> <destination>
6. get: This command copies the file or directory in HDFS identified by the source to the local file system path identified by local destination.
   Syntax: hdfs dfs –get <source> <local-destination>

7. copyToLocal: This command is similar to get command, but the destination is restricted to a local file reference.
   Syntax: hdfs dfs –copyToLocal <source> <local-destination>
8. cat: This command displays the contents of the file specified on console or stdout.
   Syntax: hdfs dfs –cat <URL of file>
9. mv: This command moves the file or directory indicated by the source to destination within HDFS
   Syntax: hadoop fs –mv <source> <destination>
10. cp: This command copies the file or directory identified by the source to destination within HDFS
    Syntax: hadoop fs –cp <source> <destination>
11. rm: This command is used to remove the file or empty directory present on the path provided by the user.
    Syntax: hdfs dfs –rm <URL>
12. du: This command shows the disk usage in bytes for all the files present on the path provided by the user.
    Syntax: hdfs dfs –du <URL>

HADOOP MAPREDUCE API:

API alg.

API (Application Programming Interface) provides the information of the classes and their methods that are involved in the operations of MapReduce Programming.

```
import java.io.*;
import java.util.*;

import org.apache.hadoop.fs.Path;        }   all these packages are present in
import org.apache.hadoop.conf.*;         }   hadoop-common.jar file
import org.apache.hadoop.io.*;           }

import org.apache.hadoop.mapreduce.*;                                 }
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;         }  all these packages are present
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;         }  in hadoop-mapreduce-client-
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;       }  core.jar file
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;       }
```

We require the following classes:

1. JobContext Interface
   The JobContext interface is the super interface for all the classes, which defines different jobs in MapReduce. It gives you a read-only view of the job that is provided to the tasks while they are running.
2. Job class
   a. The Job class is the most important class in the MapReduce API.
   b. It allows user to configure the job, submit its execution, and query the state.
   c. The set methods only work until the job is submitted.
3. Mapper class
   a. The Mapper class defines the Map job.
   b. Maps input key-value pairs to a set of intermediate key-value pairs.
   c. Maps are the individual tasks that transform the input records into intermediate records.
   d. The map( ) method is the most prominent method of the Mapper class.
   e. This method is called once for each key-value pair in the input split.
4. Reducer class
   a. The Reducer class defines the Reduce job in MapReduce.
   b. It reduces a set of intermediate values that share a key to a smaller set of values.

c. Reducer implementations can access the Configuration for a job via the JobContext.getConfiguration( ) method.
d. A Reducer has three primary phases – Shuffle , Sort , Reduce.
e. reduce( ) method is the most prominent method of the Reducer class.
f. This method is called once for each key on the collection of key-value pairs.

PROGRAMMING METHODOLOGIES AND PARADIGMS:

Hadoop MapReduce Program for WordCount:

```
import java.io.*;
import java.util.*;

import org.apache.hadoop.io.*;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.fs.Path;

import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;


public class WordCount
{
        public static class Map extends Mapper<LongWritable,Text,Text,IntWritable>
        {
                public void map(LongWritable key, Text value,Context context) throws    Exception
                        {
                        String line = value.toString();
                        StringTokenizer tokenizer = new StringTokenizer(line);
                        while (tokenizer.hasMoreTokens())
                        {
                                value.set(tokenizer.nextToken());
                                context.write(value, new IntWritable(1));
                        }
                }
        }
        public static class Reduce extends Reducer<Text,IntWritable,Text,IntWritable>
        {
                public void reduce(Text key, Iterable<IntWritable> values,Context context) throws Exception
                {
                        int sum=0;
                        for(IntWritable x: values)
                        {
                                sum+=x.get();
                        }
                        context.write(key, new IntWritable(sum));
                }
        }
public static void main(String[] args) throws Exception
{
        Configuration conf= new Configuration();
        Job job = new Job(conf,"My Word Count Program");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}
```

DRIVER – MAPPER – REDUCER:

The entire MapReduce program can be fundamentally divided into three parts.

1. Mapper
2. Reducer
3. Driver

Mapper Code:

```
public static class Map extends Mapper<LongWritable,Text,Text,IntWritable>
{
        public void map(LongWritable key, Text value,Context context) throws    Exception
        {
                String line = value.toString();
                StringTokenizer tokenizer = new StringTokenizer(line);
                while (tokenizer.hasMoreTokens())
                {
                        value.set(tokenizer.nextToken());
                        context.write(value, new IntWritable(1));
                }
        }
}
```

- We have created a class Map that extends the class Mapper which is already defined in the MapReduce Framework.
- We define the data types of input and output key/value pair after the class declaration using angle brackets.
- Both the input and output of the Mapper is a key/value pair.
- Input:
  - The *key* is nothing but the offset of each line in the text file:*LongWritable*
  - The *value* is each individual line: *Text*
- Output:
  - The *key* is the tokenized words: *Text*
  - We have the hardcoded *value* in our case which is 1: *IntWritable*

educer Code:

```
ublic static class Reduce extends Reducer<Text,IntWritable,Text,IntWritable>
	public void reduce(Text key, Iterable<IntWritable> values,Context context) throws Exception
	{
		int sum=0;
		for(IntWritable x: values)
		{
			sum+=x.get();
		}
		context.write(key, new IntWritable(sum));
	}
```

- We have created a class Reduce which extends class Reducer like that of Mapper.
- We define the data types of input and output key/value pair after the class declaration using angle brackets as done for Mapper.
- Both the input and the output of the Reducer is a key-value pair.
- Input:
  - The *key* nothing but those unique words which have been generated after the sorting and shuffling phase: *Text*
  - The *value* is a list of integers corresponding to each key: *IntWritable*
- Output:
  - The *key* is all the unique words present in the input text file: *Text*
  - The *value* is the number of occurrences of each of the unique words: *IntWritable*
- We have aggregated the values present in each of the list corresponding to each key and produced the final answer.

Driver Code:

```
Configuration conf= new Configuration();
Job job = new Job(conf,"My Word Count Program");
job.setJarByClass(WordCount.class);
job.setMapperClass(Map.class);
job.setReducerClass(Reduce.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);
Path outputPath = new Path(args[1]);
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
```

- In the driver class, we set the configuration of our MapReduce job to run in Hadoop.
- We specify the name of the job , the data type of input/output of the mapper and reducer.
- We also specify the names of the mapper and reducer classes.
- The path of the input and output folder is also specified.
- The method setInputFormatClass () is used for specifying that how a Mapper will read the input data or what will be the unit of work. Here, we have chosen TextInputFormat so that single line is read by the mapper at a time from the input text file.
- The main () method is the entry point for the driver. In this method, we instantiate a new Configuration object for the job.

TREAMING:

1. Hadoop can handle large volumes of structured and unstructured data more efficiently than [illegible] traditional enterprise Data Warehouse.
2. It stores the enormous data sets across distributed clusters of computers.
3. Hadoop Streaming uses MapReduce framework which can be used to write applications to process huge amounts of data.
4. Since MapReduce framework is based on Java, how a developer can work on it if he/she does not have experience in Java.
5. Developers can write mapper/reducer applications using their preferred language and without having much knowledge of Java, using Hadoop Streaming rather than switching to new tools or technologies like Pig and Hive.
6. Hadoop Streaming is a utility that comes with the Hadoop distribution.
7. It can be used to execute programs for big data analysis.
8. Hadoop streaming can be performed using languages like Java, Python, PHP, Scala, Perl, UNIX and many more.
9. The utility allows us to create and run Map/Reduce jobs with any executable or as the mapper and/or reducer.

6

The following example illustrate Python code for Mapper.

mapper.py

```
import sys
for line in sys.stdin:
line = line.strip()
words = line.split()
for word in words:
print'%s\t%s' % (word,1)
```

HADOOP INPUTFORMAT AND OUTPUTFORMAT:

INPUTFORMAT:

7

1. Hadoop InputFormat describes the input-specification for execution of the Map-Reduce job
2. InputFormat describes how to split up and read input files.
3. InputFormat is responsible for creating the input splits and dividing them into records.
4. InputFormat selects the files or other objects for input.
5. It also defines the Data splits.
6. It defines both the size of individual Map tasks and its potential execution server.
7. Hadoop InputFormat defines the RecordReader, it is responsible for reading actual records form the input files.
8. The following are the various Mapreduce InputFormat in Hadoop.

FileInputFormat

It is the base class for all file-based InputFormats. FileInputFormat also specifies input directory which has data files location. When we start a MapReduce job execution, FileInputFormat provides a path containing files to read. This InpuFormat will read all files. Then it divides these files into one or more InputSplits

TextInputFormat

It is the default InputFormat. This InputFormat treats each line of each input file as a separate record. It performs no parsing. TextInputFormat is useful for unformatted data or line-based records like log files. Hence,

- Key – It is the byte offset of the beginning of the line within the file (not whole file one split). So it will be unique if combined with the file name.
- Value – It is the contents of the line. It excludes line terminators.

KeyValueTextInputFormat

It is similar to TextInputFormat. This InputFormat also treats each line of input as a separate record. While the difference is that TextInputFormat treats entire line as the value, but the KeyValueTextInputFormat breaks the line itself into key and value by a tab character ('/t'). Hence,

- Key – Everything up to the tab character.
- Value – It is the remaining part of the line after tab character.

SequenceFileInputFormat

It is an InputFormat which reads sequence files. Sequence files are binary files. These files also store sequences of binary key-value pairs. These are block-compressed and provide direct serialization and deserialization of several arbitrary data. Hence, Key & Value both are user-defined.

SequenceFileAsTextInputFormat

It is the variant of SequenceFileInputFormat. This format converts the sequence file key values to Text objects. So, it performs conversion by calling **'tostring()'** on the keys and values. Hence, SequenceFileAsTextInputFormat makes sequence files suitable input for streaming.

SequenceFileAsBinaryInputFormat: By using SequenceFileInputFormat we can extract the sequence file keys and values as an opaque binary object.

NLineInputFormat

It is another form of TextInputFormat where the keys are byte offset of the line. And values are contents of the line. So, each mapper receives a variable number of lines of input with TextInputFormat and KeyValueTextInputFormat. The number depends on the size of the split. If we want our mapper to receive a fixed number of lines of input, then we use NLineInputFormat.

DBInputFormat

This InputFormat reads data from a relational database, using JDBC. It also loads small datasets, perhaps for joining with large datasets from HDFS using MultipleInputs. Hence,

- Key – LongWritables
- Value – DBWritables.

OUTPUTFORMATS:

1) OutputFormat check the output specification for execution of the Map-Reduce job.
2) It describes how RecordWriter implementation is used to write output to output files.
3) RecordWriter in MapReduce job execution writes output key-vaalue pairs from the Reducer phase to output files.
4) The following are various OutputFormat in MapReducing.

TextOutputFormat

The default OutputFormat is TextOutputFormat. It writes (key, value) pairs on individual lines of text files. Its keys and values can be of any type. The reason behind is that TextOutputFormat turns them to string by

alling toString() on them. It separates key-value pair by a tab character. KeyValueTextOutputForma
so used for reading these output text files.

equenceFileOutputFormat

his OutputFormat writes sequences files for its output. SequenceFileInputFormat is also intermedia
rmat use between MapReduce jobs. It serializes arbitrary data types to the file and the correspondi
equenceFileInputFormat will deserialize the file into the same types.

SequenceFileAsBinaryOutputFormat

It is another variant of SequenceFileInputFormat. It also writes keys and values to sequence file in binary
format.

MapFileOutputFormat

It is another form of FileOutputFormat. It also writes output as map files. The framework adds a key in a
MapFile in order. So we need to ensure that reducer emits keys in sorted order.

MultipleOutputs

This format allows writing data to files whose names are derived from the output keys and values

LazyOutputFormat

In MapReduce job execution, FileOutputFormat sometimes create output files, even if they are empty
LazyOutputFormat is also a wrapper OutputFormat.

DBOutputFormat

It is the OutputFormat for writing to relational databases and HBase. This format also sends the reduce
output to a SQL table. It also accepts key-value pairs. In this, the key has a type extending DBwritable