

## HADOOP

- 1) large storage + fast processing.
- 2) commodity Hardware
- 3) cluster

 → storage : HDFS (Hadoop file System)

→ Processing : Map Reduce.

→ Streaming Access Pattern :-

 HDFS : It is a specially designed file system for storing huge data set with cluster of commodity hardware and with streaming access pattern.

→ Streaming Access Pattern :

"Write Once read any number of times"

→ HDFS :-

① normal h/w block size ; 4 KB.

② If 2 KB is used , then remaining 2 KB is wasted

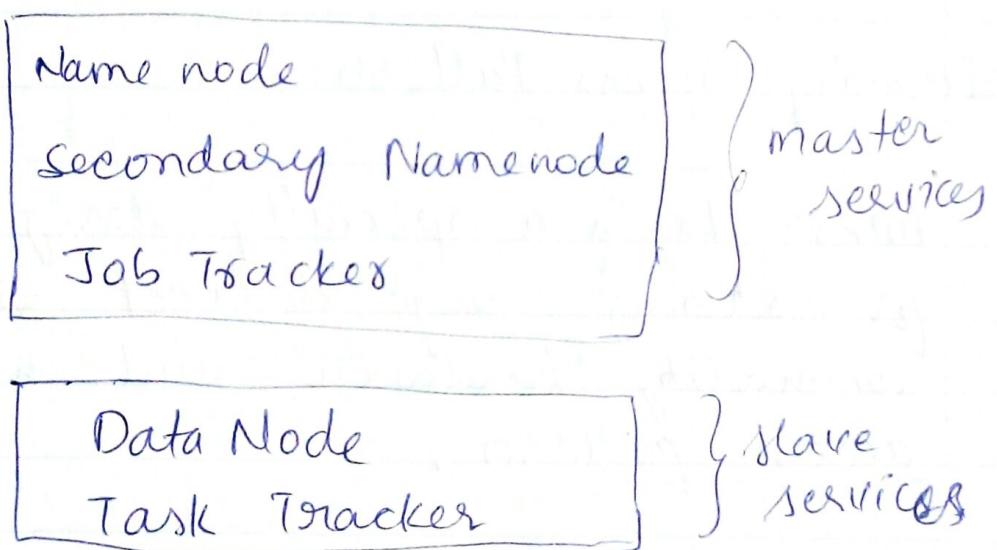
③ To store huge files, in 4 KB blocks & track the blocks, it requires to maintain huge meta-data

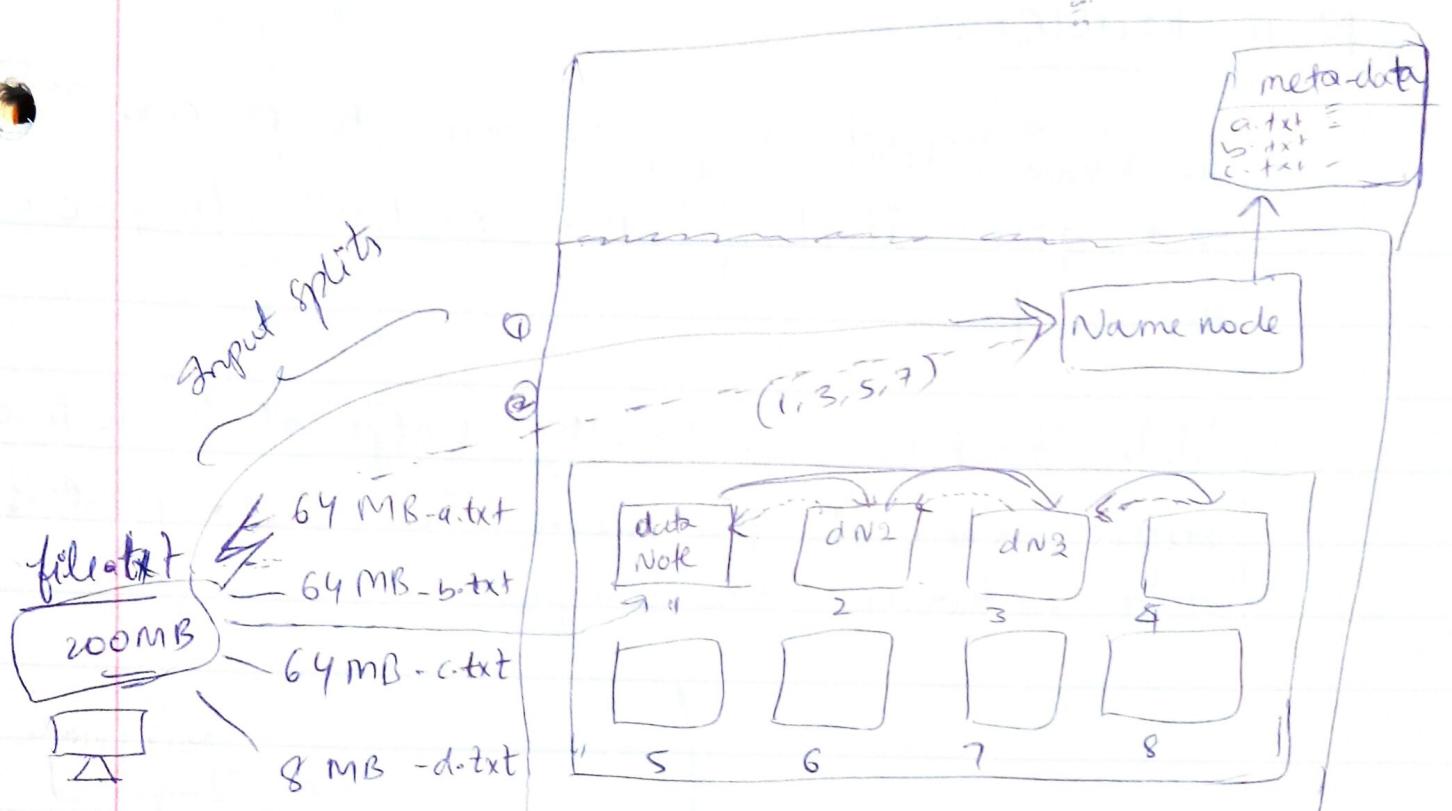
④ But in HDFS, default block size is 64 MB, & if 10 MB is used in a block, remaining block can be reclaimed. & blocks are less, it creates less meta-data.

⑤

⑥

## HDFS



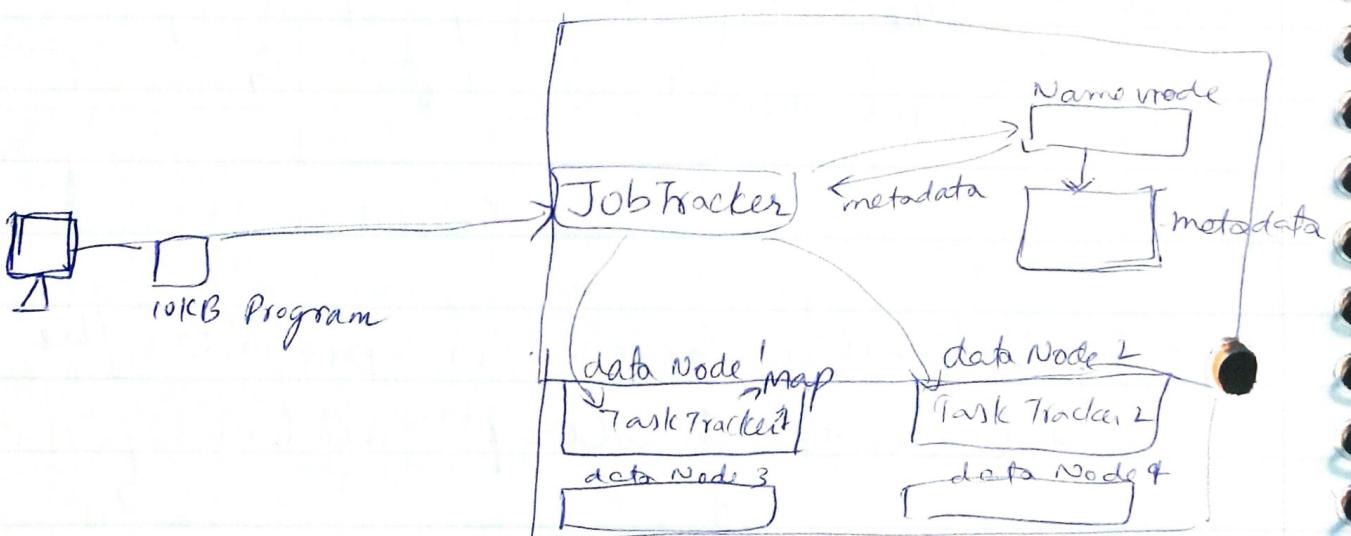


→ "Replication factor is provide the fault tolerance" hadoop system

→ To keep track of storage & running status  
for every '3' secs data node sends the  
block report & heart beat to Name node

## Map-Reduce:-

- we can write a program to process data and give it to "Job Tracker" to process the data.
- Job Tracker , with the help of Task Tracker runs (executes) program in data nodes and returns the output.



- Map :- Task Tracker takes the program received from Name node and <sup>Job Tracker</sup> executes on local machine. It is called as "Map"
- "Map" will be done on different blocks present at different data Nodes
- No of Mappers = "# of input splits"

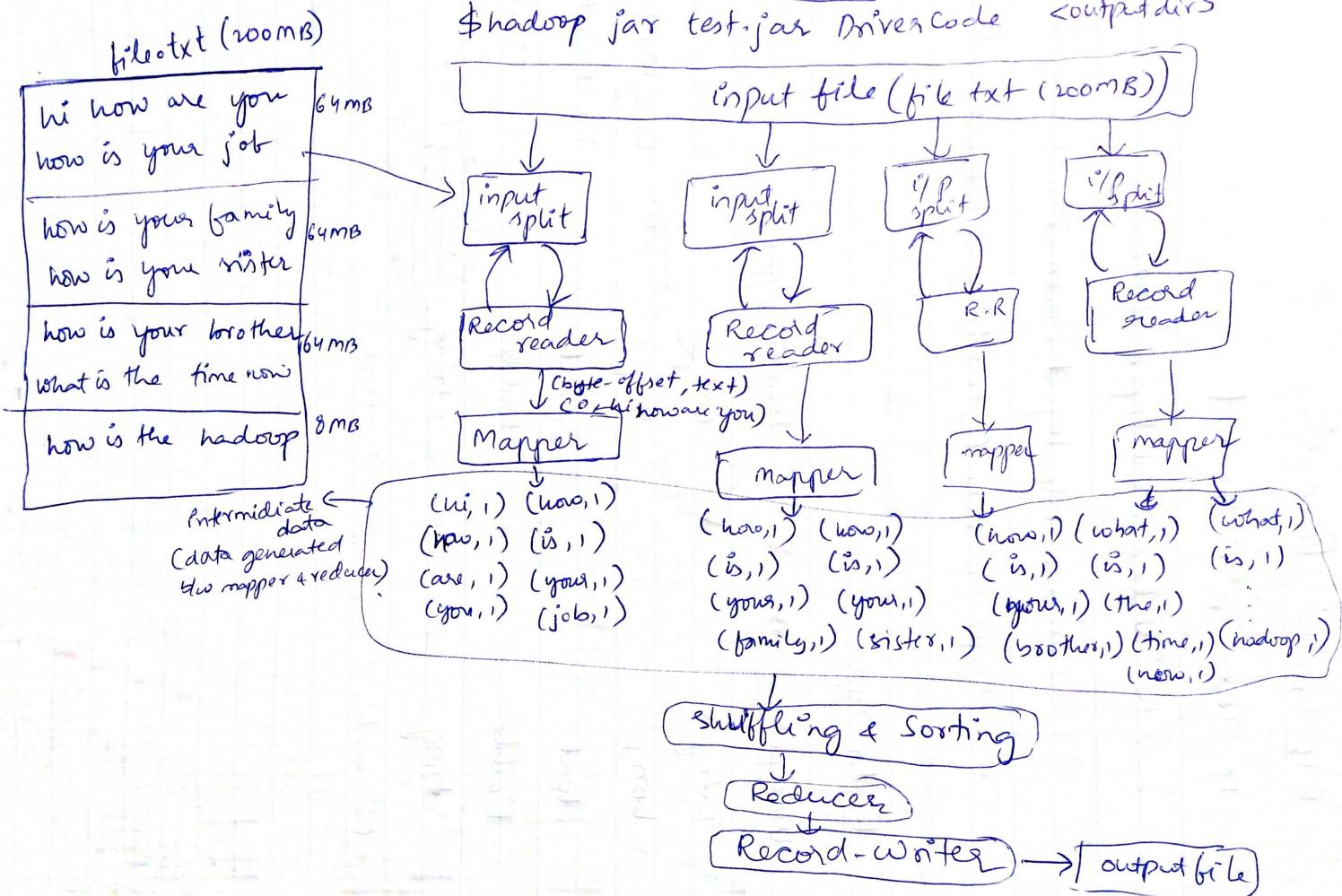
## REDUCE:-

- Combining output of all mappers and returned by reducers
- No of output files = No. of Reducers

## Map-Reduce Flow Chart

### Word-Count - Job

\$ hadoop jar test.jar DriverCode <output dir>



## File Input Formats:

1. Text Input Format
  2. Key Value Text Input Format
  3. Sequence File Input Format
  4. Sequence File As Text Input Format.
- 

## Data Types:

### Wrapper Class

Integer

Long

Float

Double

String

Character

### Primitive Types

int

long

float

double

String

char

new IntWritable().

IntWritable

LongWritable

FloatWritable

DoubleWritable

new Text()

~~new String()~~

Text

toString()

## Process Flow in Word-Count Job

1. Input file is divided into blocks of 64MB
2. "Record Reader", takes each input split and produces, `{Key, Value}` pairs based on "input file" file format.  
For example: TextInputFormat file generates, `< LongWritable, Text >` as output, where, "ByteOffset of each line" is `LongWritable` key & "content of each line" as `text`.
3. Record reader does it in a loop till all the lines in input split are completed.
4. Next, Mapper takes the output of Record-reader and generates the

Key Value pairs of below format

(byteoffset, text)

(0, hi how are you)

↓

(16, how is your job)

Mapper

↓

(hi-1) (how-1)

(how-1) (is-1)

(are-1) (your-1)

(you-1) (job-1)

5. Next in the shuffling phase, it combines all the keys to form unique keys. For example

(hi,-1) (now,~~1~~) (are,1) (is,1) (your,1) (you,1) (job,1)  
(now,1,1)

6. Next, in sorting phase, all the keys will be arranged in the sorting order

7. ~~Next~~  
7. Next in Reducer, phase all be reduced

to unique keys & their respective counts.

(hi,-1) (now,2) (are,1) (is,1) (your,1) (you,1) (job,1)

8. Then using the "RecordWriter", output will be written to file in the output directory

## File Formats:

1. In "Text InputFormat", files will be made into (Key, Value) pairs of <Byte-offset, Text>.

\* In Hadoop, everything is handled using the <KEY, VALUE> pairs format.

2. In "Key Value TextInputFormat", file will be split based on tab, For example,

101 Ram 20      <101, (Ram, 20)>  
102 Krish 30      <102, (Krish 30)>  
103 Rahim 40      <103, (Rahim 40)>

## output directory files

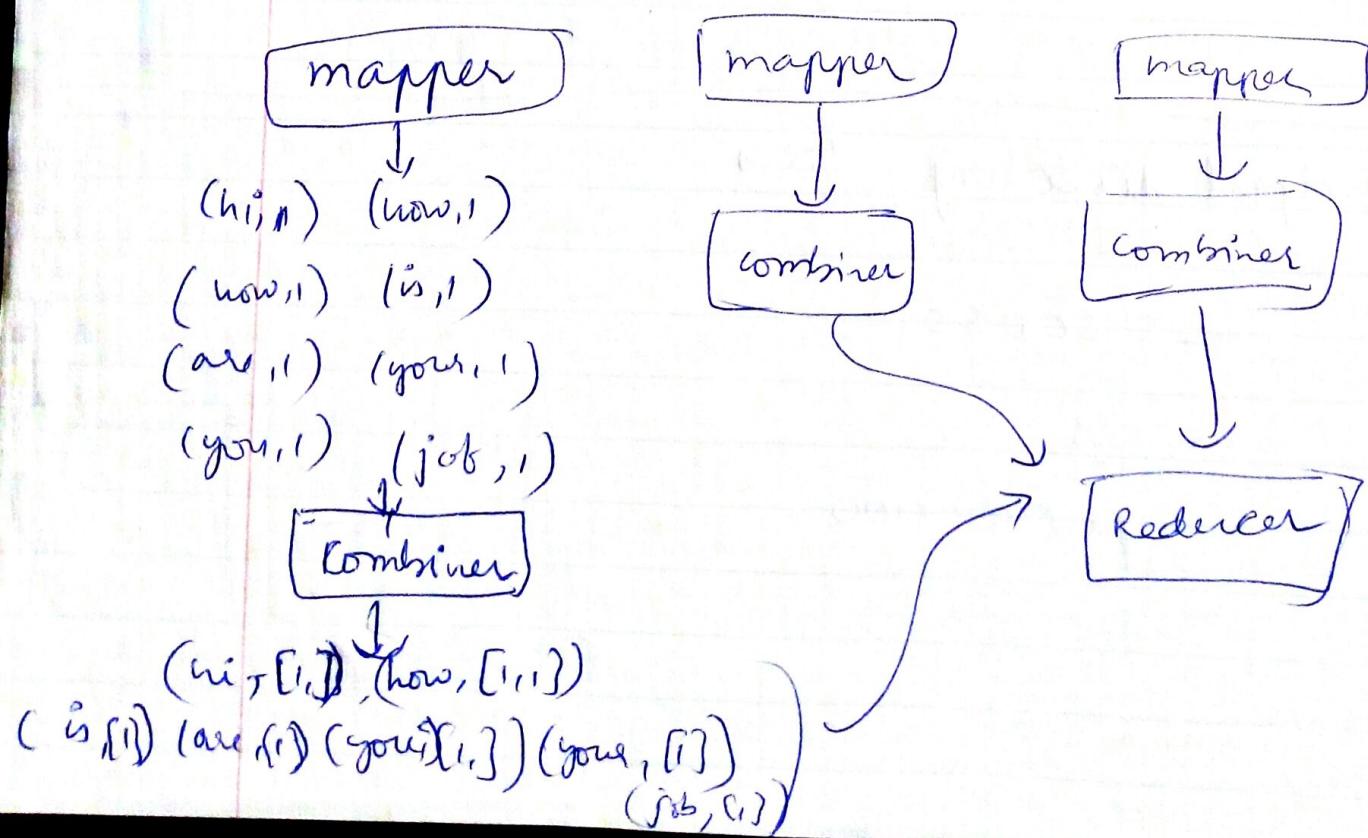
- SUCCESS

- logs

- part-00000

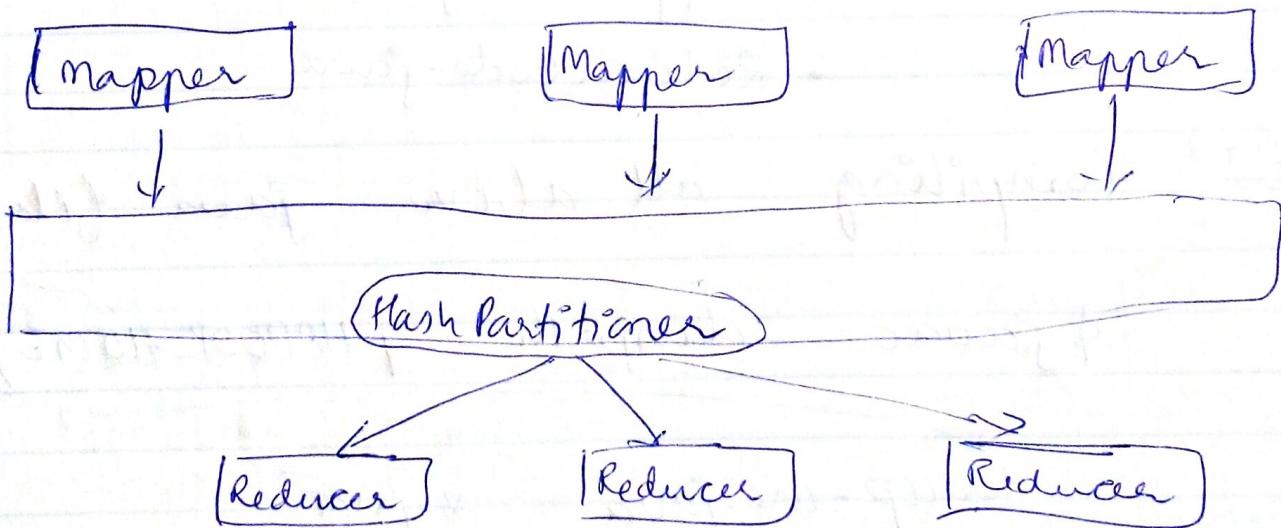
## Combiner :-

- 1. # of i/p splits = # of mappers.
- x. # of reducer = # of o/p files
- 2. By default only "one reducer"
- \* To reduce the traffic on the network due to congestion (because of all mappers sending data (key value) pairs to reducers. We use "combiner".
- \* Combiner is a mini-reducer on each mapper node.



## Partitioner

- 1) It is used to send specific,  $\langle \text{Key}, \text{Value} \rangle$  pairs to a specific Reducer.
- 2) It improves the performance & readability (for example, all Keys of length 2 to one partitioner, keys of length 3 to the other etc...)
- 3) By default, "Hash Partitioner" is used, where it randomly allocates the  $\langle \text{Key}, \text{Value} \rangle$  pairs to random reducers.  
(based on Hash code of object)



## Hello World Job (Sample Program)

Step 1) Create a file ,file.txt

vi , (or) cat (or) touch

Step 2) Loading file.txt from local file system  
to HDFS

\$ hadoop fs -put file.txt file

Step 3) writing programs

- DriverCode.java
- MapperCode.java
- ReducerCode.java

Step 4) Compiling all above java files

\$ javac -classpath \$HADOOP\_HOME /  
hadoop-core.jar \* .java

Step 5) Creating jar files

(Hadoop can only run the jar  
files)

```
$ jar cvf test.jar *-class
```

Step 6: Running above test.jar on file (which is there in hdfs).

~~\$ hadoop jar~~

```
$ hadoop jar test.jar DriverCode
```

file <test-output>

file <test-output>

## Hello World Job

### Word Count Job

Driver Code : WordCount.java

specify care of command line arguments

public class WordCount extends Configured implements  
Tool {

public int run (String args[]) throws Exception

{

if (args.length < 2)

{

System.out.println ("Plz give iip + oip directories  
properly")

return -1;

}

JobConf conf = new JobConf (WordCount.class);

conf.setJobName ("Word Count");

FileInputFormat.setInputPaths (conf, new

Path (args[0]));

FileOutputFormat.setOutputPath (conf, new

Path (args[1]));

```
conf.setMapperClass(WordMapper.class);
```

```
conf.setReducerClass(WordReducer.class);
```

```
conf.setMapOutputKeyClass(Text.class);  
Text & Text
```

```
conf.setMapOutputValueClass(IntWritable.class);
```

```
conf.setOutputKeyClass(Text.class);
```

```
conf.setOutputValueClass(IntWritable.class);
```

```
JobClient.runJob(conf);
```

```
return 0;
```

```
}
```

throw exception

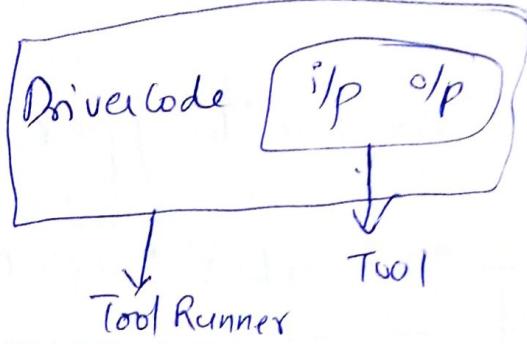
```
public static void main(String args[])
```

```
{ int exitCode = ToolRunner.run(new  
wordCount(), args);
```

```
System.exit(exitCode);
```

```
}
```

\$ hadoop jar wc.jar



→ for more information on Main class, look  
below 'ToolRunner' class implementation

Class ToolRunner

↳ public static int run(Tool tool, String args[])

```
{   f   tool.run(args);}
```

```
}
```

```
}
```

## Mapper Code :- WordMapper.java

```
public class WordMapper extends  
MapReduceBase implements Mapper<  
LongWritable, Text, Text, IntWritable>  
{  
    mapper i/p key type          ↓  
    mapper i/p value type       ↴  
    mapper o/p value type      ↗  
    {  
        public void map (LongWritable key,  
                           Text value, OutputCollector <Text, IntWritable>  
                           output, Reporter r) throws IOException &  
        {  
            String s = value.toString();  
            for (String word : s.split(" "))  
            {  
                if (word.length() > 0)  
                {  
                    output.collect(new Text(word),  
                                   new IntWritable(1));  
                }  
            }  
        }  
    }  
}
```

## Reducer Code:- Word Reducer - Java

```
public class WordReducer extends MapReduceBase  
implements Reducer<Text, IntWritable, Text,  
IntWritable>  
{  
    public void reduce (Text key, Iterator<  
IntWritable> values, OutputCollector<Text, IntWritable>  
output, Reporter r) throws IOException {  
    {  
        while (values.hasNext())  
        {  
            IntWritable i = values.next();  
            count += i.get(); // conversion from  
            // obj type to  
            // primitive  
            // type  
            output.collect(key, new IntWritable(count));  
        }  
    }  
}
```

## Partitioner code:

```
public class MyPartitioner implements  
Partitioner < Text, IntWritable >  
  
{  
    public void configure(JobConf conf)  
    {  
        3  
        public int getPartition(Text key,  
        IntWritable value, int setNumRedTasks)  
        {  
            string s = key.toString();  
            if (s.length() == 1)  
            {  
                return 0;  
            }  
            if (s.length() == 2)  
            {  
                return 1;  
            }  
            if (s.length() == 3)  
            {  
                return 2;  
            }  
            else {return 3;}  
        }  
    }  
}
```

Configure Partitioner class in Driver code, to run it.

conf.setPartitionerClass (MyPartitioner.class);

conf.setNumReduceTasks(4);

\* Maximum of reducers

= 99999

As  
otp file can be

part-00000  
↓

part-99999.

## Hadoop distributed file system (hdfs)

commands:

- 1) hadoop fs
- 2) hadoop fs -ls
  - list the metadata (directory + files)
- 3) if ↓ multinode cluster  
hadoop fs -ls & hdfs://<hostname>/user/  
<username>
- 4) create directory  
\$ hadoop fs -mkdir <dir-name>
  - hyphen-added, compared to unix systems
- 5) Directly we cannot create file in hdfs, we have to create it in local file system + copy to hdfs.
- 6) copy file to hdfs  
\$ hadoop fs -put <src> <dest>
  - ① local fs.
  - ② -copyFromLocal
  - ③ -moveFromLoc

② Get the file from hdfs file system to local file system

hdfs ↑  
↓ local fs  
(src) (dest)

\$ hadoop fs -get <src> <dest>

- copy To Local
- move To Local.

8) Writing data directly into hdfs is not possible \*

9) Viewing <sup>a file</sup> content in hdfs

\$ hadoop fs -cat <file-names>

10) remove file from hdfs

\$ hadoop fs -rm <file-name>

11) remove file forcefully

\$ hadoop fs -rmr <file-name>  
directory

12) Copy<sup>file</sup> from one location to another

\$ hadoop fs -cp <src> <dest>

13) for moving files

\$ hadoop fs -mv <src> <dest>

14) Disk Usage:

\$ hadoop fs -du <file-name>

15) set replication factor

\$ hadoop fs -setrep <int> <file-name>