

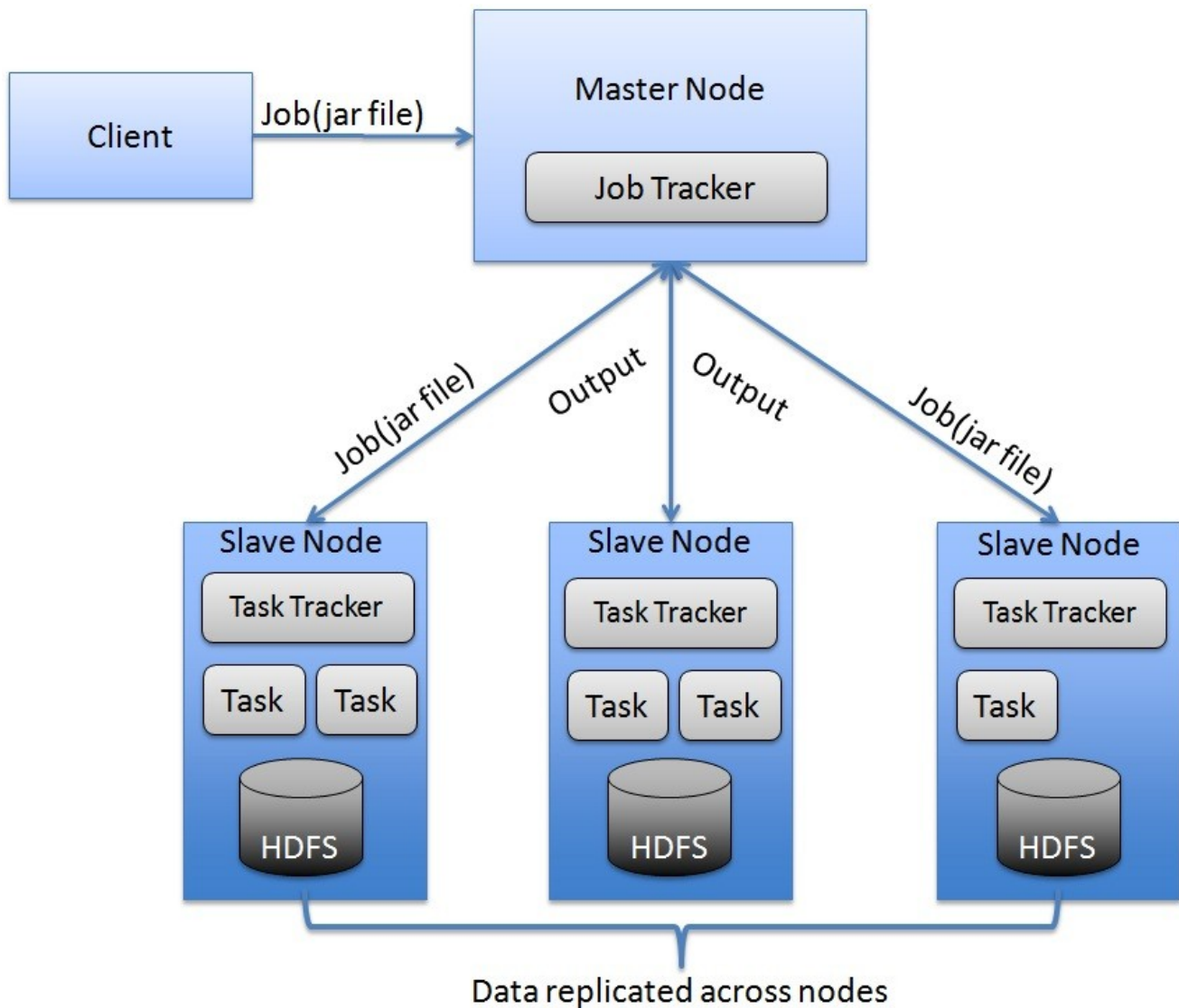
STUDY OF HADOOP MAPREDUCE FRAMEWORK
PRASAD PANDE
(psp150030@utdallas.edu)

Introduction

Apache Hadoop is an open source project that develops a reliable, scalable and distributed software solution for massively large scale data processing. Hadoop framework allows the computation of large scale structured, unstructured or semi-structured data in distributed environment on commodity hardware. Apache Hadoop consists of the **Hadoop Distributed File System (HDFS)** which is responsible for storing the data in the distributed cluster in fault-tolerant way. Computation is done using the mapreduce processing unit that designed on the principle of shipping the code where the data resides instead moving data over the network.

Hadoop Architecture

Hadoop follows master-slave architecture.



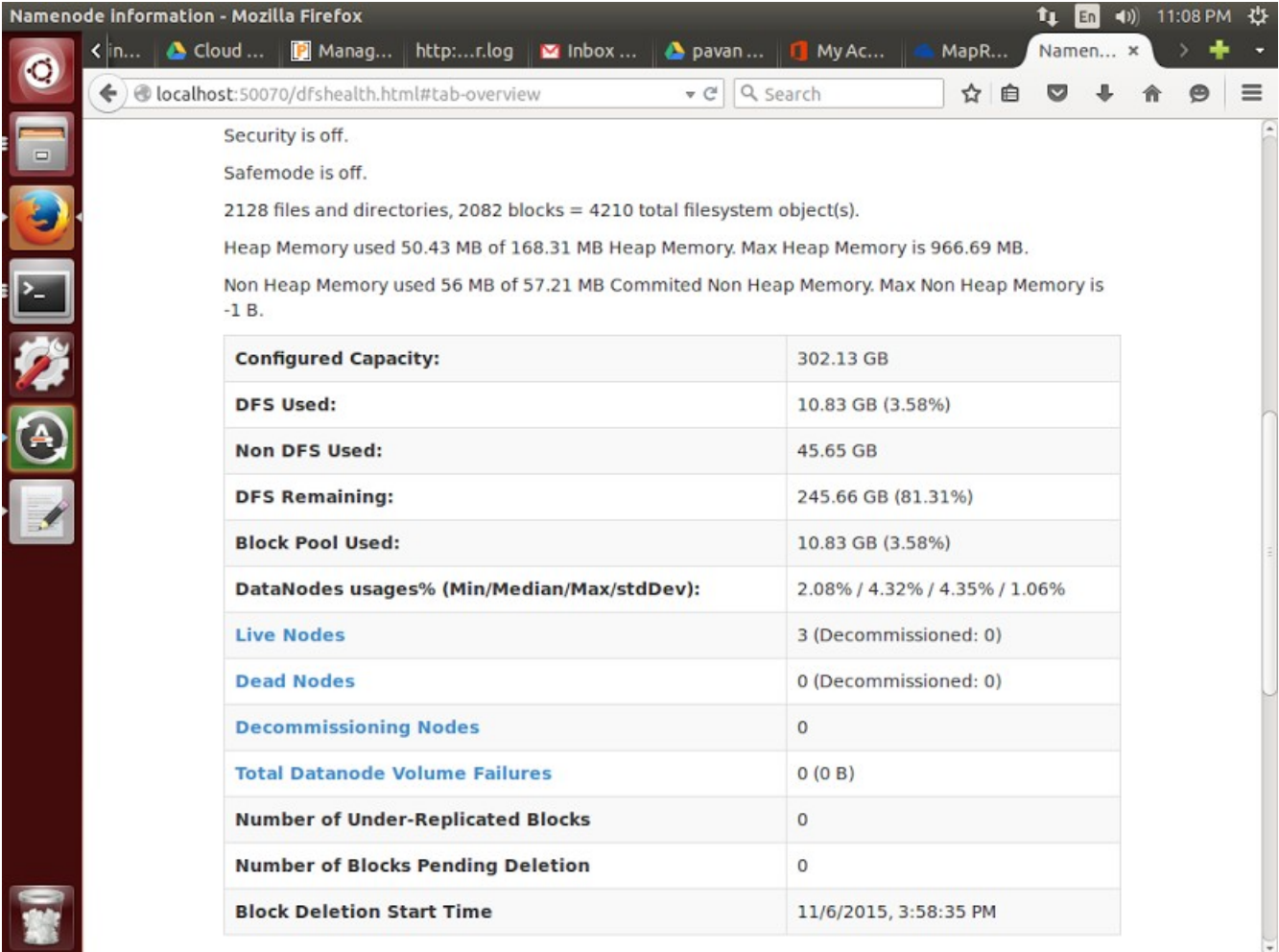
Objective

Objective of this assignment is to implement the mapreduce program on the crime data provided and study the different aspects of the HDFS. We considered the following cases:

1. How the file(s) are distributed over the nodes . One large input file or many small input files
2. How the mapper and reducer tasks are distributed over the nodes
3. The performance of map, reduce, and shuffling&sorting phases, including execution time, memory usage, etc.
4. How the system handle errors

System Configuration

We executed the mapreduce job on the cluster consisting of 3 data nodes and a name-node.



Namenode Information - Mozilla Firefox

localhost:50070/dfshealth.html#tab-overview

Security is off.
Safemode is off.
2128 files and directories, 2082 blocks = 4210 total filesystem object(s).
Heap Memory used 50.43 MB of 168.31 MB Heap Memory. Max Heap Memory is 966.69 MB.
Non Heap Memory used 56 MB of 57.21 MB Committed Non Heap Memory. Max Non Heap Memory is -1 B.

Configured Capacity:	302.13 GB
DFS Used:	10.83 GB (3.58%)
Non DFS Used:	45.65 GB
DFS Remaining:	245.66 GB (81.31%)
Block Pool Used:	10.83 GB (3.58%)
DataNodes usages% (Min/Median/Max/stdDev):	2.08% / 4.32% / 4.35% / 1.06%
Live Nodes	3 (Decommissioned: 0)
Dead Nodes	0 (Decommissioned: 0)
Decommissioning Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion	0
Block Deletion Start Time	11/6/2015, 3:58:35 PM

Implementation

I observed that, in my cluster default block size was 128 MB.

The default replication factor is 1 and we changed it to 2 in our cluster for the providing the fault-tolerance.

1) Putting the input data on the cluster

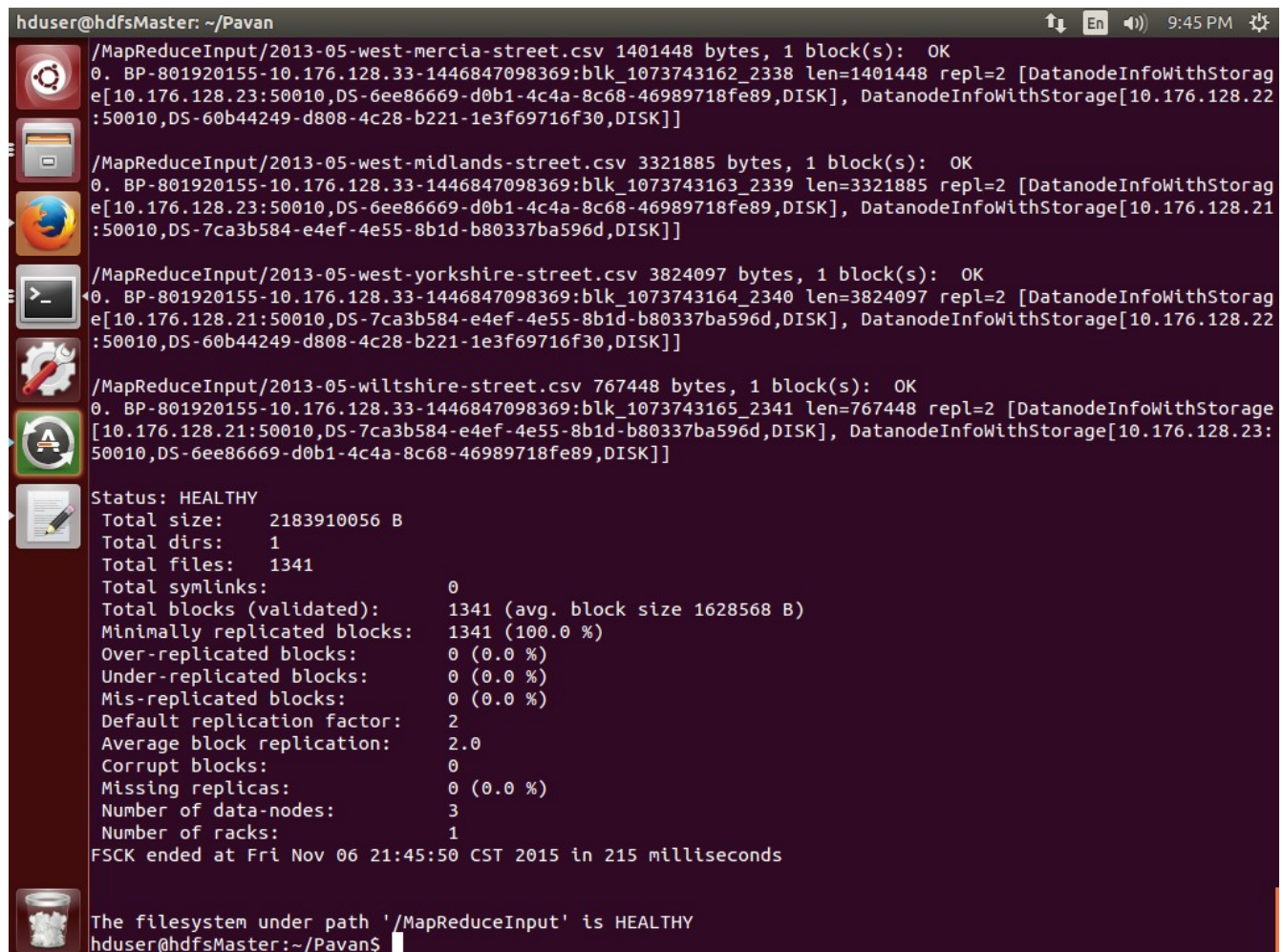
We consider 2 scenarios as follows;

1. Large number of small files on the cluster

```
hdfs dfs -mkdir /cloud_input
hdfs dfs -put /data/* /cloud_input
```

The block level information such as total number of blocks, location of each block is accessed using the following command.

```
hdfs fsck /cloud_complete_input -files -locations -blocks
```

A terminal window titled 'hduser@hdfsMaster: ~/Pavan' showing the output of the 'hdfs fsck' command. The output lists four MapReduce input files, their sizes, and their block locations. It then provides a detailed status report for the filesystem, including total size, number of files and blocks, replication statistics, and the number of data nodes and racks. The terminal also shows the time taken for the fsck operation and a final health check message.

```
hduser@hdfsMaster: ~/Pavan
/MapReduceInput/2013-05-west-mercia-street.csv 1401448 bytes, 1 block(s): OK
0. BP-801920155-10.176.128.33-1446847098369:blk_1073743162_2338 len=1401448 repl=2 [DatanodeInfoWithStorage[10.176.128.23:50010,DS-6ee86669-d0b1-4c4a-8c68-46989718fe89,DISK], DatanodeInfoWithStorage[10.176.128.22:50010,DS-60b44249-d808-4c28-b221-1e3f69716f30,DISK]]

/MapReduceInput/2013-05-west-midlands-street.csv 3321885 bytes, 1 block(s): OK
0. BP-801920155-10.176.128.33-1446847098369:blk_1073743163_2339 len=3321885 repl=2 [DatanodeInfoWithStorage[10.176.128.23:50010,DS-6ee86669-d0b1-4c4a-8c68-46989718fe89,DISK], DatanodeInfoWithStorage[10.176.128.21:50010,DS-7ca3b584-e4ef-4e55-8b1d-b80337ba596d,DISK]]

/MapReduceInput/2013-05-west-yorkshire-street.csv 3824097 bytes, 1 block(s): OK
0. BP-801920155-10.176.128.33-1446847098369:blk_1073743164_2340 len=3824097 repl=2 [DatanodeInfoWithStorage[10.176.128.21:50010,DS-7ca3b584-e4ef-4e55-8b1d-b80337ba596d,DISK], DatanodeInfoWithStorage[10.176.128.22:50010,DS-60b44249-d808-4c28-b221-1e3f69716f30,DISK]]

/MapReduceInput/2013-05-wiltshire-street.csv 767448 bytes, 1 block(s): OK
0. BP-801920155-10.176.128.33-1446847098369:blk_1073743165_2341 len=767448 repl=2 [DatanodeInfoWithStorage[10.176.128.21:50010,DS-7ca3b584-e4ef-4e55-8b1d-b80337ba596d,DISK], DatanodeInfoWithStorage[10.176.128.23:50010,DS-6ee86669-d0b1-4c4a-8c68-46989718fe89,DISK]]

Status: HEALTHY
Total size: 2183910056 B
Total dirs: 1
Total files: 1341
Total symlinks: 0
Total blocks (validated): 1341 (avg. block size 1628568 B)
Minimally replicated blocks: 1341 (100.0 %)
Over-replicated blocks: 0 (0.0 %)
Under-replicated blocks: 0 (0.0 %)
Mis-replicated blocks: 0 (0.0 %)
Default replication factor: 2
Average block replication: 2.0
Corrupt blocks: 0
Missing replicas: 0 (0.0 %)
Number of data-nodes: 3
Number of racks: 1
FSCK ended at Fri Nov 06 21:45:50 CST 2015 in 215 milliseconds

The filesystem under path '/MapReduceInput' is HEALTHY
hduser@hdfsMaster:~/Pavan$
```

Above screen shot shows the total number of blocks as well the distribution and the location of the blocks. The distribution of the blocks (number of blocks) on the 3 data nodes is shown below.

```
hduser@hdfsMaster: ~/Pavan
hduser@hdfsMaster:~/Pavan$ hadoop fsck /MapReduceInput -files -locations -blocks | grep -c "10.176.128.21"
DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.
Connecting to namenode via http://hdfsMaster:50070/fsck?ugi=hduser&files=1&locations=1&blocks=1&path=%2FMapReduceInput
909
hduser@hdfsMaster:~/Pavan$ hadoop fsck /MapReduceInput -files -locations -blocks | grep -c "10.176.128.22"
DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.
Connecting to namenode via http://hdfsMaster:50070/fsck?ugi=hduser&files=1&locations=1&blocks=1&path=%2FMapReduceInput
890
hduser@hdfsMaster:~/Pavan$ hadoop fsck /MapReduceInput -files -locations -blocks | grep -c "10.176.128.23"
DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.
Connecting to namenode via http://hdfsMaster:50070/fsck?ugi=hduser&files=1&locations=1&blocks=1&path=%2FMapReduceInput
883
hduser@hdfsMaster:~/Pavan$
```

We can see that the distribution of the blocks over the cluster is uniformly done.

2. Putting single large size consolidated file


```
hduser@hdfsMaster: ~/Pavan
.22:50010,DS-60b44249-d808-4c28-b221-1e3f69716f30,DISK]]
11. BP-801920155-10.176.128.33-1446847098369:blk_1073744500_3689 len=134217728 repl=2 [DatanodeInfoWithStorage[10.176.128.23:50010,DS-60b44249-d808-4c28-b221-1e3f69716f30,DISK], DatanodeInfoWithStorage[10.176.128.23:50010,DS-60b44249-d808-4c28-b221-1e3f69716f30,DISK]]
12. BP-801920155-10.176.128.33-1446847098369:blk_1073744501_3690 len=134217728 repl=2 [DatanodeInfoWithStorage[10.176.128.23:50010,DS-60b44249-d808-4c28-b221-1e3f69716f30,DISK], DatanodeInfoWithStorage[10.176.128.23:50010,DS-60b44249-d808-4c28-b221-1e3f69716f30,DISK]]
13. BP-801920155-10.176.128.33-1446847098369:blk_1073744502_3691 len=134217728 repl=2 [DatanodeInfoWithStorage[10.176.128.23:50010,DS-60b44249-d808-4c28-b221-1e3f69716f30,DISK], DatanodeInfoWithStorage[10.176.128.23:50010,DS-60b44249-d808-4c28-b221-1e3f69716f30,DISK]]
14. BP-801920155-10.176.128.33-1446847098369:blk_1073744503_3692 len=134217728 repl=2 [DatanodeInfoWithStorage[10.176.128.23:50010,DS-60b44249-d808-4c28-b221-1e3f69716f30,DISK], DatanodeInfoWithStorage[10.176.128.23:50010,DS-60b44249-d808-4c28-b221-1e3f69716f30,DISK]]
15. BP-801920155-10.176.128.33-1446847098369:blk_1073744504_3693 len=134217728 repl=2 [DatanodeInfoWithStorage[10.176.128.23:50010,DS-60b44249-d808-4c28-b221-1e3f69716f30,DISK], DatanodeInfoWithStorage[10.176.128.23:50010,DS-60b44249-d808-4c28-b221-1e3f69716f30,DISK]]
16. BP-801920155-10.176.128.33-1446847098369:blk_1073744505_3694 len=36426409 repl=2 [DatanodeInfoWithStorage[10.176.128.23:50010,DS-60b44249-d808-4c28-b221-1e3f69716f30,DISK], DatanodeInfoWithStorage[10.176.128.23:50010,DS-60b44249-d808-4c28-b221-1e3f69716f30,DISK]]

Status: HEALTHY
Total size: 2183910057 B
Total dirs: 0
Total files: 1
Total symlinks: 0
Total blocks (validated): 17 (avg. block size 128465297 B)
Minimally replicated blocks: 17 (100.0 %)
Over-replicated blocks: 0 (0.0 %)
Under-replicated blocks: 0 (0.0 %)
Mis-replicated blocks: 0 (0.0 %)
Default replication factor: 2
Average block replication: 2.0
Corrupt blocks: 0
Missing replicas: 0 (0.0 %)
Number of data-nodes: 3
Number of racks: 1
FSCK ended at Fri Nov 06 22:43:22 CST 2015 in 0 milliseconds

The filesystem under path '/largeFile.csv' is HEALTHY
hduser@hdfsMaster:~/Pavan$
```

As we can see the number of blocks for the single large files are very very low(i.e. 17) as compared to the large number of small files. The length of each block is larger than the case 1)

Distribution of the blocks over the 3 data nodes is shown below:

```
hduser@hdfsMaster: ~/Pavan
hduser@hdfsMaster:~/Pavan$ hadoop fsck /largeFile.csv -files -locations -blocks | grep -c "10.176.128.21"
DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.
Connecting to namenode via http://hdfsMaster:50070/fsck?ugi=hduser&files=1&locations=1&blocks=1&path=%2FlargeFile.csv
1
hduser@hdfsMaster:~/Pavan$ hadoop fsck /largeFile.csv -files -locations -blocks | grep -c "10.176.128.22"
DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.
Connecting to namenode via http://hdfsMaster:50070/fsck?ugi=hduser&files=1&locations=1&blocks=1&path=%2FlargeFile.csv
16
hduser@hdfsMaster:~/Pavan$ hadoop fsck /largeFile.csv -files -locations -blocks | grep -c "10.176.128.23"
DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.
Connecting to namenode via http://hdfsMaster:50070/fsck?ugi=hduser&files=1&locations=1&blocks=1&path=%2FlargeFile.csv
17
hduser@hdfsMaster:~/Pavan$
```

2) Map Reduce Program

Mapper:

```
public class CrimeMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
```

```
    public int getFirstDigit(int number){
```

```
        int digit=0;
```

```
        while(number>0) {
```

```
            digit = number % 10;
```

```
            number = number /10;
```

```
        }
```

```
        return digit;
```

```
    }
```

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

```
        String line = value.toString();
```

```
        if(!line.split(",")[0].equals("Crime ID") &&  
            line.split(",").length>=8){
```

```
            int easting = (line.split(",")[4].length()!=0)?
```

```
                Integer.parseInt(line.split(",")[4]):0;
```

```
            int northing = (line.split(",")[5].length()!=0)?
```

```

Integer.parseInt(line.split(",")[5]):0;

String crime_type = line.split(",")[7];
String new_key = getFirstDigit(easting)+" "+getFirstDigit(northing)+"
               "+crime_type;
context.write(new Text(new_key), new IntWritable(1));
}
}

```

In the mapper part, I take one line at once in the input, took the easting, northing and crime_category by spiting the input line. Output of the reducer is the composite key consisting of the easting, northing and crime_category as the input and 1 as value. **GetFirstDigit()** function helps to extract the first digit of of the given easting/northing value.

Reducer

```

public class CrimeReducer extends Reducer<Text, IntWritable, Text, IntWritable>{

    public void reduce(Text key,Iterable<IntWritable> values, Context context)
throws IOException, InterruptedException {
        int sum=0;
        for(IntWritable val: values) {
            sum+=val.get();
        }
        context.write(key, new IntWritable(sum));
    }

}

```

Reducer simply takes the output of the mapper and since we are using the default partitioner, all the key-values with same key (i.e same easting, northing and crime_category value) goes to same reducer. Default partitioner takes the hash value of the given key and divide it by number of reducer. Reducer simply sum up all the values to output the key and the cout as the output.

```

public class CrimeDriver {

    public static void main(String[] args) throws IOException,
ClassNotFoundException, InterruptedException {
        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf,
args).getRemainingArgs();
        if(otherArgs.length!=2) {
            System.err.println("Usage: crimeDriver <in> <out>");
            System.exit(2);
        }
        System.out.println("In Driver Program");
        System.out.println("Input: "+otherArgs[0]);
        System.out.println("Output: "+otherArgs[1]);
        Job job = new Job(conf,"CrimeDataAnalysis");
        job.setJarByClass(CrimeDriver.class);
        job.setMapperClass(CrimeMapper.class);
        job.setReducerClass(CrimeReducer.class);
        //job.setCombinerClass(CrimeReducer.class);
        job.setOutputKeyClass(Text.class);
    }
}

```



```

        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
        long start = new Date().getTime();
        job.waitForCompletion(true);
        long end = new Date().getTime();
        System.out.println("Job took " + (end-start) + " milliseconds");
        System.exit(0);
    }
}

```

Driver program initiates the configurations and set the mapper and the reducer classes. Also it sets the output key and value format. Addition to that, to record the execution time for the given job, I included the code for the time before and after the execution.

Result Scenarios

Run1: Large number of small Files: (1st digit for region description + Crime Category)

Number of Reducer Tasks	1
Number of Mapper Tasks	1341
Bytes Shuffled over the network (bytes)	424424437
Execution Time (milliseconds)	487198
Heap Usage (bytes)	671816548352
Output Bytes written to disk (bytes)	17519

Run2: Large number of small Files: (1st digit for region description + Crime Category)

Number of Reducer Tasks	5
Number of Mapper Tasks	6705
Bytes Shuffled over the network (bytes)	424456721
Execution Time (milliseconds)	506266
Heap Usage (bytes)	673901641728
Output Bytes written to disk (bytes)	17519

Run3: Single Large File: (1st digit for region description + Crime Category)

Number of Reducer Tasks	1
Number of Mapper Tasks	17
Bytes Shuffled over the network (bytes)	424416504
Execution Time (milliseconds)	229672

Heap Usage (bytes)	3499483136
Output Bytes written to disk (bytes)	17526

Run4: Single Large File: (1st digit for region description + Crime Category)

Number of Reducer Tasks	5
Number of Mapper Tasks	17
Bytes Shuffled over the network (bytes)	424416912
Execution Time (milliseconds)	237399
Heap Usage (bytes)	3882745856
Output Bytes written to disk (bytes)	17526

Run5: Large number of small Files: (1st 3 digits in region description + Crime category)

Number of Reducer Tasks	1
Number of Mapper Tasks	1341
Bytes Shuffled over the network (bytes)	486684977
Execution Time (milliseconds)	472273
Heap Usage (bytes)	672009777152
Output Bytes written to disk (bytes)	12558050

Run6: Large number of small Files : (1st 3 digits in region description + Crime category)

Number of Reducer Tasks	5
Number of Mapper Tasks	1341
Bytes Shuffled over the network (bytes)	486717161
Execution Time (milliseconds)	534809
Heap Usage (bytes)	673840549888
Output Bytes written to disk (bytes)	12558050

Run7: Single Large File:(1st 3 digits in region description + Crime category)

Number of Reducer Tasks	1
Number of Mapper Tasks	17
Bytes Shuffled over the network (bytes)	486677048

Execution Time (milliseconds)	249596
Heap Usage (bytes)	3519283200
Output Bytes written to disk (bytes)	12558061

Run8: Single Large File:(1st 3 digits in region description + Crime category)

Number of Reducer Tasks	5
Number of Mapper Tasks	17
Bytes Shuffled over the network (bytes)	486677456
Execution Time (milliseconds)	248817
Heap Usage (bytes)	3882745856
Output Bytes written to disk (bytes)	12558061

Error Handling

MapReduce is designed to be a fault-tolerant system by replicating the data on the HDFS and re-executing the tasks that are failed during the execution. These tasks are handled by the system itself and client don't need to monitor this.

Failure of name-node:

To handle the name-node failure, there are two ways we can recover with the minimum or no down time. We can design a secondary name node which will copy the logs and meta-data of the name node and the logs. Logs contain in the *edit.logs* and the *fsimages* files. The job of the secondary name node is to merge these log files and in case of failure these files and metadata will be copied to the new name node and will be run with minimum downtime.

Sometime in order to reduce the down time, a stand-by node is kept for the current node which continuously copies the logs from the name node to the stand-by node.

Failure of data node:

Data node failure can occur in two ways. During execution if one of the data node fails, name-node assigns the execution of the mapreduce task to the node where the replica of the current data block exists. HDFS configuration file *hdfs-site.xml* contains the replication factor that is configured for the given cluster for which default value is 1.

The placement of the replicas in the cluster is determined by the algorithm which ensures that the data blocks will be available even if the rack fails to provide maximum availability.

Task distribution

Hadoop MapReduce works on the principle of moving code over the network instead of data. When client request for the particular job name node identifies the nodes where the respective data resides and client directly execute the code on the nodes with the help of job tracker and task tracker. The communication between the client and the data node is not through the namenode. Namenode only helps in identifying the data. Once the data location is clear, client directly communicates with the respective data nodes. So the distribution of the tasks depends on the data locality. If data is evenly

distributed then execution engine (jobtracker) assigns the execution work to those nodes uniformly.

Conclusion

From the above scenarios we discussed following are the conclusions we can draw some of the conclusion as follows:

1. Execution time reduces considerably when we execute the single large file instead of the large number of small files.
2. When we use the single large file, the consumption of the disk happens effectively as the one file is divided into the 128 MB blocks. In case of large number of small files, the average block size comes as only ~1MB since the file size is very much less than 128MB resulting in the less efficient storage.
3. The heap uses decreases on large scale (approximately 50%) when we go from large number of small file to the single large file.
4. If we write the combiner code it will perform the local aggregation on the mapper output which will reduce the data that will be shuffled over the network resulting in the less execution time and less consumption of the network resources.

In this project we understand the architecture and the working of the hadoop mapreduce framework. Hadoop is designed for the processing of the job in the distributed fashion which allows the parallel execution of the tasks as well as the fault-tolerant way of storing the large amount of data on low price commodity hardware.