Wordcount using MapReduce

Tushar B. Kute,
http://tusharkute.com

What is MapReduce?

- MapReduce is a framework using which we can write applications to process huge amounts of data, in parallel, on large clusters of commodity hardware in a reliable manner.
- MapReduce is a processing technique and a program model for distributed computing based on java.
- The MapReduce algorithm contains two important tasks, namely Map and Reduce.

Map and Reduce

- Map takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs).
- Secondly, reduce task, which takes the output from a map as an input and combines those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce task is always performed after the map job.

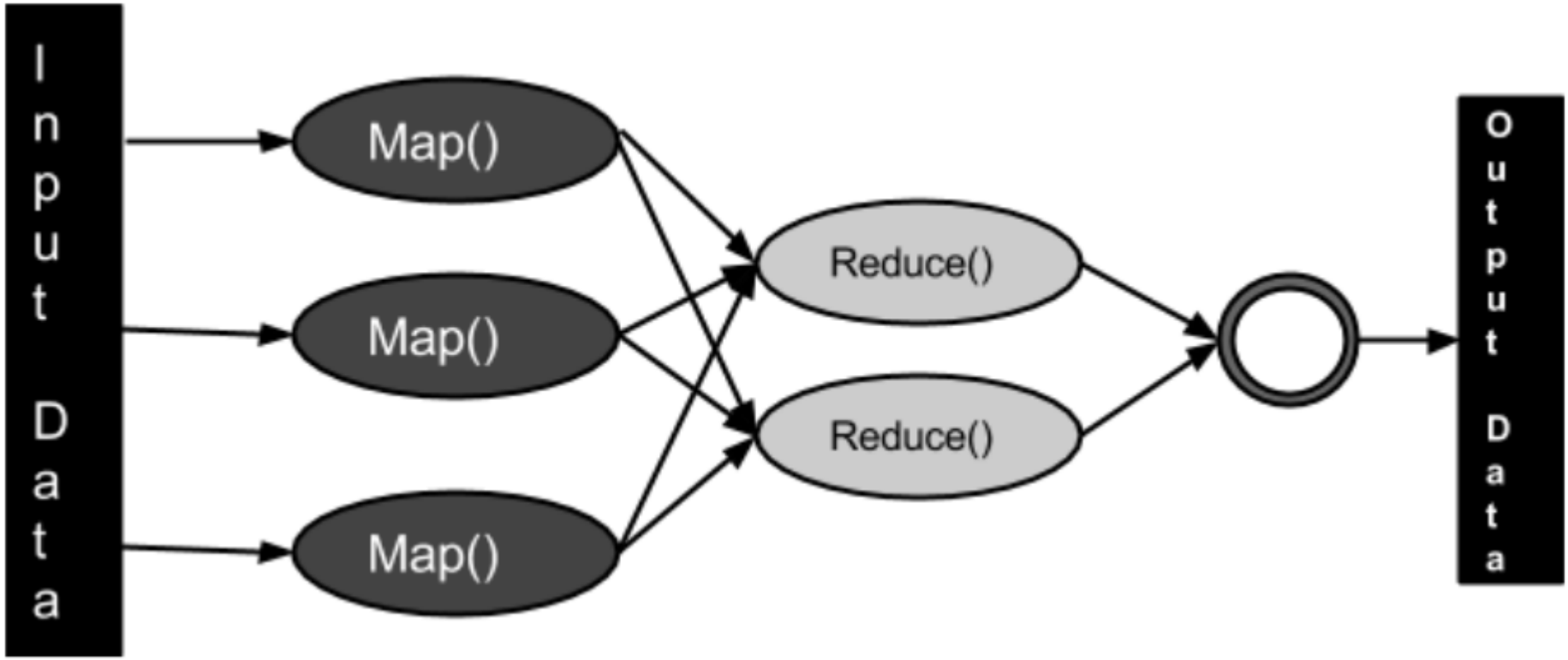
Map and Reduce

- The major advantage of MapReduce is that it is easy to scale data processing over multiple computing nodes.
- Under the MapReduce model, the data processing primitives are called mappers and reducers.
- Decomposing a data processing application into mappers and reducers is sometimes nontrivial. But, once we write an application in the MapReduce form, scaling the application to run over hundreds, thousands, or even tens of thousands of machines in a cluster is merely a configuration change.
- This simple scalability is what has attracted many programmers to use the MapReduce model.

The Algorithm

- MapReduce program executes in three stages, namely map stage, shuffle stage, and reduce stage.
- **Map stage**: The map or mapper's job is to process the input data. Generally the input data is in the form of file or directory and is stored in the Hadoop file system (HDFS). The input file is passed to the mapper function line by line. The mapper processes the data and creates several small chunks of data.
- **Reduce stage**: This stage is the combination of the Shuffle stage and the Reduce stage. The Reducer's job is to process the data that comes from the mapper. After processing, it produces a new set of output, which will be stored in the HDFS.

The MapReduce



Inserting Data into HDFS

- The MapReduce framework operates on <key, value> pairs, that is, the framework views the input to the job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job, conceivably of different types.
- The key and the value classes should be in serialized manner by the framework and hence, need to implement the Writable interface. Additionally, the key classes have to implement the Writable-Comparable interface to facilitate sorting by the framework.
- Input and Output types of a MapReduce job: (Input) <k1,v1> -> map -> <k2, v2>-> reduce -> <k3, v3> (Output).

Data input and output

	Input	Output
Map	<k1, v1>	list (<k2, v2>)
Reduce	<k2, list(v2)>	list (<k3, v3>)

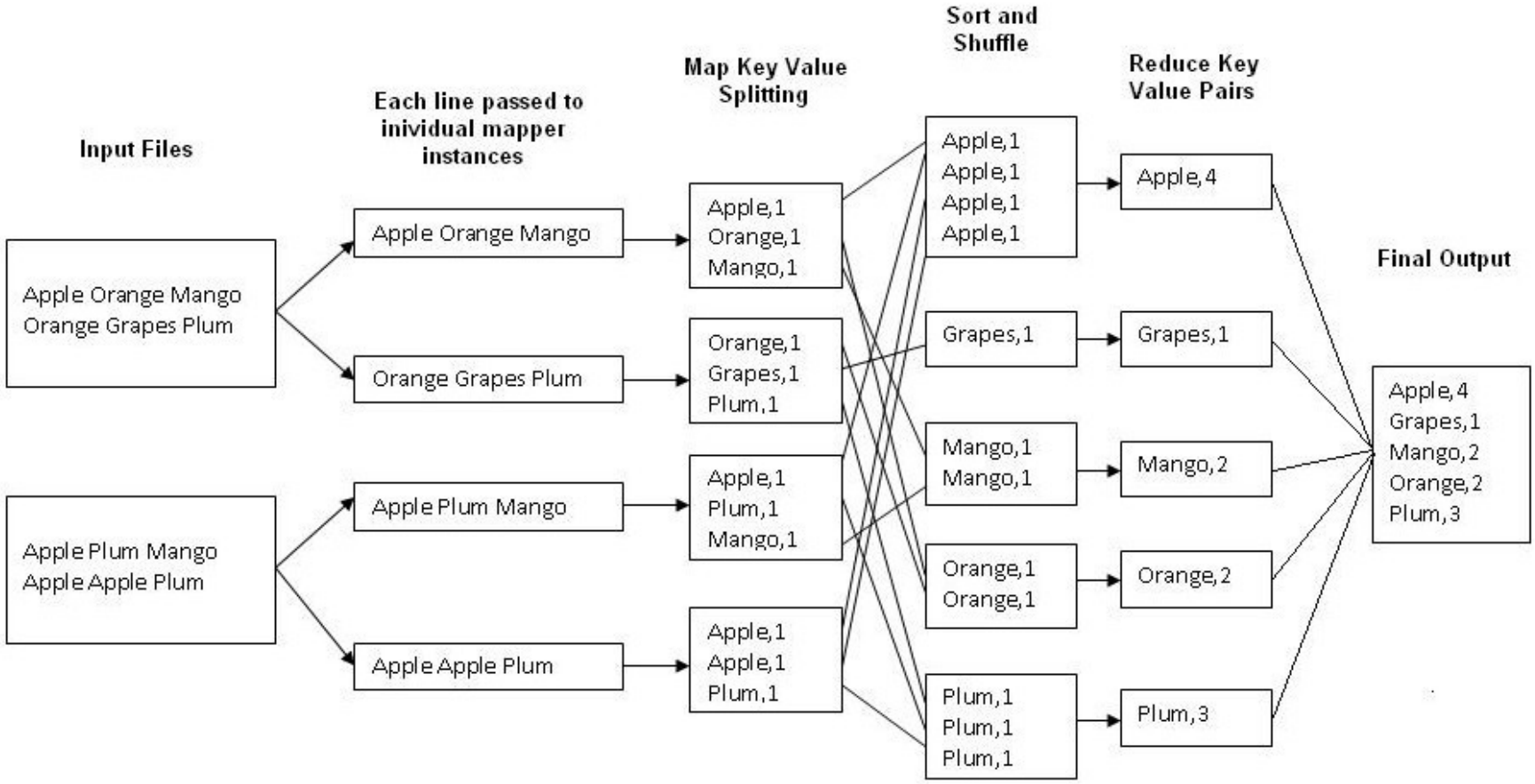
Terminologies

- Mapper - Mapper maps the input key/value pairs to a set of intermediate key/value pair.
- NamedNode - Node that manages the Hadoop Distributed File System (HDFS).
- DataNode - Node where data is presented in advance before any processing takes place.
- MasterNode - Node where JobTracker runs and which accepts job requests from clients.
- SlaveNode - Node where Map and Reduce program runs.
- JobTracker - Schedules jobs and tracks the assign jobs to Task tracker.
- Task Tracker - Tracks the task and reports status to JobTracker.
- Job - A program is an execution of a Mapper and Reducer across a dataset.
- Task - An execution of a Mapper or a Reducer on a slice of data.

Example:

- Write a program using Hadoop that counts number of occurrences of words in a file.

The wordcount flow



The dataset: fruits.txt

```
1 apple orange mango
2 apple plum grapes
3 pineapple raspberry banana
4 banana orange apple
5 plum raspberry pomgranate
6 grapes grapes apple mango
```

Example:

- Wordcount.java

mapper

```
public class WordCount {

 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context)
                                    throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
 }
```

reducer

```
public static class Reduce 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));
    }
}
```

main

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

    Job job = new Job(conf, "wordcount");

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

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

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

    job.waitForCompletion(true);
  }
}
```

Compilation and Execution

- Let us assume we are in the home directory of a Hadoop user (e.g. /home/rashmi).
- Follow the steps given below to compile and execute the above program.
- **Step 1**
  - The following command is to create a directory to store the compiled java classes.
  - `$ mkdir words`

Compilation and Execution

- **Step 2**

  Download hadoop-core-1.2.1.jar, which is used to compile and execute the MapReduce program. Visit the following link

  http://mvnrepository.com/artifact/org.apache.hadoop/hadoop-core/1.2.1
  To download the jar. Let us assume the downloaded folder is /home/rashmi/words.

- **Step 3**

  The following commands are used for compiling the wordcount.java program and creating a jar for the program.

  ```
  $ javac -classpath hadoop-core-1.2.1.jar words/WordCount.java
  $ jar -cvf  words.jar  -C  words/  .
  ```

Compilation and Execution

- **Step 4**
  - The following command is used to create an input directory in HDFS.
  - $hadoop fs -mkdir /input
- **Step 5**
  - The following command is used to copy input dataset file on HDFS.
  - $hadoop fs -put fruits.txt /input
- **Step 6**
  - The following command is used to verify the files in the input directory.
  - $hadoop fs -ls /input

Compilation and Execution

- **Step 7**
  - The following command is used to run the Wordcount application by taking the input files from the input directory.
  - $hadoop jar words.jar WordCount /input /output
  - Wait for a while until the file is executed. After execution, the output will contain the number of input splits, the number of Map tasks, the number of reducer tasks, etc. The output directory must *not* be existing already.

Compilation and Execution

- **Step 8**
  - The following command is used to verify the resultant files in the output folder.
  - $hadoop fs -ls /output
- **Step 9**
  - The following command is used to see the output in part-r-00000 file. This file is generated by HDFS.
  - $hadoop fs -cat /output/part-r-00000

Compilation and Execution

- **Step 10**

  The following command is used to copy the output file from HDFS to the local file system for analyzing.

  - $hadoop fs -get /output/part-r-00000

Output:

```
apple    4
banana   2
grapes   3
mango    2
orange   2
pineapple        1
plum     2
pomgranate       1
raspberry        2
```

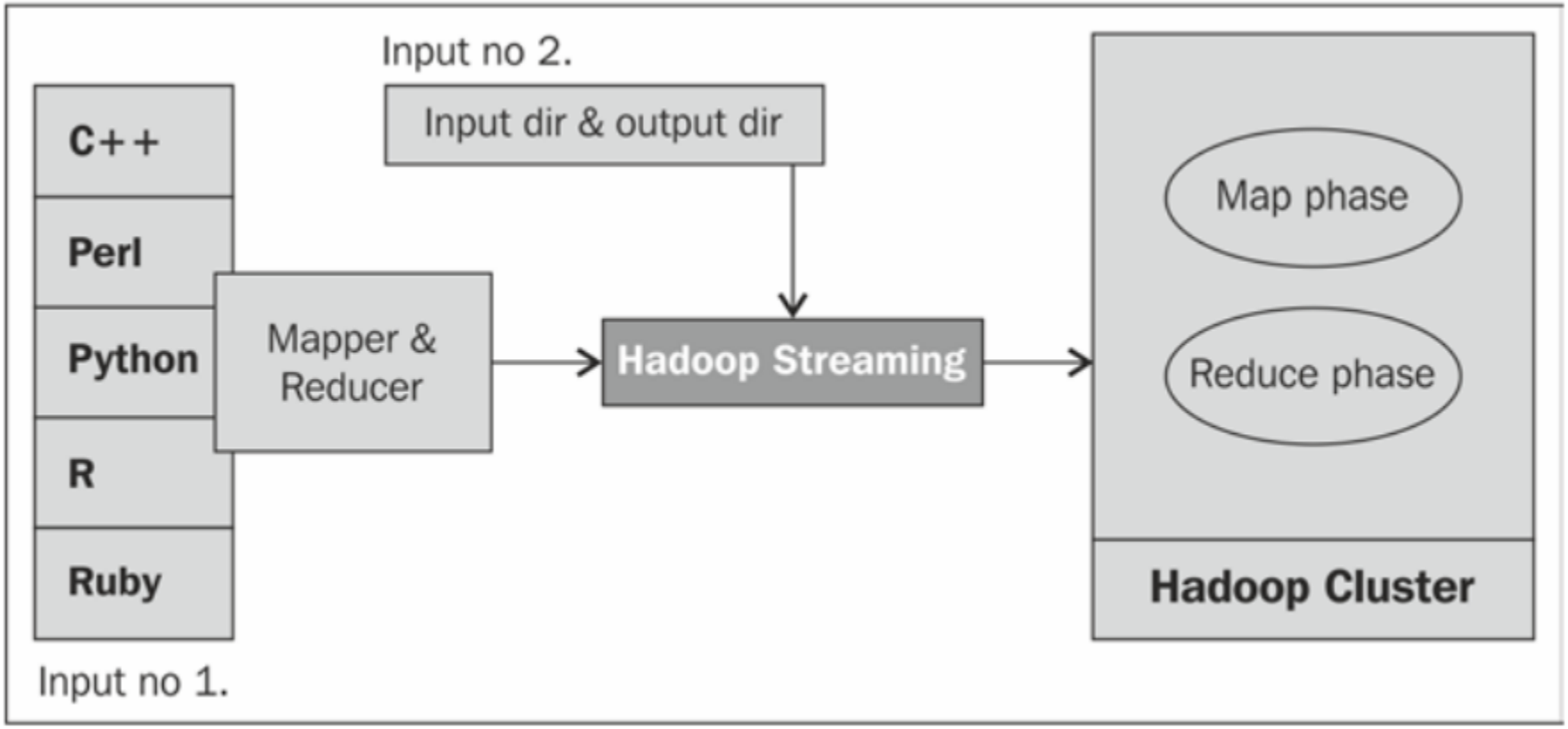
Hadoop Streaming

- Hadoop streaming is a Hadoop utility for running the Hadoop MapReduce job with executable scripts such as Mapper and Reducer.
- This is similar to the pipe operation in Linux.
- With this, the text input file is printed on stream (stdin), which is provided as an input to Mapper and the output (stdout) of Mapper is provided as an input to Reducer; finally, Reducer writes the output to the HDFS directory.

Hadoop Streaming

- The main advantage of the Hadoop streaming utility is that it allows Java as well as non-Java programmed MapReduce jobs to be executed over Hadoop clusters.
- Also, it takes care of the progress of running MapReduce jobs.
- The Hadoop streaming supports the Perl, Python, PHP, R, and C++ programming languages.
- To run an application written in other programming languages, the developer just needs to translate the application logic into the Mapper and Reducer sections with the key and value output elements.

Hadoop Streaming



Hadoop Streaming Command

```
${HADOOP_HOME}/bin/hadoop \
        jar $HADOOP_HOME/contrib/*.jar \                    Line 1
        -input /app/haadoop/input \                         Line 2
        -output /app/haadoop/output \                       Line 3
        -file /usr/local/hadoop/code_mapper.R \             Line 4
        -mapper code_mapper.R \                             Line 5
        -file /usr/local/hadoop/code_reducer.R \            Line 6
        -reducer code_reducer.R                             Line 7
```

mapper.r

```
#! /usr/bin/env Rscript
# mapper.R - Wordcount program in R
# script for Mapper (R-Hadoop integration)

trimWhiteSpace <- function(line) gsub("(^ +)|( +$)", "", line)
splitIntoWords <- function(line) unlist(strsplit(line, "[[:space:]]+"))

## **** could wo with a single readLines or in blocks
con <- file("stdin", open = "r")
while (length(line <- readLines(con, n = 1, warn = FALSE)) > 0) {
    line <- trimWhiteSpace(line)
    words <- splitIntoWords(line)
    ## **** can be done as cat(paste(words, "\t1\n", sep=""), sep="")
    for (w in words)
        cat(w, "\t1\n", sep="")
}
close(con)
```

reducer.r

```
# reducer.R - Wordcount program in R
# script for Reducer (R-Hadoop integration)
trimWhiteSpace <- function(line) gsub("(^ +)|( +$)", "", line)
splitLine <- function(line) {
    val <- unlist(strsplit(line, "\t"))
    list(word = val[1], count = as.integer(val[2]))
}
env <- new.env(hash = TRUE)
con <- file("stdin", open = "r")
while (length(line <- readLines(con, n = 1, warn = FALSE)) > 0) {
    line <- trimWhiteSpace(line)
    split <- splitLine(line)
    word <- split$word
    count <- split$count
    if (exists(word, envir = env, inherits = FALSE)) {
        oldcount <- get(word, envir = env)
        assign(word, oldcount + count, envir = env)
    }
    else assign(word, count, envir = env)
}
close(con)
for (w in ls(env, all = TRUE))
    cat(w, "\t", get(w, envir = env), "\n", sep = "")
```

mapper.py

```
#!/usr/bin/python
import sys
# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)
```

reducer.py

```
#!/usr/bin/python
from operator import itemgetter
import sys
current_word = None
current_count = 0
word = None
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)
    # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        continue
    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word == word:
        current_count += count
    else:
```

Output

```
mitu@skillologies:~/words$ cat fruits.txt | python mapper.py
    | sort | python reducer.py
apple    4
banana   2
grapes   3
mango    2
orange   2
pineapple        1
plum     2
pomgranate       1
raspberry        2
```

Thank you

This presentation is created using LibreOffice Impress 5.1.6.2, can be used freely as per GNU General Public License

@mitu_skillologies

/mITuSkillologies

@mitu_group

Web Resources
http://mitu.co.in
http://tusharkute.com

tushar@tusharkute.com

contact@mitu.co.in