**PROGRAM-1**

**Write a MapReduce script to count the occurrence of each word in a file.**

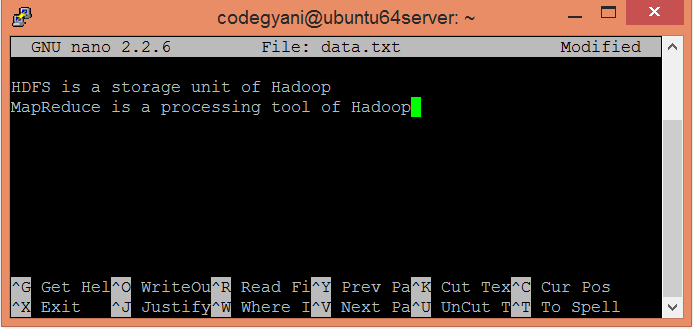
In MapReduce word count, we find out the frequency of each word. Here, the role of Mapper is to map the keys to the existing values and the role of Reducer is to aggregate the keys of common values. So, everything is represented in the form of Key-value pair.

**Pre-requisite**

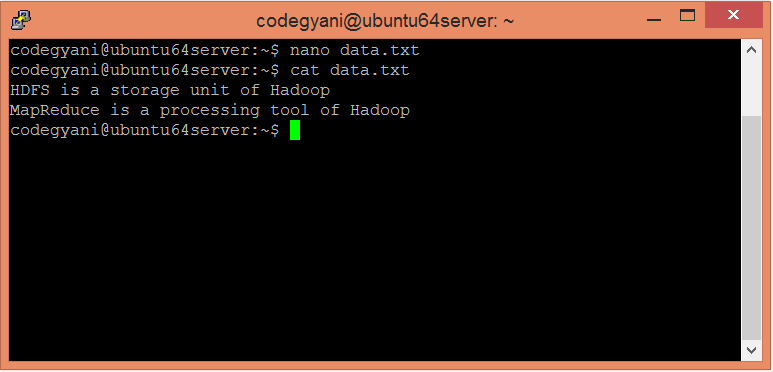
* **Java Installation -** Check whether the Java is installed or not using the following command.  
  java -version
* **Hadoop Installation -** Check whether the Hadoop is installed or not using the following command.  
  hadoop version

**Steps to execute MapReduce word count example**

* Create a text file in our local machine and write some text into it.  
  $ nano data.txt



Check the text written in the data.txt file.   
$ cat data.txt



In this example, we find out the frequency of each word exists in this text file.

* Create a directory in HDFS, where to kept text file.  
  $ hdfs dfs -mkdir /test
* Upload the data.txt file on HDFS in the specific directory.  
  $ hdfs dfs -put /home/codegyani/data.txt /test

### File: WC\_Mapper.java

1. package com.javatpoint;
2. import java.io.IOException;
3. import java.util.StringTokenizer;
4. import org.apache.hadoop.io.IntWritable;
5. import org.apache.hadoop.io.LongWritable;
6. import org.apache.hadoop.io.Text;
7. import org.apache.hadoop.mapred.MapReduceBase;
8. import org.apache.hadoop.mapred.Mapper;
9. import org.apache.hadoop.mapred.OutputCollector;
10. import org.apache.hadoop.mapred.Reporter;
11. public class WC\_Mapper extends MapReduceBase implements Mapper<LongWritable,Text,Text,IntWritable>{
12. private final static IntWritable one = new IntWritable(1);
13. private Text word = new Text();
14. public void map(LongWritable key, Text value,OutputCollector<Text,IntWritable> output,
15. Reporter reporter) throws IOException{
16. String line = value.toString();
17. StringTokenizer  tokenizer = new StringTokenizer(line);
18. while (tokenizer.hasMoreTokens()){
19. word.set(tokenizer.nextToken());
20. output.collect(word, one);
21. }
22. }
24. }

### File: WC\_Reducer.java

1. package com.javatpoint;
2. import java.io.IOException;
3. import java.util.Iterator;
4. import org.apache.hadoop.io.IntWritable;
5. import org.apache.hadoop.io.Text;
6. import org.apache.hadoop.mapred.MapReduceBase;
7. import org.apache.hadoop.mapred.OutputCollector;
8. import org.apache.hadoop.mapred.Reducer;
9. import org.apache.hadoop.mapred.Reporter;
10. public class WC\_Reducer  extends MapReduceBase implements Reducer<Text,IntWritable,Text,
11. IntWritable> {
12. public void reduce(Text key, Iterator<IntWritable> values,OutputCollector<Text,IntWritable>
13. output,
14. Reporter reporter) throws IOException {
15. int sum=0;
16. while (values.hasNext()) {
17. sum+=values.next().get();
18. }
19. output.collect(key,new IntWritable(sum));
20. }
21. }

### File: WC\_Runner.java

1. package com.javatpoint;
3. import java.io.IOException;
4. import org.apache.hadoop.fs.Path;
5. import org.apache.hadoop.io.IntWritable;
6. import org.apache.hadoop.io.Text;
7. import org.apache.hadoop.mapred.FileInputFormat;
8. import org.apache.hadoop.mapred.FileOutputFormat;
9. import org.apache.hadoop.mapred.JobClient;
10. import org.apache.hadoop.mapred.JobConf;
11. import org.apache.hadoop.mapred.TextInputFormat;
12. import org.apache.hadoop.mapred.TextOutputFormat;
13. public class WC\_Runner {
14. public static void main(String[] args) throws IOException{
15. JobConf conf = new JobConf(WC\_Runner.class);
16. conf.setJobName("WordCount");
17. conf.setOutputKeyClass(Text.class);
18. conf.setOutputValueClass(IntWritable.class);
19. conf.setMapperClass(WC\_Mapper.class);
20. conf.setCombinerClass(WC\_Reducer.class);
21. conf.setReducerClass(WC\_Reducer.class);
22. conf.setInputFormat(TextInputFormat.class);
23. conf.setOutputFormat(TextOutputFormat.class);
24. FileInputFormat.setInputPaths(conf,new Path(args[0]));
25. FileOutputFormat.setOutputPath(conf,new Path(args[1]));
26. JobClient.runJob(conf);
27. }
28. }

**PROGRAM-2**

**Write a MapReduce script to find the max and min temperature recorded in last 50 years.**

**Problem Statement**:   
1. The system receives temperatures of various cities(Austin, Boston,etc) of USA captured at regular intervals of time on each day in an input file.   
2. System will process the input data file and generates a report with Maximum and Minimum temperatures of each day along with time in last 50 years.   
3. Generates a separate output report for each city.   
Ex: Austin-r-00000   
 Boston-r-00000   
 Newjersy-r-00000   
 Baltimore-r-00000   
 California-r-00000   
 Newyork-r-00000   
 **Expected output:-**In each output file record should be like this:  
25-Jan-2014 Time: 12:34:542 MinTemp: -22.3 Time: 05:12:345 MaxTemp: 35.7

First download input file which contains temperature statistics with time for multiple cities.Schema of record set : CA\_25-Jan-2014 00:12:345 15.7 01:19:345 23.1 02:34:542 12.3 ......

CA is city code, here it stands for California followed by date. After that each pair of values represent time and temperature.

**Mapper class and map method:-**

The very first thing which is required for any map reduce problem is to understand what will be the type of keyIn, ValueIn, KeyOut,ValueOut for the given Mapper class and followed by type of map method parameters.

* public class WhetherForcastMapper extends Mapper <**Object, Text, Text, Text**>

Object (keyIn) - Offset for each line, line number 1, 2...  
Text **(**ValueIn**) -** Whole string for each line (CA\_25-Jan-2014 00:12:345 ......)  
Text **(**KeyOut**) -** City information with date information as string  
Text **(**ValueOut**) -** Temperature and time information which need to be passed to reducer as string.

* public void map(**Object keyOffset, Text dayReport, Context con**) { }

*KeyOffset* is like line number for each line in input file.  
*dayreport* is input to map method - whole string present in one line of input file.  
*con* is context where we write mapper output and it is used by reducer.

**Reducer class and reducer method:-**

Similarly,we have to decide what will be the type of keyIn, ValueIn, KeyOut,ValueOut for the given Reducer class and followed by type of reducer method parameters.

* public class WhetherForcastReducer extends Reducer<**Text, Text, Text, Text**>

Text(keyIn) - it is same as keyOut of Mapper.  
Text(ValueIn)- it is same as valueOut of Mapper.  
Text(KeyOut)- date as string  
text(ValueOut) - reducer writes max and min temperature with time as string

* public void reduce(**Text key, Iterable<Text> values, Context context**)

Text key is value of mapper output. i.e:- City & date information  
Iterable<Text> values - values stores multiple temperature values for a given city and date.  
context object is where reducer write it's processed outcome and finally written in file.

**MultipleOutputs :-** In general, reducer generates output file(i.e: part\_r\_0000), however in this use case we want to generate multiple output files. In order to deal with such scenario we need to use MultipleOutputs of "org.apache.hadoop.mapreduce.lib.output.MultipleOutputs" which provides a way to write multiple file depending on reducer outcome. See below reducer class for more details.For each reducer task multipleoutput object is created and key/result is written to appropriate file.

Lets [create a Map/Reduce project in eclipse](http://www.devinline.com/2015/10/how-to-run-mapreduce-program-in-eclipse.html) and create a class file name it as CalculateMaxAndMinTemeratureWithTime. For simplicity,here we have written mapper and reducer class as inner static class.

**import** **java.io.IOException**;

**import** **java.util.StringTokenizer**;

**import** **org.apache.hadoop.io.Text**;

**import** **org.apache.hadoop.mapreduce.Mapper**;

**import** **org.apache.hadoop.mapreduce.Reducer**;

**import** **org.apache.hadoop.mapreduce.lib.output.MultipleOutputs**;

**import** **org.apache.hadoop.conf.Configuration**;

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

**import** **org.apache.hadoop.mapreduce.Job**;

**import** **org.apache.hadoop.mapreduce.lib.input.FileInputFormat**;

**import** **org.apache.hadoop.mapreduce.lib.output.FileOutputFormat**;

**import** **org.apache.hadoop.mapreduce.lib.output.TextOutputFormat**;

**public** **class** **CalculateMaxAndMinTemeratureWithTime** {

**public** **static** String calOutputName = "California";

**public** **static** String nyOutputName = "Newyork";

**public** **static** String njOutputName = "Newjersy";

**public** **static** String ausOutputName = "Austin";

**public** **static** String bosOutputName = "Boston";

**public** **static** String balOutputName = "Baltimore";

**public** **static** **class** **WhetherForcastMapper** **extends**

Mapper<Object, Text, Text, Text> {

**public** **void** **map**(Object keyOffset, Text dayReport, Context con)

**throws** IOException, InterruptedException {

StringTokenizer strTokens = **new** StringTokenizer(

dayReport.toString(), "\t");

**int** counter = **0**;

Float currnetTemp = **null**;

Float minTemp = Float.MAX\_VALUE;

Float maxTemp = Float.MIN\_VALUE;

String date = **null**;

String currentTime = **null**;

String minTempANDTime = **null**;

String maxTempANDTime = **null**;

**while** (strTokens.hasMoreElements()) {

**if** (counter == **0**) {

date = strTokens.nextToken();

} **else** {

**if** (counter % **2** == **1**) {

currentTime = strTokens.nextToken();

} **else** {

currnetTemp = Float.parseFloat(strTokens.nextToken());

**if** (minTemp > currnetTemp) {

minTemp = currnetTemp;

minTempANDTime = minTemp + "AND" + currentTime;

}

**if** (maxTemp < currnetTemp) {

maxTemp = currnetTemp;

maxTempANDTime = maxTemp + "AND" + currentTime;

}

}

}

counter++;

}

// Write to context - MinTemp, MaxTemp and corresponding time

Text temp = **new** Text();

temp.set(maxTempANDTime);

Text dateText = **new** Text();

dateText.set(date);

**try** {

con.write(dateText, temp);

} **catch** (Exception e) {

e.printStackTrace();

}

temp.set(minTempANDTime);

dateText.set(date);

con.write(dateText, temp);

}

}

**public** **static** **class** **WhetherForcastReducer** **extends**

Reducer<Text, Text, Text, Text> {

MultipleOutputs<Text, Text> mos;

**public** **void** **setup**(Context context) {

mos = **new** MultipleOutputs<Text, Text>(context);

}

**public** **void** **reduce**(Text key, Iterable<Text> values, Context context)

**throws** IOException, InterruptedException {

**int** counter = **0**;

String reducerInputStr[] = **null**;

String f1Time = "";

String f2Time = "";

String f1 = "", f2 = "";

Text result = **new** Text();

**for** (Text value : values) {

**if** (counter == **0**) {

reducerInputStr = value.toString().split("AND");

f1 = reducerInputStr[**0**];

f1Time = reducerInputStr[**1**];

}

**else** {

reducerInputStr = value.toString().split("AND");

f2 = reducerInputStr[**0**];

f2Time = reducerInputStr[**1**];

}

counter = counter + **1**;

}

**if** (Float.parseFloat(f1) > Float.parseFloat(f2)) {

result = **new** Text("Time: " + f2Time + " MinTemp: " + f2 + "\t"

+ "Time: " + f1Time + " MaxTemp: " + f1);

} **else** {

result = **new** Text("Time: " + f1Time + " MinTemp: " + f1 + "\t"

+ "Time: " + f2Time + " MaxTemp: " + f2);

}

String fileName = "";

**if** (key.toString().substring(**0**, **2**).equals("CA")) {

fileName = CalculateMaxAndMinTemeratureTime.calOutputName;

} **else** **if** (key.toString().substring(**0**, **2**).equals("NY")) {

fileName = CalculateMaxAndMinTemeratureTime.nyOutputName;

} **else** **if** (key.toString().substring(**0**, **2**).equals("NJ")) {

fileName = CalculateMaxAndMinTemeratureTime.njOutputName;

} **else** **if** (key.toString().substring(**0**, **3**).equals("AUS")) {

fileName = CalculateMaxAndMinTemeratureTime.ausOutputName;

} **else** **if** (key.toString().substring(**0**, **3**).equals("BOS")) {

fileName = CalculateMaxAndMinTemeratureTime.bosOutputName;

} **else** **if** (key.toString().substring(**0**, **3**).equals("BAL")) {

fileName = CalculateMaxAndMinTemeratureTime.balOutputName;

}

String strArr[] = key.toString().split("\_");

key.set(strArr[**1**]); //Key is date value

mos.write(fileName, key, result);

}

**@Override**

**public** **void** **cleanup**(Context context) **throws** IOException,

InterruptedException {

mos.close();

}

}

**public** **static** **void** **main**(String[] args) **throws** IOException,

ClassNotFoundException, InterruptedException {

Configuration conf = **new** Configuration();

Job job = Job.getInstance(conf, "Wheather Statistics of USA");

job.setJarByClass(CalculateMaxAndMinTemeratureWithTime.class);

job.setMapperClass(WhetherForcastMapper.class);

job.setReducerClass(WhetherForcastReducer.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(Text.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(Text.class);

MultipleOutputs.addNamedOutput(job, calOutputName,

TextOutputFormat.class, Text.class, Text.class);

MultipleOutputs.addNamedOutput(job, nyOutputName,

TextOutputFormat.class, Text.class, Text.class);

MultipleOutputs.addNamedOutput(job, njOutputName,

TextOutputFormat.class, Text.class, Text.class);

MultipleOutputs.addNamedOutput(job, bosOutputName,

TextOutputFormat.class, Text.class, Text.class);

MultipleOutputs.addNamedOutput(job, ausOutputName,

TextOutputFormat.class, Text.class, Text.class);

MultipleOutputs.addNamedOutput(job, balOutputName,

TextOutputFormat.class, Text.class, Text.class);

// FileInputFormat.addInputPath(job, new Path(args[0]));

// FileOutputFormat.setOutputPath(job, new Path(args[1]));

Path pathInput = **new** Path(

"hdfs://192.168.213.133:54310/weatherInputData/input\_temp.txt");

Path pathOutputDir = **new** Path(

"hdfs://192.168.213.133:54310/user/hduser1/testfs/output\_mapred3");

FileInputFormat.addInputPath(job, pathInput);

FileOutputFormat.setOutputPath(job, pathOutputDir);

**try** {

System.exit(job.waitForCompletion(**true**) ? **0** : **1**);

} **catch** (Exception e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

**Explanation**:-   
In **map method**, we are parsing each input line and maintains a counter for extracting date and each temperature & time information.For a given input line, first extract date(counter ==0) and followed by alternatively extract time(counter%2==1) since time is on odd number position like (1,3,5....) and get temperature otherwise. Compare for max & min temperature and store it accordingly. Once while loop terminates for a given input line, write maxTempTime and minTempTime with date.  
In **reduce method**, for each reducer task, setup method is executed and create MultipleOutput object. For a given key, we have two entry (maxtempANDTime and mintempANDTime). Iterate values list , split value and get temperature & time value. Compare temperature value and create actual value sting which reducer write in appropriate file.  
In **main method**,a instance of Job is created with Configuration object. Job is configured with mapper, reducer class and along with input and output format. MultipleOutputs information added to Job to indicate file name to be used with input format. For this sample program, we are using input file("/weatherInputData/input\_temp.txt") placed on HDFS and output directory (/user/hduser1/testfs/output\_mapred5) will be also created on HDFS. Refer below command to copy downloaded input file from local file system to HDFS and  give write permission to client who is executing this program unit so that output directory can be created.  
**Copy a input file form local file system to HDFS**

**hduser1@ubuntu:/usr/local/hadoop2.6.1/bin$** ./hadoop fs -put /home/zytham/input\_temp.txt /weatherInputData/

**Give write permission to all user for creating output directory**

**hduser1@ubuntu:/usr/local/hadoop2.6.1/bin$** ./hadoop fs -chmod -R 777 /user/hduser1/testfs/

Before executing above program unit make sure hadoop services are running(to start all service execute ./start-all.sh from <hadoop\_home>/sbin).  
Now execute above sample program. Run -> Run as hadoop. Wait for a moment and check whether output directory is in place on HDFS. Execute following command to verify the same.

**hduser1@ubuntu:/usr/local/hadoop2.6.1/bin$** ./hadoop fs -ls /user/hduser1/testfs/output\_mapred3

Found 8 items

-rw-r--r-- 3 zytham supergroup 438 2015-12-11 19:21 /user/hduser1/testfs/output\_mapred3/**Austin-r-00000**

-rw-r--r-- 3 zytham supergroup 219 2015-12-11 19:21 /user/hduser1/testfs/output\_mapred3/**Baltimore-r-00000**

-rw-r--r-- 3 zytham supergroup 219 2015-12-11 19:21 /user/hduser1/testfs/output\_mapred3/**Boston-r-00000**

-rw-r--r-- 3 zytham supergroup 511 2015-12-11 19:21 /user/hduser1/testfs/output\_mapred3/**California-r-00000**

-rw-r--r-- 3 zytham supergroup 146 2015-12-11 19:21 /user/hduser1/testfs/output\_mapred3/**Newjersy-r-00000**

-rw-r--r-- 3 zytham supergroup 219 2015-12-11 19:21 /user/hduser1/testfs/output\_mapred3/**Newyork-r-00000**

-rw-r--r-- 3 zytham supergroup 0 2015-12-11 19:21 /user/hduser1/testfs/output\_mapred3/\_SUCCESS

-rw-r--r-- 3 zytham supergroup 0 2015-12-11 19:21 /user/hduser1/testfs/output\_mapred3/part-r-00000

Open one of the file and verify expected output schema, execute following command for the same.

**hduser1@ubuntu:/usr/local/hadoop2.6.1/bin$** ./hadoop fs -cat /user/hduser1/testfs/output\_mapred3/Austin-r-00000

**25-Jan-2014 Time: 12:34:542 MinTemp: -22.3 Time: 05:12:345 MaxTemp: 35.7**

26-Jan-2014 Time: 22:00:093 MinTemp: -27.0 Time: 05:12:345 MaxTemp: 55.7

27-Jan-2014 Time: 02:34:542 MinTemp: -22.3 Time: 05:12:345 MaxTemp: 55.7

29-Jan-2014 Time: 14:00:093 MinTemp: -17.0 Time: 02:34:542 MaxTemp: 62.9

30-Jan-2014 Time: 22:00:093 MinTemp: -27.0 Time: 05:12:345 MaxTemp: 49.2

31-Jan-2014 Time: 14:00:093 MinTemp: -17.0 Time: 03:12:187 MaxTemp: 56.0

**Note**:-

* In order to reference input file from local file system instead of HDFS, uncomment below lines in main method and comment below added addInputPath and setOutputPath lines. Here Path(args[0]) and Path(args[1]) [read input and output location path from program arguments](http://www.devinline.com/2015/10/program-and-vm-arguments-in-java.html). OR create path object with sting input of input file and output location.  
   // FileInputFormat.addInputPath(job, new Path(args[0]));  
  //  FileOutputFormat.setOutputPath(job, new Path(args[1]));

**Execute WeatherReportPOC.jar on single node cluster**

We can create jar file out of this project and run on single node cluster too. [Download WeatherReportPOC jar](https://drive.google.com/file/d/0B-ur4R5mlgGLMzVyTmdITTVmbjA/view?usp=sharing) and place at some convenient location.Start hadoop services(./start-all.sh from <hadoop\_home>/sbin). I have placed jar at "/home/zytham/Downloads/WeatherReportPOC.jar".   
Execute following command to submit job with input file HDFS location is "/wheatherInputData/input\_temp.txt" and output directory location is "/user/hduser1/testfs/output\_mapred7"

**hduser1@ubuntu:/usr/local/hadoop2.6.1/bin$** ./hadoop jar /home/zytham/Downloads/WeatherReportPOC.jar **CalculateMaxAndMinTemeratureWithTime /wheatherInputData/input\_temp.txt** **/user/hduser1/testfs/output\_mapred7**

15/12/11 22:16:12 INFO Configuration.deprecation: session.id is deprecated. Instead, use dfs.metrics.session-id

15/12/11 22:16:12 INFO jvm.JvmMetrics: Initializing JVM Metrics with processName=JobTracker, sessionId=

15/12/11 22:16:14 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to remedy this.

15/12/11 22:16:26 INFO output.FileOutputCommitter: Saved output of task 'attempt\_local1563851561\_0001\_r\_000000\_0' to hdfs://hostname:54310/user/hduser1/testfs/output\_mapred7/\_temporary/0/task\_local1563851561\_0001\_r\_000000

15/12/11 22:16:26 INFO mapred.LocalJobRunner: reduce > reduce

15/12/11 22:16:26 INFO mapred.Task: Task 'attempt\_local1563851561\_0001\_r\_000000\_0' done.

15/12/11 22:16:26 INFO mapred.LocalJobRunner: Finishing task: attempt\_local1563851561\_0001\_r\_000000\_0

15/12/11 22:16:26 INFO mapred.LocalJobRunner: reduce task executor complete.

15/12/11 22:16:26 INFO mapreduce.Job: map 100% reduce 100%

15/12/11 22:16:27 INFO mapreduce.Job: Job job\_local1563851561\_0001 completed successfully

15/12/11 22:16:27 INFO mapreduce.Job: Counters: 38

......

**Output:-**In each output file record should be like this:  
25-Jan-2014 Time: 12:34:542 MinTemp: -22.3 Time: 05:12:345 MaxTemp: 35.7

**PROGRAM-3**

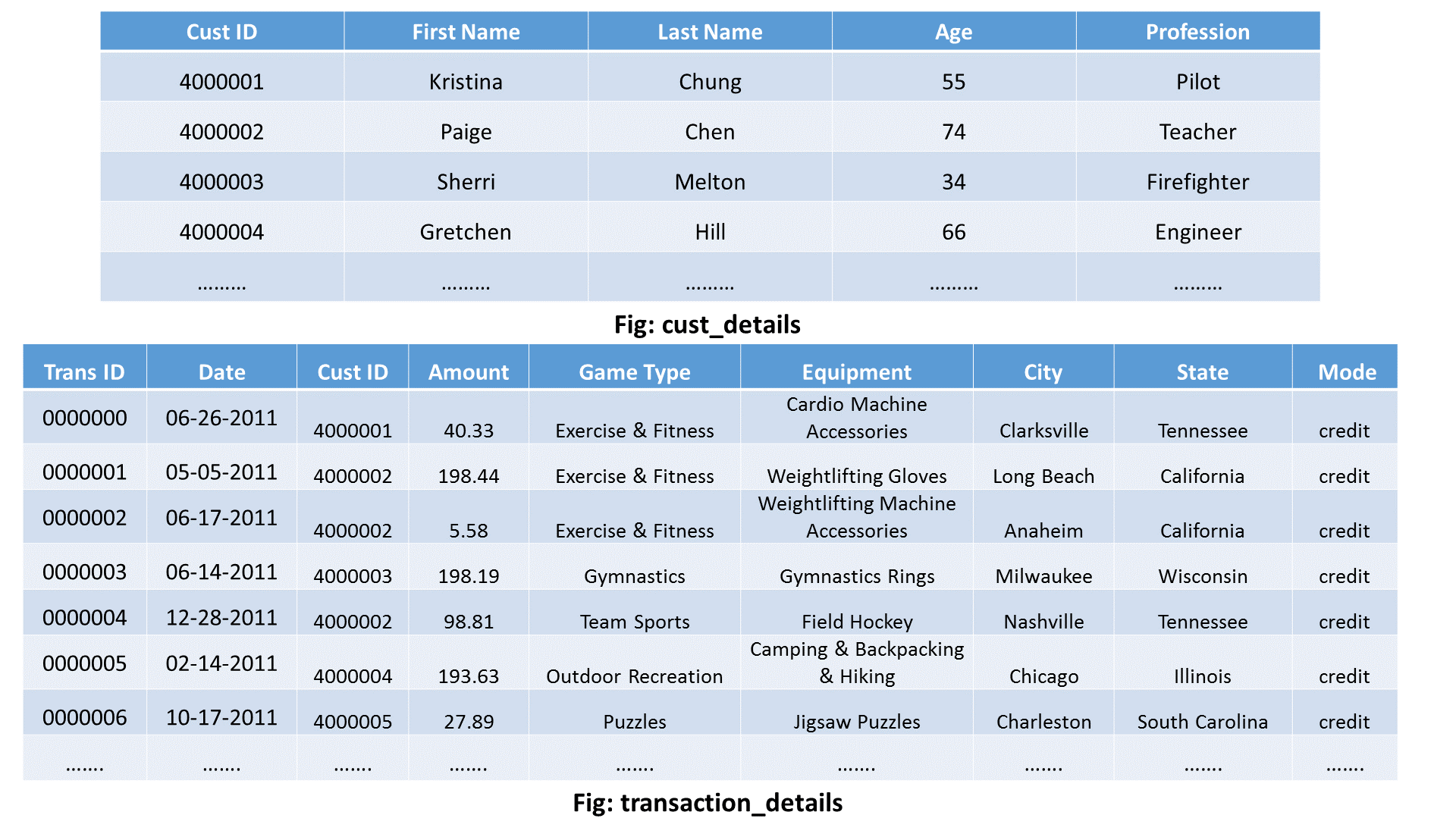
**Write a MapReduce script to get the name of the customer, how many transactions a customer has made and how much amount of transactions a particular customer has made.**

Suppose that we have two separate datasets of a sports complex:

* **cust\_details:** It contains the details of the customer.
* **transaction\_details:** It contains the transaction record of the customer.

Using these two datasets, we want to know the lifetime value of each customer. In doing so, we will be needing the following things:

* The person’s name along with the frequency of the visits by that person.
* The total amount spent by him/her for purchasing the equipment.



The above figure is just to show us the schema of the two datasets on which we will perform the reduce side join operation.

Kindly, keep the following things in mind while importing the above MapReduce example project on reduce side join into Eclipse:

* The input files are in input\_files directory of the project. Load these into our HDFS.
* Don’t forget to build the path of Hadoop Reference Jars (present in reduce side join project lib directory)  according to our system or VM.

Now, let us understand what happens inside the map and reduce phases in this MapReduce example on reduce side join:

### ****1. Map Phase:****

We will have a separate mapper for each of the two datasets i.e. One mapper for cust\_details input and other for transaction\_details input.

***Mapper for cust\_details:***

public static class CustsMapper extends Mapper <Object, Text, Text, Text>

{

public void map(Object key, Text value, Context context) throws IOException,

InterruptedException

{

String record = value.toString();

String[] parts = record.split(",");

context.write(new Text(parts[0]), new Text("cust " + parts[1]));

}

}

* I will read the input taking one tuple at a time.
* Then,wewill tokenize each word in that tuple and fetch the cust ID along with the name of the person.
* The cust ID will be my key of the key-value pair that my mapper will generate eventually.
* I will also add a tag “cust” to indicate that this input tuple is of cust\_details type.
* Therefore, my mapper for cust\_details will produce following intermediate key-value pair:

**Key – Value pair: [cust ID, cust        name]**

Example: [4000001, cust    Kristina], [4000002, cust   Paige], etc.

***Mapper for transaction\_details:***

public static class TxnsMapper extends Mapper <Object, Text, Text, Text>

{

public void map(Object key, Text value, Context context) throws IOException,

InterruptedException

{

String record = value.toString();

String[] parts = record.split(",");

context.write(new Text(parts[2]), new Text("tnxn " + parts[3]));

}

}

* Like mapper for cust\_details,wewill follow the similar steps here. Though, there will be a few differences:
  + I will fetch the amount value instead of name of the person.
  + In this case, we will use “tnxn” as a tag.
* Therefore, the cust ID will be my key of the key-value pair that the mapper will generate eventually.
* Finally, the output of my mapper for transaction\_details will be of the following format:

**Key, Value Pair: [cust ID, tnxn   amount]**

Example: [4000001, tnxn   40.33], [4000002, tnxn   198.44], etc.

### ****2. Sorting and Shuffling Phase****

The sorting and shuffling phase will generate an array list of values corresponding to each key. In other words, it will put together all the values corresponding to each unique key in the intermediate key-value pair. The output of sorting and shuffling phase will be of the following format:

**Key – list of Values:**

* {cust ID1 – [(cust    name1), (tnxn    amount1), (tnxn    amount2), (tnxn    amount3),…..]}
* {cust ID2 – [(cust    name2), (tnxn    amount1), (tnxn    amount2), (tnxn    amount3),…..]}
* ……
* {4000001 – [(cust    kristina), (tnxn    40.33), (tnxn    47.05),…]};
* {4000002 – [(cust    paige), (tnxn    198.44), (tnxn     5.58),…]};
* ……

Now, the framework will call reduce() method (reduce(Text key, Iterable<Text> values, Context context)) for each unique join key (cust id) and the corresponding list of values. Then, the reducer will perform the join operation on the values present in the respective list of values to calculate the desired output eventually. Therefore, the number of reducer task performed will be equal to the number of unique cust ID.

Let us now understand how the reducer performs the join operation in this MapReduce example.

### ****3. Reducer Phase****

If we remember, the primary goal to perform this reduce-side join operation was to find out that how many times a particular customer has visited sports complex and the total amount spent by that very customer on different sports. Therefore, my final output should be of the following format:

**Key – Value pair: [Name of the customer] (Key) – [total amount, frequency of the visit] (Value)**

### Reducer Code:

**public static class ReduceJoinReducer extends Reducer <Text, Text, Text, Text>**

**{**

**public void reduce(Text key, Iterable<Text> values, Context context)**

**throws IOException, InterruptedException**

**{**

**String name = "";**

**double total = 0.0;**

**int count = 0;**

**for (Text t : values)**

**{**

**String parts[] = t.toString().split(" ");**

**if (parts[0].equals("tnxn"))**

**{**

**count++;**

**total += Float.parseFloat(parts[1]);**

**}**

**else if (parts[0].equals("cust"))**

**{**

**name = parts[1];**

**}**

**}**

**String str = String.format("%d %f", count, total);**

**context.write(new Text(name), new Text(str));**

**}}**

So, following steps will be taken in each of the reducers to achieve the desired output:

* In each of the reducerwewill have a key & list of values where the key is nothing but the cust ID. The list of values will have the input from both the datasets i.e. Amount from transaction\_details and name from cust\_details.
* Now,wewill loop through the values present in the list of values in the reducer.
* Then,we will split the list of values and check whether the value is of transaction\_details type or cust\_details type.
* If it is of the transaction\_details type,we will perform the following steps:
  + I will increase the counter value by one to calculate the frequency of visit by the very person.
  + I will cumulatively update the amount value to calculate the total amount spent by that person.
* On the other hand, if the value is of cust\_details type,we will store it in a string variable. Later,we will assign the name as my key  in my output key-value pair.
* Finally,we will write the output key-value pair in the output folder in my HDFS.

Hence, the final output that my reducer will generate is given below:

**Kristina, 651.05 8**

**Paige, 706.97  6**

**…..**

And, this whole process that we did above is called ***Reduce Side Join*** in MapReduce.

### ****Source Code:****

The source code for the above MapReduce example of the reduce side join is given below:

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

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.MultipleInputs;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class ReduceJoin {

public static class CustsMapper extends Mapper <Object, Text, Text, Text>

{

public void map(Object key, Text value, Context context)

throws IOException, InterruptedException

{

String record = value.toString();

String[] parts = record.split(",");

context.write(new Text(parts[0]), new Text("cust " + parts[1]));

}

}

public static class TxnsMapper extends Mapper <Object, Text, Text, Text>

{

public void map(Object key, Text value, Context context)

throws IOException, InterruptedException

{

String record = value.toString();

String[] parts = record.split(",");

context.write(new Text(parts[2]), new Text("tnxn " + parts[3]));

}

}

public static class ReduceJoinReducer extends Reducer <Text, Text, Text, Text>

{

public void reduce(Text key, Iterable<Text> values, Context context)

throws IOException, InterruptedException

{

String name = "";

double total = 0.0;

int count = 0;

for (Text t : values)

{

String parts[] = t.toString().split(" ");

if (parts[0].equals("tnxn"))

{

count++;

total += Float.parseFloat(parts[1]);

}

else if (parts[0].equals("cust"))

{

name = parts[1];

}

}

String str = String.format("%d %f", count, total);

context.write(new Text(name), new Text(str));

}}

public static void main(String[] args) throws Exception

{

Configuration conf = new Configuration();

Job job = new Job(conf, "Reduce-side join");

job.setJarByClass(ReduceJoin.class);

job.setReducerClass(ReduceJoinReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(Text.class);

MultipleInputs.addInputPath(job, new Path(args[0]),TextInputFormat.class,

CustsMapper.class);

MultipleInputs.addInputPath(job, new Path(args[1]),TextInputFormat.class,

TxnsMapper.class);

Path outputPath = new Path(args[2]);

FileOutputFormat.setOutputPath(job, outputPath);

outputPath.getFileSystem(conf).delete(outputPath);

System.exit(job.waitForCompletion(true) ? 0 : 1);

}}

**Run this Program**

Finally, the command to run the above MapReduce example program on reduce side join is given below:

*hadoop jar reducejoin.jar ReduceJoin /sample/input/cust\_details /sample/input/transaction\_details /sample/output*

# PROGRAM-4

# Analyze document database such as MongoDB using CRUD operations.

CRUD operations *create*, *read*, *update*, and *delete* [documents](https://docs.mongodb.com/manual/core/document/#bson-document-format).

**Create Operations**

Create or insert operations add new [documents](https://docs.mongodb.com/manual/core/document/#bson-document-format) to a [collection](https://docs.mongodb.com/manual/core/databases-and-collections/#collections). If the collection does not currently exist, insert operations will create the collection.

MongoDB provides the following methods to insert documents into a collection:

* [db.collection.insertOne()](https://docs.mongodb.com/manual/reference/method/db.collection.insertOne/#db.collection.insertOne) *New in version 3.2*
* [db.collection.insertMany()](https://docs.mongodb.com/manual/reference/method/db.collection.insertMany/#db.collection.insertMany) *New in version 3.2*

In MongoDB, insert operations target a single [collection](https://docs.mongodb.com/manual/reference/glossary/#term-collection). All write operations in MongoDB are [atomic](https://docs.mongodb.com/manual/core/write-operations-atomicity/) on the level of a single [document](https://docs.mongodb.com/manual/core/document/).

For examples, see [Insert Documents](https://docs.mongodb.com/manual/tutorial/insert-documents/).

**Read Operations**

Read operations retrieves [documents](https://docs.mongodb.com/manual/core/document/#bson-document-format) from a [collection](https://docs.mongodb.com/manual/core/databases-and-collections/#collections); i.e. queries a collection for documents. MongoDB provides the following methods to read documents from a collection:

* [db.collection.find()](https://docs.mongodb.com/manual/reference/method/db.collection.find/#db.collection.find)

You can specify [query filters or criteria](https://docs.mongodb.com/manual/tutorial/query-documents/#read-operations-query-argument) that identify the documents to return.

For examples, see:

* [Query Documents](https://docs.mongodb.com/manual/tutorial/query-documents/)
* [Query on Embedded/Nested Documents](https://docs.mongodb.com/manual/tutorial/query-embedded-documents/)
* [Query an Array](https://docs.mongodb.com/manual/tutorial/query-arrays/)
* [Query an Array of Embedded Documents](https://docs.mongodb.com/manual/tutorial/query-array-of-documents/)

**Update Operations**

Update operations modify existing [documents](https://docs.mongodb.com/manual/core/document/#bson-document-format) in a [collection](https://docs.mongodb.com/manual/core/databases-and-collections/#collections). MongoDB provides the following methods to update documents of a collection:

* [db.collection.updateOne()](https://docs.mongodb.com/manual/reference/method/db.collection.updateOne/#db.collection.updateOne) *New in version 3.2*
* [db.collection.updateMany()](https://docs.mongodb.com/manual/reference/method/db.collection.updateMany/#db.collection.updateMany) *New in version 3.2*
* [db.collection.replaceOne()](https://docs.mongodb.com/manual/reference/method/db.collection.replaceOne/#db.collection.replaceOne) *New in version 3.2*

In MongoDB, update operations target a single collection. All write operations in MongoDB are [atomic](https://docs.mongodb.com/manual/core/write-operations-atomicity/) on the level of a single document.

You can specify criteria, or filters, that identify the documents to update. These [filters](https://docs.mongodb.com/manual/core/document/#document-query-filter) use the same syntax as read operations.

For examples, see [Update Documents](https://docs.mongodb.com/manual/tutorial/update-documents/).

**Delete Operations**

Delete operations remove documents from a collection. MongoDB provides the following methods to delete documents of a collection:

* [db.collection.deleteOne()](https://docs.mongodb.com/manual/reference/method/db.collection.deleteOne/#db.collection.deleteOne) *New in version 3.2*
* [db.collection.deleteMany()](https://docs.mongodb.com/manual/reference/method/db.collection.deleteMany/#db.collection.deleteMany) *New in version 3.2*

In MongoDB, delete operations target a single [collection](https://docs.mongodb.com/manual/reference/glossary/#term-collection). All write operations in MongoDB are [atomic](https://docs.mongodb.com/manual/core/write-operations-atomicity/) on the level of a single document.

You can specify criteria, or filters, that identify the documents to remove. These [filters](https://docs.mongodb.com/manual/core/document/#document-query-filter) use the same syntax as read operations.

.

**PROGRAM-5**

**Write a Hive script to analyze last 10 years of crime data.**

**Preparing the data for Hive**

Firstly we download the dataset from the following location:

<https://www.kaggle.com/jboysen/london-crime?source=post_page-----2137532dc8db----------->

The first thing we need to do is get the headers from the file:

suran@Kenobi:~/Documents$ head -1 london\_crime\_by\_lsoa.csv   
lsoa\_code, borough, major\_category, minor\_category,value, year, month

We are only concerned with the first six columns , a little unix command line wizardry should help:

suran@Kenobi:~/Documents$ cut -d, -f1-6 london\_crime\_by\_lsoa.csv >london\_crime\_data.csv  
suran@Kenobi:~/Documents$ head -1 london\_crime\_data.csv   
lsoa\_code,borough,major\_category,minor\_category,value,year

and now we finally strip the headers from the csv file as we do not need it when loading the data into the hive table.

suran@Kenobi:~/Documents$ sed '1d' london\_crime\_data.csv >london\_crime\_clean.csv  
suran@Kenobi:~/Documents$ head -1 london\_crime\_clean.csv   
E01001116,Croydon,Burglary,Burglary in Other Buildings,0,2016

Now with our clean csv file in place, lets create our hive table and load the csv file. Firstly, let’s create an external table so we can load the csv file, after that we create an internal table and load the data from the external table.

hive> create database london\_crimes;  
OK  
Time taken: 5.054 seconds  
hive> use london\_crimes;  
OK  
Time taken: 0.046 seconds  
hive> create table if not exists crimes (  
> lsoa\_code string,  
> borough string,  
> major\_category string,  
> minor\_category string,  
> value int,  
> year int)  
> row format delimited  
> fields terminated by ','  
> stored as textfile;  
OK  
Time taken: 0.633 seconds

**Load the csv data:**

hive> load data local inpath '/home/suran/Documents/london\_crime\_clean.csv' overwrite into table crimes;  
Loading data to table london\_crimes.crimes  
OK  
Time taken: 14.429 seconds  
hive> select \* from crimes limit 10;  
OK  
E01001116 Croydon Burglary Burglary in Other Buildings 0 2016  
E01001646 Greenwich Violence Against the Person Other violence 0 2016  
E01000677 Bromley Violence Against the Person Other violence 0 2015  
....  
Time taken: 1.935 seconds, Fetched: 10 row(s)

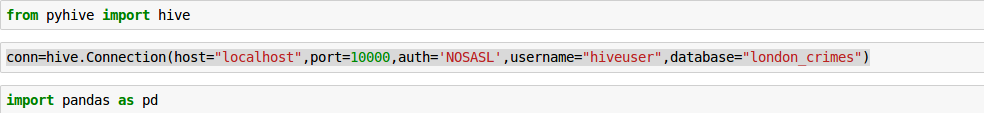
**Finally we can create the hive managed orc table:**

hive> create table londoncrimes stored as orc as  
> select \* from crimes;  
.......  
Job running in-process (local Hadoop)  
2018-02-07 23:18:17,272 Stage-3 map = 100%, reduce = 0%  
Ended Job = job\_local1010979679\_0006  
Moving data to directory hdfs://localhost:54310/user/hive/warehouse/london\_crimes.db/londoncrimes  
MapReduce Jobs Launched:   
Stage-Stage-1: HDFS Read: 9977052694 HDFS Write: 4266210090 SUCCESS  
Stage-Stage-3: HDFS Read: 2829480705 HDFS Write: 1145913018 SUCCESS  
Total MapReduce CPU Time Spent: 0 msec  
OK  
Time taken: 44.978 seconds  
hive> select \* from londoncrimes limit 10;  
OK  
E01001116 Croydon Burglary Burglary in Other Buildings 0 2016  
E01001646 Greenwich Violence Against the Person Other violence 0 2016  
...  
Time taken: 0.254 seconds, Fetched: 10 row(s)

In order to analyse the data directly from hive we first need to install the pyhive and pandas toolkit:

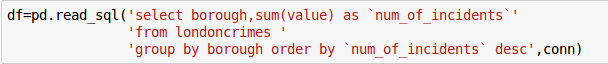
**pip install pyhive  
pip install pandas**

We’ll be using jupyter notebook as this gives me an interactive python shell where we can see results on the fly and also save the contents of the notebook for reference at a later date.

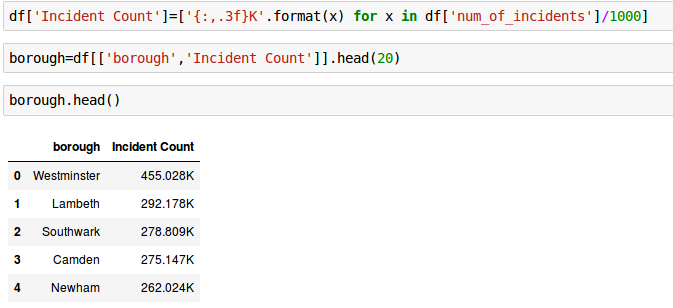


The above code is telling python to load the hive module from the pyhive library and then create a connection to the hive database “london\_crimes” and finally import the pandas library.

**Total Number of Crimes by Borough between 2008–2016**

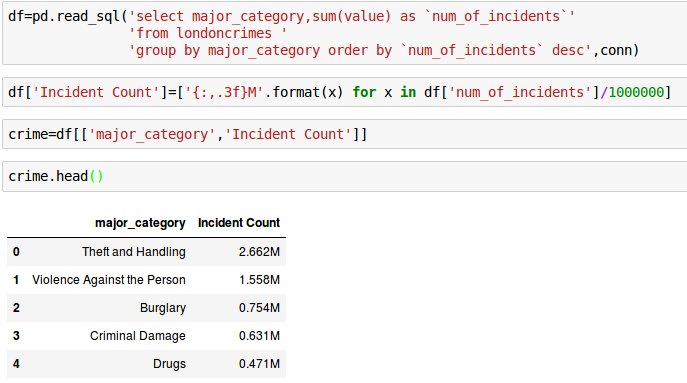


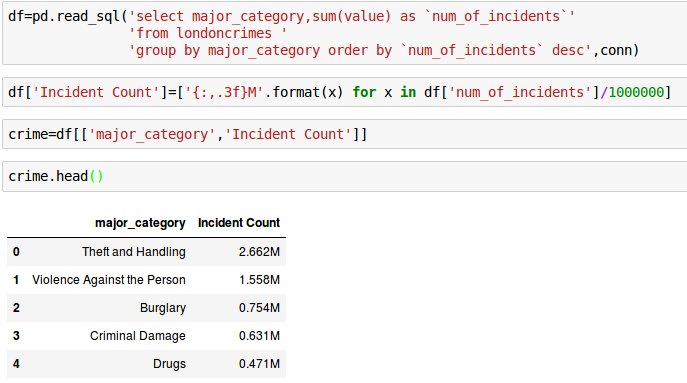
in the code snippet above we create a df (dataframe) that consists of an sql query to the London crimes hive table.



as we ordered it in descending order in our query, we can see Westminster is the borough with the most number of reported crimes. The head lists only the first 5 records.

**Which crime occured the most?**





**Group**

Pandas allows us to group our data much the same way as group by in SQL. Suppose we wanted to group by ‘borough and incident’ well we simply use the group by function:

