LAB 1 - MAP/REDUCE
SCIENTIFIC APPLICATIONS FOR IOT WORKSHOP

March 23rd 2015

The objectives of this session are:

- Understand the Map Reduce programming model
- Familiarize with the Hadoop and HDFS APIs
- Setup a simple Hadoop MapReduce project
- Run Hadoop jobs

INTRODUCTION

The instructions contained in this sheet assume you are using a Hadoop environment preconfigured as in the Cloudera QuickStart VM version. You can complete this lab from your laptop (needs to have a reasonable amount of memory and 64bit OS), by downloading the VM from this link.

This lab sheet uses Apache Ant for automating the build process of the MapReduce project. For a complete reference on the ant building system you can check http://ant.apache.org
We recommend you use Eclipse as the development IDE for managing your Java projects, and speeding up development. However, the project must be built and packaged using the Ant script. When you create a Hadoop project in Eclipse, create a new Java project with the detault options. Then, right click on the project, and select Add External Libraries... Go to the **`/usr/lib/hadoop/client`** folder, select all of them, and add them to the project. This will allow you to see compilation errors in hadoop. For generating the Hadoop executable you **must** use the provided Ant script.

PART 0: HDFS SETUP

Hadoop clusters use a distributed filesystem (HDFS) to store all the input and output data with a replication factor. The Cloudera Quickstart VM has a running HDFS service, although for obvious reasons there is no replication involved. The first section of this lab aims to familiarize you with HDFS commands.
HDFS interaction must be done through the Linux command line, by opening a console and invoking commands in the shape of **`hadoop fs -<command>`** options
You will be introduced to the essential HDFS commands that you need to interact with the filesystem, but you can check the complete list of commands by invoking **`hadoop fs -help`**
First, let's have a look at what is currently available in the ITL HDFS:

You can browse the contents of a remote HDFS folder through the **hadoop fs –ls** command, which is similar to the Unix listing command. If you specify no parameters you will check your empty home holder.

You can check the root folder of HDFS by running **hadoop fs -ls /**

For our lab we will need to copy the input file into our HDFS folder.

First, we need to create a folder in our HDFS personal space to store the input data. That operation can be solved with the –mkdir option:

```
hadoop fs –mkdir stock
```

If you browse now the contents of your folder you should see the newly created folder.

```
hadoop fs -ls
```

The input for this job is going to be a 500MB file (*NASDAQ.csv*) collecting opening and closing values from the NASDAQ stock exchange from multiple company. You can download this from the provided Google docs link. Now, let's copy the text file to the HDFS. File transfer from a local filesystem to Hadoop is performed using the –copyFromLocal command:

Go in your console shell to the folder where you downloaded the NASDAQ.csv file and run the following command:

```
hadoop fs -copyFromLocal NASDAQ.csv stock
```

If you now check the contents of the input folder in HDFS you should see the file appearing.

```
hadoop fs –ls stock
```

DATASET INFORMATION

Over this lab session we will use a dataset listing the stock daily variations of NASDAQ between 1970 and 2010. The data has been downloaded from Infochimps, a web service collecting numerous datasets.

The data is stored in csv, a tabular data format encoded in plain text. Every line of text represents a single entry, with the following structure

```
exchange, stock_symbol, date, stock_price_open, stock_price_high,
stock_price_low, stock_price_close, stock_volume,stock_price_adj_close.
```

An example row could be something like:

```
NASDAQ,AAPL,1970-10-22, ...
```

We will be using this dataset for the remaining of the lab session.

PART 1: STOCK COUNT IN HADOOP

In this session we are going to replicate the Word Count program covered in the lecture. In a similar spirit, we are going to compute how many times each different stock symbol appears in the dataset. As we get familiar with Java's Hadoop API, we will have to adapt Mapper and Reducer to fit the API.

A Map/Reduce is usually composed of three classes: The **Mapper**, which implements the Map function, the **Reducer**, which implements the Reduce function, and the **main class** that configures the complete job.
The input and output of both the Map and the Reduce function is a tuple of key and value. Hadoop defines several special classes to store the values from the tuple. In this first lab we will use the IntWritable object, which stores an int, and the Text object, which stores a String. The provided source code will show how to initialize and use these objects.

Open the slides from the morning tutorial, and review the word count example. Make sure that you understand the basic flow.
First we run the Mapper: This element first splits the line of text, selecting the information that we need to aggregate for the analysis (In the case of this problem, the symbol name). Then, the Mapper emits a key value pair (with the key being a String with the word, and value being '1').
The Reducer receives as input the aggregated results from the Mapper execution: for each unique Key, it provides a list of values for that key that have been generated by the Mappers. With that information it must be able to count the number of occurrences for the symbol, and generate as result for that word another tuple, with key being a Text containing the String with the word, and value being a IntWritable with the total count stored as an integer.

We will implement that algorithm in Hadoop with the following steps:
First, create a folder for storing your project in your home user folder.
Download the ant build file (*build.xml*) and copy it to the root of your project.
Now it is time to define the Java classes that will implement the complete MapReduce program
Create a new class in the **`src`**/ folder of your project named **`CompanyCountMapper.java`**, and fill it with the following mapper code:

```
import java.io.IOException;


import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class CompanyCountMapper extends Mapper<LongWritable, Text, Text, IntWritable>{
private IntWritable one = new IntWritable(1);

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

```
        String[] fields = line.toString().split(",");

         //Fields contains line as follows.
         //   0         1           2      3    ..........
         //exchange, stock_symbol, date, stock_price_open, stock_price_high,
stock_price_low, stock_price_close, stock_volume,stock_price_adj_close.

         context.write(new Text(fields[1]), one);



    }
}
```

Compare this code to the pseudocode that was displayed in the slides.

- The method has two input arguments that correspond to an entry from the input data. You can ignore the LongWritable (it provides the offset from the start of the document). The Text object contains a String with the value of one line of text from the file (which corresponds to an entry of the dataset as described above).
- IntWritable and Text are the Hadoop equivalents to int and String Java types. They follow their own hierarchy of classes so that they can be moved over the network during the shuffle and partition steps.
- Rather than splitting the text in words, we split them according to commas. Then, we select the symbol field by specifying index 1 for the array.
- The emit command is equivalent to context.write in real MapReduce code.
- Look at the signature of CompanyCountMapper, and try to understand how it relates to pairs <k1,v1>, <k2,v2>

Now, create a new file in the **`src`**/ folder named **`IntSumReducer.java`** and copy the following skeleton of code. It is an incomplete implementation of the reducer.

```
import java.io.IOException;


import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;


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


        private IntWritable result = new IntWritable();


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

```
              throws IOException, InterruptedException {


       int sum = 0;
        for (IntWritable value : values) {
            sum = sum + value.get();
        }
               result.set(sum);


        context.write(key, result);
    }
}
```

Look at the provided structure and understand the differences between Hadoop classes and the pseudocode used in the slides. The code in the complete Mapper class provides a good starting point.

Finally, we need to define a MapReduce program that orchestrates the two classes we have just defined. The following MapReduce Hadoop setup code does this task. Name the file **`CountCompany.java`**

```
import java.util.Arrays;


import org.apache.commons.lang.StringUtils;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;


public class CountCompany {
    public static void runJob(String[] input, String output) throws Exception {
        Configuration conf = new Configuration();
        Job job = new Job(conf);
        job.setJarByClass(CountCompany.class);
        job.setReducerClass(IntSumReducer.class);
        job.setMapperClass(CompanyCountMapper.class);
```

```
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Intritable.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        Path outputPath = new Path(output);
        FileInputFormat.setInputPaths(job, StringUtils.join(input, ","));
        FileOutputFormat.setOutputPath(job, outputPath);
        outputPath.getFileSystem(conf).delete(outputPath, true);
        job.waitForCompletion(true);
        }
    public static void main(String[] args) throws Exception {
        runJob(Arrays.copyOfRange(args, 0, args.length - 1), args[args.length - 1]);
    }
}
```

The main method configures the Hadoop job through a JobConf object. Method calls explicitly specify the mapper and reducer, the path to the input files, the path where results will be written to, and the types of the output keys for the reducer.

The code of your mapreduce project is complete. In order to create a package ready to be executed by Hadoop go to the project root folder and invoke the ant command

`ant clean dist`

If the code compiles correctly a compressed jar file named HadoopTest.jar will appear in a newly created dist/ folder

Execute the job from the base folder of your project running the following line:

`hadoop jar dist/CompanyCount.jar CountCompany stock out`

If the job has executed correctly there should be two files in the out folder. Only the part-XXXX files contain the output of Reducers. _SUCCESS is empty.

`part-r-00000  _SUCCESS`

These files are stored in the 'distributed' filesystem, so you need to download them to your local filesystem before you can read them. You can copy the results file back to your local filesystem by using the fs –copyToLocal option. Remember that Reducer#1 output is provided in a file called part-00000

`hadoop fs –copyToLocal <hdfsfile> <localdestination>`

After retrieving the output you can remove the output folder through the –rmr fs command. If the designated output folder was out, you can remove it by doing:

`hadoop fs –rmr out`

Once you have completed the base Count exercise, answer correctly the following questions:

Q1 *How many entries does the dataset have for the record AAPL?*
Q2 *The Hadoop job outputs a set of statistics after the completion. What was the number of key-value pairs generated by the Mapper, the number of unique inputs to the Reducer, and the final number of records emitted by the Reducer? Based on these values answer the following two questions: How many records has the document in total? How many unique companies appear in the NASDAQ dataset?*

PART 2: WRITING MORE COMPLEX MAPREDUCE PROGRAMS

The objective for this second part is to write your own Map/Reduce program, applying the aggregate summarisation pattern described in the morning lecture.
The result must be a list of key/value pairs, showing for each company, the maximum close price over the whole history of the dataset.

Create a new folder for your project.
Copy the ant file from the previous one to the root, and create a src folder.
Similarly to the previous one, you need to create three files: Mapper, Reducer and Main class.
- Mapper: Selects the stock symbol (index 1), and the close price for that row (index 6). Emits first as key (Text), the second as value (DoubleWritable, will need conversion from String to double).
- Reducer: Initiates minimum with a high value (eg Double.MAX_DOUBLE), iterates over all the values for a company, updating the minimum estimation if the current one is smaller. Finally emits for the company, the computed minimum.
- Main class: You will need to update the names of the program parts if you change them.
Additionally, be careful with the MapOutputKey/Value, ReduceOutputKey/Value classes, as they are possibly different from the previous exercise.

Compile again your jar (ant clean dist), and run the job, specifying a different output folder in order to not overwrite the results.
Q3: *By Looking at the results, what is the minimum closing price for IBM?*

PART 3: (OPTIONAL) ADDITIONAL MAP/REDUCE ANALYSIS ON THE DATASET

The goal of this job is to generate a custom report where that shows several statistics for each company each year they have been in the stock market. Results must be sorted first by company, then by year. The information collected in each yearly report will include three values: the minimum closing value, maximum closing value, and total volume of operations over the year.

ACAD,1996,20.0,29.7,278798798798

ACAD,1997,25.6,34.2,2342039482094