MapReduce Laboratory

In this laboratory students will learn how to use the Hadoop client API by working on a series of exercises:

- The classic Word Count and variations on the theme
- Design Pattern: Pair and Stripes
- Design Pattern: Order Inversion
- Join implementations

Note that the design patterns outlined above have been originally discussed in:

- Jimmy Lin, Chris Dyer, "Data-Intensive Text Processing with MapReduce," Morgan Claypool

1 Run the Virtual Machine

1.1 VM details

- It is preconfigured with all the software needed to get through the assignments.
- It has an Ubuntu 12.04 LTE (64bit) with XFCE Desktop Environment
- It requires $\geq$ 2GB of memory
- It has a single account:
    - username: student
    - password: password
    - Notice that login is password less and root commands can be run using sudo which is password less as well.

IMPORTANT: Shut down the VM once finished the session (do not close abruptly the window)

1.2 Run the VM on the laptop

Download the image tarball from

http://tinyurl.com/mr-lab-bigfoot

(or ask to the teacher the USB key) and uncompress it
The image it can be used either VMplayer or Virtualbox

VMplayer:

- Windows/Linux: download the freeware player from here
- Mac OS: unfortunately there is no freeware player, but it can be used VMware Fusion which has a 30-day trial.

Virtualbox:

- The image can be loaded in Virtualbox with no issues. Be sure to select the file HadoopVM.vmdk as a virtual hard-drive (and not one of the HadoopVM-sXXX.vmdk files).
- To install guest additions, once the image is running:
    - sudo apt-get install dkms build-essential
    - download the additions (devices → install guest additions)
    - sudo su -
    - cd /media/VIRTUALBOXADDITION_version
    - ./VBoxLinuxAddition.run

1.3 Run the VM on the laboratory machines

The VM image is stored in a local directory. This means that:

- The student needs to use every time the same physical machine; any modification will affect the VM on that specific physical machine;
- The physical machine and the local directory can be accessed by anyone;

  - To protect the access to the VM, once the VM is up and running, the student may add a login screen (remember to change the password of the user “student”).
  - This may not prevent other people from deleting the whole VM from the machine, therefore remember to save the work somewhere outside the VM (e.g., backup of the modified Java files using a shared directory, which can be inside the student user space).

The VM is accessible from the user menu. It can be found in **Menu → Istruzione → Virtual Machine Hadoop**

IMPORTANT: Shut down the VM before leaving (do not close abruptly the window)

2 Setup the laboratory sources in Eclipse

First, open the browser and go to the course web page, where you can find the file with the source code that will be used in this lab session. Download the source code and uncompress it in any of your local directory on the VM.

Then, create a new Java Project in Eclipse:
- Inside Eclipse select the menu item **File > New > Project ...** to open the **New Project** wizard.
- Select **Java Project** then click **Next** to start the **New Java Project**
- Type a name for your new project, such as "mr-lab"
- Ensure to use *JavaSE-1.6* as JRE, then click on **Next**
- Go on **Libraries** tab, and click on **Add External Jars**
- Add the following jars:
  - /usr/lib/hadoop/hadoop-common.jar
  - /usr/lib/hadoop-0.20-mapreduce/hadoop-core.jar
  - /usr/lib/hadoop-hdfs/hadoop-hdfs.jar
  - Click on **Finish**

Now you have a new project.

The next step is to import the source files to complete in the Project.
- Inside Eclipse, select the **src** directory in your project.
- Right-click on the src directory and select **Import**
- Select **General / File System** then click **Next**
- In **From Directory**, select *./mr/src/*, which is inside the directory where you cloned the git repository
- In the tree on the left, expand *src* and select (put the tick on) *fr*, then click **Finish**.

At this point you should have a java project named mapred-lab already configured. The next step is starting with the first exercise. Note that there can be errors in the source code you imported. This is normal, since there are uncompleted source files that will be corrected in the exercises.

3 Utilities

3.1 How to launch a job

In order to launch a job, you need first to export a **JAR** file. Therefore, from Eclipse:
- Select the menu item **File > Export**
- Type **JAR**, select **JAR file** and click on **Next**
- In the textbox **Select the export destination**, type the name of the new **JAR** you want to export, i.e. '*/home/student/mrlab.jar*'
- Type **Finish**.

Once you have exported your jar file, you can run your code using the local version of Hadoop. Open a *terminal*, and type:

```
hadoop jar <jarname.jar> <fully.qualified.class.Name> <Parameters>
```

For example, for running the *WordCount* exercise, type:

```
hadoop jar mrlab.jar fr.eurecom.dsg.mapreduce.WordCount 2 INPUT/text/quote.txt
OUTPUT/wordcount/
```

Note that you need to specify a *non existing* output directory, or to delete it before running the job.

3.2 Navigate HDFS

The HDFS client can talk to the namenode to obtain information on the distributed file system, and to make a number of operations. The basic command is

- `hadoop fs [option]`
- where option can be
  - `–ls –R` → Displays recursively the content of the file system
  - `-get <src> <localdst>` → Copies files from hdfs to the local file system
  - `–cat <file>` → Displays <file> on the terminal (copies to stdout)

  - and many others… see the Hadoop FS shell commands official documentation.

Alternatively, one can use the web interface to interact with the namenode (see next section).

3.3 Web interfaces: Monitor job progress

Hadoop publishes some web interfaces that display JobTracker and HDFS statuses. You can access them using the following links:

- Jobtracker Web Interface: http://127.0.0.1:50030/
- NameNode Web Interface: http://127.0.0.1:50070/

4 Exercises

Note, exercises are organized in ascending order of difficulty.

4.1 Exercise 1 – Word Count

Count the occurrences of each word in a text file. This is the simplest example of MapReduce job: in the following we illustrate three possible implementations.

- **Basic**: the map method emits 1 for each word. The reduce aggregates the ones it receives from mappers for each key it is responsible for and save on disk (HDFS) the result
- **In-Memory Combiner**: instead of emitting 1 for each encountered word, the map method (partially) aggregates the ones and emit the result for each word
- **Combiner**: the same as In-Memory Combiner but using the MapReduce Combiner class

Instructions

For the **basic** version the student has to modify the file *WordCount.java* in the package *fr.eurecom.dsg.mapreduce*. The user must operate on each TODO filling the gaps with code, following the description associated to each TODO. The package *java.util* contains a class *StringTokenizer* that can be used to tokenize the text.

For the **In-Memory** Combiner and the **Combiner** the student has to modify *WordCountIMC.java* and *WordCountCombiner.java* in the same package referenced above. The student has to operate on each TODO using the same code of the basic word count example except for the TODOs marked with a star *. Those must be completed with using the appropriate design pattern.

When an exercise is completed you can export it into a jar file that will then be used to execute it.

Example of usage

The final version should get in input three arguments: the number of reducers, the input file and the output path. Example of execution are:

```
hadoop jar <compiled_jar> fr.eurecom.dsg.mapreduce.WordCount 3 <input_file>
<output_path> hadoop jar <compiled_jar> fr.eurecom.dsg.mapreduce.WordCountIMC 3
<input_file> <output_path> hadoop jar <compiled_jar>
fr.eurecom.dsg.mapreduce.WordCountCombiner 3 <input_file> <output_path>
```

To test your code use the file `/user/student/INPUT/text/quote.txt`, saved in the local HDFS fs.

To run the final version of your job, you can use a bigger file, `/user/student/INPUT/text/gutenberg-partial.txt`, which contains an extract of the English books from Project Gutenberg (http://www.gutenberg.org/), which provides a collection of full texts of public domain books.

4.2 Exercise 2 – Term co-occurrences

In the following exercise, we need to build the term co-occurrence matrix for a text collection. A co-occurrence matrix is a n x n matrix, where n is the number of unique words in the text. For each couple of words, we count the number of times they co-occurred in the text in the same line.

4.2.1 Pairs Design Pattern

The basic (and maybe most intuitive) implementation of this exercise is the *Pair*. The basic idea is to emit, for each couple of words in the same line, the couple itself (or *pair*) and the value 1. For example, in the line `w1 w2 w3 w1`, we emit `(w1,w2):1, (w1,w3):1, (w2,w1):1, (w2,w3):1, (w2,w1):1, (w3,w1):1, (w3,w2):1, (w3,w1):1`.

In this exercise, we need to use a composite key to emit an occurrence of a pair of words. The student will understand how to create a custom Hadoop data type to be used as key type.

A Pair is a tuple composed by two elements that can be used to ship two objects within a parent object. For this exercise the student has to implement a TextPair, that is a Pair that contains two words.

Instructions

There are two files for this exercise:

- *TextPair.java*: data structure to be implemented by the student. Besides the implementation of the data structure itself, the student has to implement the serialization Hadoop API (write and read Fields).
- *Pair.java*: the implementation of a pair example using *TextPair.java* as datatype.

Example of usage

The final version should get in input three arguments: the number of reducers, the input file and the output path. Example of execution are:

```
hadoop jar <compiled_jar> fr.eurecom.dsg.mapreduce.Pair 1 <input_file>
<output_path>
```

To test your code use the file `/user/student/INPUT/text/quote.txt`, saved in the local HDFS fs.

To run the final version of your job, you can use a bigger file, `/user/student/INPUT/text/gutenberg-partial.txt`.

4.2.2 Stripes Design Pattern

This approach is similar to the previous one: for each line, co-occurring pairs are generated. However, now, instead of emitting every pair as soon as it is generated, intermediate results are stored in an associative array. We use an associative array, and, for each word, we emit the word itself as key and a *Stripe*, that is the map of co-occurring words with the number of associated occurrence.
For example, in the line `w1 w2 w3 w1`, we emit:

```
w1:{w2:1, w3:1}, w2:{w1:2,w3:1}, w3:{w1:2, w2:1}, w1:{w2:1, w3:1}
```

Note that, instead, we could emit also:

```
w1:{w2:2, w3:2}, w2:{w1:2,w3:1}, w3:{w1:2, w2:1}
```

In this exercise the student will understand how to create a custom Hadoop data type to be used as value type.

Instructions

There are two files for this exercise:

- *StringToIntMapWritable.java*: the data structure file, to be implemented
- *Stripes.java*: the MapReduce job, that the student must implement using the StringToIntMapWritable data structure

Example of usage

```
hadoop jar <compiled_jar> fr.eurecom.dsg.mapreduce.Stripes 2 <input_file>
<output_path>
```

To test your code use the `file /user/student/INPUT/text/quote.txt`, saved in the local HDFS fs.

To run the final version of your job, you can use a bigger file, `/user/student/INPUT/text/gutenberg-partial.txt`.

4.3 Exercise 3 – Relative term co-occurrence and Order Inversion Design Pattern

In this example we need to compute the co-occurrence matrix, like the one in the previous exercise, but using the relative frequencies of each pair, instead of the absolute value. Pratically, we need to count the number of times each pair (w_i, w_j) occurs divided by the number of total pairs with w_i (marginal).

The student has to implement the `Map` and `Reduce` methods and the special partitioner (see `OrderInversion#PartitionerTextPair` class), which apply the partitioner only according to the first element in the Pair, sending all data regarding the same word to the same reducer. Note that inside the `OrderInversion` class there is a field called `ASTERISK` which should be used to output the total number of occourrences of a word.

Instructions

There is one file for this exercise called `OrderInversion.java`. The `run` method of the job is already implemented, the student should complete the mapper, the reducer and the partitioner, as explained in the TODOs.

Example of usage

```
hadoop jar <compiled_jar> fr.eurecom.fr.mapreduce.OrderInversion 4 <input_file>
<output_path>
```

To test your code use the file `/user/student/INPUT/text/quote.txt`, saved in the local HDFS fs.

To run the final version of your job, you can use a bigger file, `/user/student/INPUT/text/gutenberg-partial.txt`.

4.4 Exercise 4 – Joins

In MapReduce the term join refers to merging two different dataset stored as unstructured files in HDFS. As for databases, in MapReduce there are many different kind of joins, each with its use-cases and constraints. In this laboratory the student will implement the **Reduce-Side Join**: the map phase tags each record such that records of different inputs that have to be joined will have the same tag. Each reducer will receive a tag with a list of records and perform the join.

Jobs

- **Reduce Side Join**: You need to find the two-hops friends, i.e. the friends of friends of each user, in the twitter dataset. In particular, you need to implement a self-join, that is a join between two instances of the same dataset, on the twitter graph. To test your code use the file `/user/student/INPUT/twitter/twitter-small.txt`, saved in the local HDFS fs. To run the final version of your job, you can use a bigger file, `/user/student/INPUT/twitter/twitter-big-sample.txt`. Both files contain lines in the form `userid friendid`.

Instructions

- **Reduce Side Join**: use the file *ReduceSideJoin.java* as starting point. This exercise is different from the others because it does not contain any information on how to do it. The student is free to choose how to implement it.

Example of usage

```
hadoop jar <compiled_jar> fr.eurecom.fr.mapreduce.ReduceSideJoin 1 <input_file2>
<input_file1> <output_path>
```

4.5 Exercise 5 – PageRank

In this example, we compute the simple version of the PageRank, i.e., we do not consider the jump factor and the presence of sink nodes (every node has at least one output link, and the graph is completely connected).
The graph is described through its adjacency list: in particular, the starting input file has the following format for each line:

```
nodeID   currentPageRank   neighNodeID1   neighNodeID2   neighNodeID3   ...
```

where "`nodeID`" is the unique identifier of the node, "`currentPageRank`" is the current value of the PageRank (initially set to the inverse of the number of nodes), and then there are the identifiers of the output links (the graph is directed, i.e., the adjacency matrix is not symmetric). The files of two different graphs can be downloaded from the course webpage (and should be stored in HDFS).

The student has to implement the `Map` and `Reduce` methods for the simplified PageRank computation. The `run` method of the job should launch a finite number of iterations (given as input parameter). At each iteration, the intermediate directory should be removed, so that, at the end, there will be only the initial directory and the final one. Note that, since the output of a MapReduce computation is a directory, and the PageRank computation is iterative, then the input (even at the beginning) should be always a directory. The student, therefore, should put the files describing a graph in separated directories.