UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
School of Professional & Executive Development

Big Data Management and Analytics
Session: MapReduce

Lecturers: Petar Jovanovic and Sergi Nadal

November 11th, 2016

1 Required Tools

- SSH and SCP client
- `eclipse` IDE with JDK 7 and Maven plugin installed

2 Tasks To Do Before The Session

Before coming to this MapReduce session, you should have checked the session slides and write down all doubts you might have about them.

For this session you will be given some MapReduce examples and then you will be asked to implement some other MapReduce jobs based on such examples. More precisely, you will be given the *Projection*, *Grouping(Sum)* and *Cartesian product* relational algebra operations and you will be asked to implement the *Selection*, *Grouping(Avg)* and *Join*. In order for you to avoid wasting too much time on understanding the examples, we strongly recommend you to take a look at the code before coming to class. Some questions you can try to answer to check whether you understand what is going on are:

- What is the meaning of the two following Mapper and Reducer class' headers?

```
[...] extends Mapper<LongWritable, Text, Text, DoubleWritable>
[...] extends Reducer<Text, DoubleWritable, Text, DoubleWritable>
```

- Why is the *Projection* job not including a Reducer task? Would you say there is still a MergeSort?
- How can we send arguments to the MapReduce job? For instance, how do the Mappers know the group by and aggregation attributes?
- Are you able to sketch how the *Cartesian product* works through the different MapReduce phases?

3 Part A: Examples & Questions (30min)

The first 30 minutes of the session will be spent on answering all the possible questions you might have come up with during the week by preparing the tasks before the session (see above).

4 Part B: In-class Practice (2h 30min)

4.1 Exercise 1 (30min): Setting up the cluster

Follow the document enclosed to this exercise sheet to start up a YARN/MapReduce cluster, compile the new Java code as you did in previous sessions, make a JAR out of it and upload it to the master node. **Let the lecturer know when this is finished.**

4.2 Exercise 2 (2h): Implement your own MapReduce jobs

Generate 1000 rows of data in a sequence file and load it into HDFS by means of the following commands (*labo3.jar* is the JAR file for this session). Note that we are loading the file by using a block size of 128k (this file configuration is set in the Java code), which is really small (1000 rows makes the file very small as well). This is so to avoid very long execution times (specially in the case of the cartesian product).

```
hadoop-2.5.1/bin/hadoop jar labo3.jar mapreduce
       -hdfsSequence -instances 1000 wines.1000.128k
```

Then you can run the examples. It can take a bit until the job is started so please be patient:

```
hadoop-2.5.1/bin/hadoop jar labo3.jar mapreduce
       -projection wines.1000.128k projection.1000.128k
hadoop-2.5.1/bin/hadoop jar labo3.jar mapreduce
       -groupByAndAgg wines.1000.128k groupByAndAgg.1000.128k
hadoop-2.5.1/bin/hadoop jar labo3.jar mapreduce
       -cartesian wines.1000.128k cartesian.1000.128k
```

This will leave the MapReduce output in the folders *projection.1000.128k*, *groupByAndAgg.1000.128k* and *cartesian.1000.128k*, respectively. Unfortunately, the content in these folders is one file for Reducer run. In order to bring a single merged file to the local system and to be able to explore it more comfortably, run the following commands:

```
hadoop-2.5.1/bin/hdfs dfs -getmerge projection.1000.128k projection.1000.128k
hadoop-2.5.1/bin/hdfs dfs -getmerge groupByAndAgg.1000.128k groupByAndAgg.1000.128k
hadoop-2.5.1/bin/hdfs dfs -getmerge cartesian.1000.128k cartesian.1000.128k
```

Now do the following items:

1. Look at the console output of any of the examples and search for the values of *Launched map tasks* and *Map input records*. What are these values? How many are they? Do these numbers make sense to you? Justify your answer.

2. Implement a MapReduce job that performs a selection on the data set. For instance, select only tuples in which the attribute type equals *type_1*.

   ```
   hadoop-2.5.1/bin/hadoop jar labo3.jar mapreduce
          -selection wines.1000.128k selection.1000.128k
   ```

3. As seen, the *GroupByAndAgg* MapReduce job groups rows by the attribute *type* and performs the sum of the attribute *col* (i.e., wine color intensity). Modify it such that, instead of the sum, the aggregation function is the average.

4. Implement a MapReduce job that performs a join operation based on the *region* between the subsets of rows with *type type_1* and *type_2* (*type_3* rows are simply ignored like in the Cartesian product job).

   ```
   hadoop-2.5.1/bin/hadoop jar labo3.jar mapreduce
          -join wines.1000.128k join.1000.128k
   ```

Deliverable: Upload a single zip file to the campus containing a text file with your answer to questions 1 and the Java classes you have modified for the rest of exercises. Name such zip file as "name_surname.zip"

5 Part C: Optional Practice (3h)

In this section we are going to use MapReduce to perform some predictions in a given data set. More precisely, we are going to classify some breast tumors as malign or benign. The algorithm we will use is one nearest neighbor (1NN from now on) which is a very simple one. Later on in the course, you will be taught much more sophisticated techniques to perform classification (and not all of them will be easy to distribute in MapReduce jobs).

As many predictive algorithms, 1NN departs from having some (relatively large) data set that contains rows that are used as baseline for the predictions and other different rows to predict the class (in this context the class is "malign" or "benign"). More specifically, the former is typically called the **train set** and the latter is the **test set**. The idea behind 1NN then is to compute, for every row in the test set, its distance (Euclidean; see formula 1, where x and y are two different rows and d is the number of variables) with respect to all rows from the train set. The final predicted class corresponds to the closest row class from the train set.

$$dist(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + ... + (x_d - y_d)^2} \tag{1}$$

The data set you are given is a CSV file (*breast.csv*) with multiple columns. You can ignore the first column since it simply is an identifier but you should pay attention on the rest. Check *breast.names* to see more information about this data set:

- Variable *diagnosis* represents whether the tumor is malign ("M") or benign ("B").
- Variable *train* says if such row belongs to the train (if value equals 1) or test set (otherwise).
- The rest of variables must be used to compute distances.

According to this, in this exercise you are asked to complete the Java classes *OneNNMapReduce_1* and *OneNNMapReduce_2* in order to implement a MapReduce version of 1NN. The first one will implement the 1NN algorithm itself and will perform the predictions. The second one will take as input the predictions made by the previous one and will compute the accuracy of such predictions. You can follow the next guidelines to make it easier.

1. In *OneNNMapReduce_1*:

   (a) As previously explained, 1NN computes the distance between all the rows in the train and the test sets. Thus, this MapReduce Mapper should start performing a cartesian product (use the *Cartesian product* job you were given as an example) between such both sets. Be aware of ignoring the first row of the CSV file (the headers).

   (b) Now in the Reducer you first need to separate the rows a Reducer has received from the Mappers in two different arrays (still you can use the *Cartesian product* job as example). Once this is done, obtain the closest train row for every test row and assign the latter the same class as the former.

   (c) Finally output the predicted class and the real class. For instance, output a string like "B_B" to mean that 1NN predicted benign and the real class is also benign, or "M_B" to say that 1NN predicted malign but the real class is benign. Note the rows from the test set still have the *diagnosis* variable so we can use it for comparison.

2. *OneNNMapReduce_2* should essentially be a word count implementation (see slides for examples).

3. Additionally, some other notes:

   - Note the input file now is a CSV file (not a sequence file) and therefore it is written in HDFS as a plain file. Consequently, when you process it in *OneNNMapReduce_1*, you should forget about the input key and simply attack to the input value which is the parameter that will contain the whole row itself.
   - Since the input is a CSV file, you will need to split each line by commas. Then you can use the following functions:
     - *Utils_1NN.getAttribute(String[] row, String attribute)* is analogous to *Utils.getAttribute(String[] row, String attribute)* function we used in section 4.
     - *Utils_1NN.getPredictors()* returns the list of variable that need to be use to compute the distances.

Steps to execute the job:

1. Upload the file *breast.csv* to the master and load the CSV file (forget about the number of blocks for this exercise; we are simply focusing on the MapReduce implementation):

   ```
   hadoop-2.5.1/bin/hdfs dfs -put breast.csv
   ```

2. Run the MapReduce jobs. The output file of *OneNNMapReduce_1* should be the same as input file of *OneNNMapReduce_2*. Remember to remove the outputted files before running the job if you want to reuse them.

   ```
   hadoop-2.5.1/bin/hadoop dfs -rm -r breast.1nn_1 breast.1nn_2

   hadoop-2.5.1/bin/hadoop jar labo3.jar mapreduce
          -1nn_1 breast.csv breast.1nn_1
   hadoop-2.5.1/bin/hadoop jar labo3.jar mapreduce
          -1nn_2 breast.1nn_1 breast.1nn_2
   ```

3. Merge and see the final outcome of the predictions:

   ```
   hadoop-2.5.1/bin/hadoop dfs -getmerge breast.1nn_2 breast.txt
   cat breast.txt
   ```

Deliverable: Upload a single zip file to the campus containing the OneNNMapReduce_1 and OneNNMapReduce_2 Java classes, and a text file with the final output of OneNNMapReduce_2. Name such zip file as "name_surname.zip"