Authors: Carayon Chloé - Taillieu Victor **Date**: 11/11/2020

TP4: MapReduce Java

Introduction

In this TP, we have to create our own MapReduce programs in Java.

Firstly, we have to import the jar in the cluster. We have to do it again each time we create new MapReduce programs. By building Maven, we get jar and import it in our cluster.

```
$ scp hadoop-examples-mapreduce-1.0-SNAPSHOT-jar-with-dependencies.jar ccarayon@hadoop-edge01.efrei.online:/home/ccarayon/
Welcome to EFREI Hadoop Cluster

Password:
hadoop-examples-mapreduce-1.0-SNAPSHOT-jar-wi 100%   51MB   2.3MB/s   00:22
```

By running *wordcount*, we obtain:

```
-sh-4.2$ yarn jar  hadoop-examples-mapreduce-1.0-SNAPSHOT-jar-with-dependencies.jar
An example program must be given as the first argument.
Valid program names are:
  wordcount: A map/reduce program that counts the words in the input files.
```

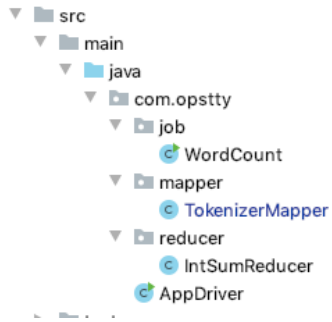
Now let's test the wordcount MapReduce program on our csv file trees.csv:

```
-sh-4.2$ yarn jar  hadoop-examples-mapreduce-1.0-SNAPSHOT-jar-with-dependencies.jar \wordcount trees.csv treeoutput
```

We obtain this result:

```
-sh-4.2$  hdfs dfs -cat treeoutput/part-r-00000 | head -n 8
(48.8183933679, 1
(48.8201249835, 1
(48.8204495642, 1
(48.8210086122, 1
(48.8213214388, 1
(48.8215800145, 1
(48.8220238534, 1
(48.8224956954, 1
```

Our mission is to create several MapReduce programs as the wordcount example. By looking at the intial project:



We understand that we have jobs that use mappers and reducers. In order to create new jobs, we have to create new mappers and reducers, use them in the corresponding job and include the jobs in the AppDriver to be able to run them. To test them, we implemented *UnitTests*.

1.8.1 Districts containing trees

1.8.1.1 Implementation

We create *ListDistrics* to display the list of districts containing trees. It uses a new mapper and a new reducer. By looking at the csv file, we see that the districts are in the second column. For each line, the mapper *ListDistrictsMapper* only returns one information: the district as key (and null for the value).

For the reducer and combiner *ListReducer*, we just combine the keys from the mapper. In *ListDistrics*, we specify that we use *ListDistrictsMapper* as mapper and *ListReducer* as combiner and reducer. Finally, we have to call our *ListDistrics* in the AppDriver in order to be able to run it.

1.8.1.2 Commands and result

We can see all MapReduce programs available:

```
--sh-4.2$ yarn jar  hadoop-examples-mapreduce-1.0-SNAPSHOT-jar-with-dependencies.jar
An example program must be given as the first argument.
Valid program names are:
  listdistricts: A map/reduce program that displays the list of districts in the input files.
  wordcount: A map/reduce program that counts the words in the input files.
```

Then we execute listdistricts:

```
-sh-4.2$ yarn jar  hadoop-examples-mapreduce-1.0-SNAPSHOT-jar-with-dependencies.jar listdistricts trees.csv listdistrictout
```

We obtain the following result, it lists all districts containing trees:

```
-sh-4.2$ hdfs dfs -cat listdistrictout/part-r-00000
11
12
13
14
15
16
17
18
19
20
3
4
5
6
7
8
9
```

1.8.2 Show all existing kinds

1.8.2.1 Implementation

We create *ListKinds* to display the list of different trees kinds. It uses a new mapper and the previous reducer.

By looking at the csv file, we see that the kinds are in the third column. For each line, the mapper *ListKindsMapper* only returns one information: the kinds as key (and null for the value) as inputs for the reducer. For the reducer and combiner we reuse *ListReducer*.

In *ListKinds*, we specify that we use *ListKindsMapper* as mapper and *ListReducer* as combiner and reducer. And we call our *ListKinds* in the AppDriver in order to be able to run it.

1.8.2.2 Commands and result

Now let's take a look at the output. We run:

```
-sh-4.2$ yarn jar  hadoop-examples-mapreduce-1.0-SNAPSHOT-jar-with-dependencies.jar listkinds trees.csv listkindsout
```

And we obtain:

```
-sh-4.2$ hdfs dfs -cat listkindsout/part-r-00000
Acer
Aesculus
Ailanthus
Alnus
Araucaria
Broussonetia
Calocedrus
Catalpa
Cedrus
Celtis
Corylus
Davidia
Diospyros
Eucommia
Fagus
Fraxinus
Ginkgo
Gymnocladus
Juglans
Liriodendron
Maclura
Magnolia
Paulownia
Pinus
Platanus
Pterocarya
Quercus
Robinia
Sequoia
Sequoiadendron
Styphnolobium
Taxodium
Taxus
Tilia
Ulmus
Zelkova
```

1.8.3 Number of trees by kind

1.8.3.1 Implementation

For this job, we need to calculate the number of trees for each kind. We can reuse the previous question which displays the list of kinds.

Firstly, we create *CountKinds* which displays the number of trees by kind. For the mapper, we reuse *ListKindsMapper*, which returns the kinds as input keys for the reducer. For the reducer and combiner we reused *IntSumReducer* which was given for the wordcount, it combines the keys and counts their occurence.

In *CountKinds*, we specify that we use *ListKindsMapper* as mapper and *IntSumReducer* as combiner and reducer. And we call our *CountKinds* in the AppDriver in order to be able to run it.

1.8.3.2 Commands and result

Now let's take a look at the output. We run:

```
-sh-4.2$ yarn jar  hadoop-examples-mapreduce-1.0-SNAPSHOT-jar-with-dependencies.jar countkinds trees.csv countkindsout
```

We obtain:

```
-sh-4.2$ hdfs dfs -cat countkindsout/part-r-00000
Acer    3
Aesculus        3
Ailanthus       1
Alnus   1
Araucaria       1
Broussonetia    1
Calocedrus      1
Catalpa 1
Cedrus  4
Celtis  1
Corylus 3
Davidia 1
Diospyros       4
Eucommia        1
Fagus   8
Fraxinus        1
Ginkgo  5
Gymnocladus     1
Juglans 1
Liriodendron    2
Maclura 1
Magnolia        1
Paulownia       1
Pinus   5
Platanus        19
Pterocarya      3
Quercus 4
Robinia 1
Sequoia 1
Sequoiadendron  5
Styphnolobium   1
Taxodium        3
Taxus   2
Tilia   1
Ulmus   1
Zelkova 4
```

1.8.4 Maximum height per kind of tree

1.8.4.1 Implementation

We create *TallestKinds* which calculates the height of the tallest tree of each kind. It implements a new mapper and a new reducer.

For the mapper *TallestKindsMapper*, we focus on two information: we need the kind as a key and the height as a value for our key value couple.

For the reducer and combiner *MaxReducer*, we focus on the maximum height of each kind. For each key, we compare each height with the current maximum and change it if it is higher. We return the key with the corresponding maximum value of height.

In *TallestKinds*, we specify that we use *TallestKindsMapper* as mapper and *MaxReducer* as combiner and reducer. Finally, we call our *TallestKinds* in the AppDriver in order to be able to run it.

1.8.4.2 Commands and result

Now let's take a look at the output. We run:

```
-sh-4.2$ yarn jar  hadoop-examples-mapreduce-1.0-SNAPSHOT-jar-with-dependencies.jar tallestkinds trees.csv tallestkindsout
```

We obtain:

```
-sh-4.2$ hdfs dfs -cat tallestkindsout/part-r-00000
Acer    16.0
Aesculus        30.0
```

```
Ailanthus       35.0
Alnus   16.0
Araucaria       9.0
Broussonetia    12.0
Calocedrus      20.0
Catalpa 15.0
Cedrus  30.0
Celtis  16.0
Corylus 20.0
Davidia 12.0
Diospyros       14.0
Eucommia        12.0
Fagus   30.0
Fraxinus        30.0
Ginkgo  33.0
Gymnocladus     10.0
Juglans 28.0
Liriodendron    35.0
Maclura 13.0
Magnolia        12.0
Paulownia       20.0
Pinus   30.0
Platanus        45.0
Pterocarya      30.0
Quercus 31.0
Robinia 11.0
Sequoia 30.0
Sequoiadendron  35.0
Styphnolobium   10.0
Taxodium        35.0
Taxus   13.0
Tilia   20.0
Ulmus   15.0
Zelkova 30.0
```

1.8.5 Sort the trees height from smallest to largest

1.8.5.1 Implementation

We create *SortHeight* to sort the trees height from the smallest to the largest. It implements a new mapper and a new reducer.

For the mapper *SortHeightMapper*, we focus on two information: we take the height (as a float) of the tree as a key and the id (an int) as value. For the reducer and combiner called *SortReducer*, our keys are of type float so it orders the trees by itself. We add a for loop to write all the key value couple. For this part, we had to specify in those class that we use FloatWritable and not Text as we have input and output key which are float.

In *SortHeight*, we specify that we use *SortHeightMapper* as mapper and *SortReducer* as combiner and reducer. Finally, we call our *SortHeight* in the AppDriver in order to be able to run it.

1.8.5.2 Commands and result

Now let's take a look at the output. We run:

```
-sh-4.2$ yarn jar  hadoop-examples-mapreduce-1.0-SNAPSHOT-jar-with-dependencies.jar sortheight trees.csv sortheightout
```

And we obtain:

```
-sh-4.2$ hdfs dfs -cat sortheightout/part-r-00000

2.0     3
5.0     89
6.0     62
9.0     39
10.0    44
10.0    32
10.0    95
10.0    61
```

```
10.0    63
11.0    4
12.0    93
12.0    66
12.0    50
12.0    7
12.0    48
12.0    58
12.0    33
12.0    71
13.0    36
13.0    6
14.0    68
14.0    96
14.0    94
15.0    91
15.0    5
15.0    70
15.0    2
15.0    98
16.0    28
16.0    78
16.0    75
16.0    16
18.0    64
18.0    83
18.0    23
18.0    60
20.0    34
20.0    51
20.0    43
20.0    15
20.0    11
20.0    1
20.0    8
20.0    20
20.0    35
20.0    12
20.0    87
20.0    13
22.0    10
22.0    47
22.0    86
22.0    14
22.0    88
23.0    18
25.0    24
25.0    31
25.0    92
25.0    49
25.0    97
25.0    84
26.0    73
27.0    65
27.0    42
28.0    85
30.0    76
30.0    19
30.0    72
30.0    54
30.0    29
30.0    27
30.0    25
30.0    41
30.0    77
30.0    55
30.0    69
30.0    38
30.0    59
30.0    52
30.0    37
30.0    22
30.0    30
31.0    80
31.0    9
32.0    82
```

```
33.0    46
34.0    45
35.0    56
35.0    81
35.0    17
35.0    53
35.0    57
40.0    26
40.0    74
40.0    40
42.0    90
45.0    21
```

1.8.6 District containing the oldest tree

1.8.6.1 Implementation

This time, we have to create our own Writable subclass *DoubleIntWritable*, we create two IntWritable attributs val1 for district and val2 for year, their methods and the specific methods of Writable classes : readfields and write. The purpose of this custom Writable class is to be able to have 2 attributs for the value in the key value couple in order to do comparison in the reducer.

Then we create *SortHeight* to display the district of the oldest tree. It implements a new mapper and a new reducer.

For the mapper *DistrictOldest*, we have to be carefull by taking as value output: DoubleIntWritable. If we are not in the header, we set val1 to district (2nd column) and val2 to year (6th column) which are the two attributs of our DoubleIntWritable object. Our outputs have the form: IntWritable (of value 1) for the key and the DoubleIntWritable object as the value. By doing this, all our values are store with only one key and we are able to compare them in the reducer and return only one element.

For the reducer *OldestReducer*, we compare the year thanks to a for loop going through the values, and return the minimum one. We also have to specify that we have IntWritable and DoubleIntWritable as input and IntWritable and NullWritable as output. This difference between the input and output class bring us some issues: As the output from the Mapper was different from the input of the combiner and the reducer, we had to supress the combiner from DistrictOldest job and we had to specify the MapOutputKey class and MapOutputValue class. In *DistrictOldest*, we specify that we use *SortHeightMapper* as mapper and *SortReducer* as reducer.

1.8.6.2 Commands and result

Now let's take a look at the output. We run:

```
-sh-4.2$ yarn jar  hadoop-examples-mapreduce-1.0-SNAPSHOT-jar-with-dependencies.jar  districtoldest trees.csv oldestout
```

We obtain:

```
-sh-4.2$ hdfs dfs -cat oldestout/part-r-00000
5
```

1.8.7 District containing the most trees

1.8.7.1 Implementation

For this last exercise, we have to do two MapReduce phases as we want to display the district which contains the most trees.

Count district

Firstly, we count the number of trees per district. For the mapper, we create *CountDistrictsMapper* which gives the list of districts containing trees with district as key and 1 as value. Then using *IntSumReducer*, we concatenate and reduce, it returns the district as key and the sum of its values (number of trees) as value.

Concatenate

Then, we have to find the district containing the most trees. For the mapper *ConcatMapper*, we take the output of the first job and as we want to compare several information and return only one element, we reused what we have done in the previous exercise, the custom Writable class. We save the values of the district and number of trees as attributs of an object of our class *DoubleIntWritable*. So our outputs for the Mapper are on the form: an IntWritable (of value 1) for the key and a DoubleIntWritable object as the value.

For the reducer *ConcatMaxReducer*, we compare the different values of the key to return the maximum one. As the value of the key value couple, is a DoubleIntWritable object, we compare the attribut number of trees and keep the maximum one and its district. As output, we only return the district.

The Job class DistrictMost

The Job class *DistrictMost* was tricky. In fact, we had to create two jobs in one class knowing that the output of the first job would be the input of the second one. We store the result of the first job in a file called *temp* and we use it for the second one as an input. As in the previous exercise, for the second job, we didn't use a combiner as we have different input and output for the reducer and we have to set the output key and value class for the mapper.

1.8.7.2 Commands and result

Now let's take a look at the output. First we run:

```
-sh-4.2$ yarn jar  hadoop-examples-mapreduce-1.0-SNAPSHOT-jar-with-dependencies.jar  districtmost trees.csv mostout
```

We obtain two files:

```
-sh-4.2$ hdfs dfs -ls mostout
Found 2 items
drwxr-xr-x   - ccarayon hdfs          0 2020-11-03 23:02 mostout/out
drwxr-xr-x   - ccarayon hdfs          0 2020-11-03 23:01 mostout/temp
```

The intermediate result *temp*:

```
-sh-4.2$ hdfs dfs -cat mostout/temp/part-r-00000
11      1
12      29
13      2
14      3
15      1
16      36
17      1
18      1
19      6
20      3
3       1
4       1
5       2
6       1
7       3
8       5
9       1
```

And the final result *out*:

```
-sh-4.2$ hdfs dfs -cat mostout/out/part-r-00000
16      36
```

Conclusion

Thanks to this project, we learned how to create and implement our own MapReduce Jobs and use them in the cluster. We also made UnitTests with JUnit for the first time to check our implementation. Doing our own MapReduce Jobs was fastidious. We know that there are other frameworks and technologies that are easier and faster to perform this task, but it was instructive to build it from scratch.