Practice 3: Writing MapReduce Programs

Projects and Directories Used in this Exercise

Eclipse project: `practice_3`

MapReduce programs:

`MR2AvgWordLength.java (uses MR2)`

`SparkAvgWordLength.java (uses Spark)`

Test data (HDFS):

`shakespeare/poetry`

Exercise directory: `/home/cloudera/Desktop/workspace/practice_3`

In this exercise you write a MapReduce job that reads any text input and computes the average length of all words that start with each character.

For any text input, the job should report the average length of words that begin with ‘a’, ‘b’, and so forth. For example, for input:

No now is definitely not the time

The output would be:

N 2.0
n 3.0
d 10.0
i 2.0
t 3.5

(Your program should be case-sensitive as shown in this example.)

The Algorithm

The algorithm for this program is a simple two stage program:

Stage 1

For each word in the line, emit the first letter of the word as a key, and the length of the word as a value. For example, for input value:

No now is definitely not the time

Stage 1 should emit:

key	**value**
N	*2*
n	*3*
i	*2*
d	*10*
n	*3*
t	*3*
t	*4*

Stage 2

In the next stage, all the values for one key should be grouped together. After grouping, the results should be (though not necessarily in this order):

N	*(2)*
n	*(3,3)*
d	*(10)*
i	*(2)*
n	*(3,3)*
t	*(3,4)*

The final result should be should be (again, not in any specific order):

N	*2.0*
d	*10.0*
i	*2.0*
n	*3.0*
t	*3.5*

Write the MapReduce Programs in Java

- If you are using Eclipse, open Eclipse and then open the `practice_3` project.
- If you prefer to work in the shell, the files for practice_3 are in `~/Desktop/workspace/practice_3.`

The source for practice in `src/main/java`. There are **hints** and **solutions** for both MR2 and Spark implementations.

There are three packages: `hints, solutions` and `statsExample.`

In `hints` there are two files. If you want to use Spark, open `SparkAvgWordLength.java`. If you want to use MR2, open `MR2AvgWordLength.java`.

As you work, you can always refer back to the `practice_2` project or - if you get really stuck - you can look at the `solutions`.

Writing an MR2 program (MR2AvgWordLength.java)

Here are a few details to help you begin your `MR2` programming:

Define the main method.

This method should configure and submit your basic job. Among the basic steps here, define the input and output paths, configure the job with the Mapper class and the Reducer class you will write, and define the data types of the intermediate and final keys and values.

Define the Mapper.

Note these simple string operations in Java:

```
str.substring(0, 1)  // returns the first letter of str
str.length()         // returns the length of str
```

Define the Reducer

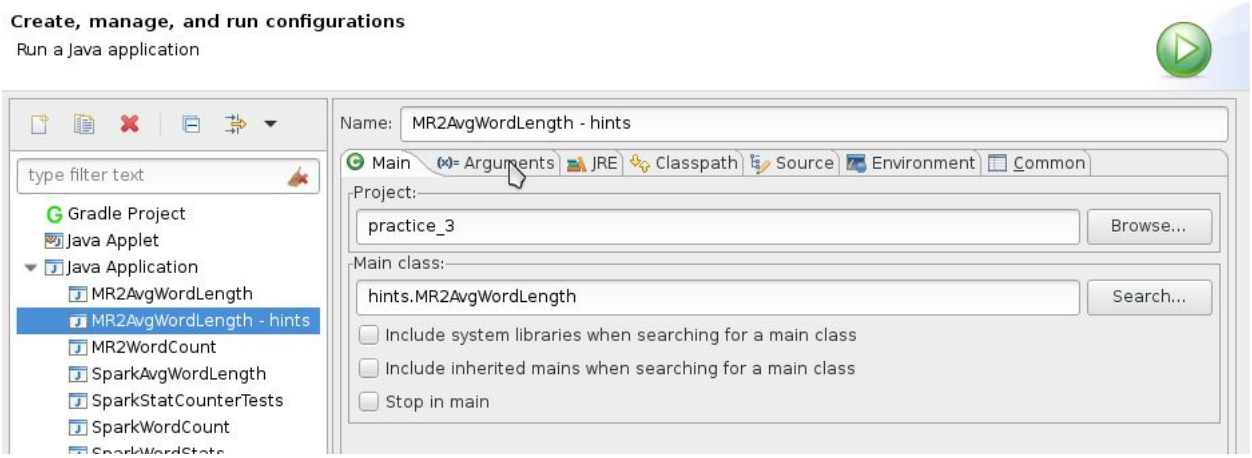
In a single invocation the reduce() method receives a string containing one letter (the key) along with an iterable collection of integers (the values), and should emit a single key-value pair: the letter and the average of the integers.

If you get stuck… that's why you also have the solutions.

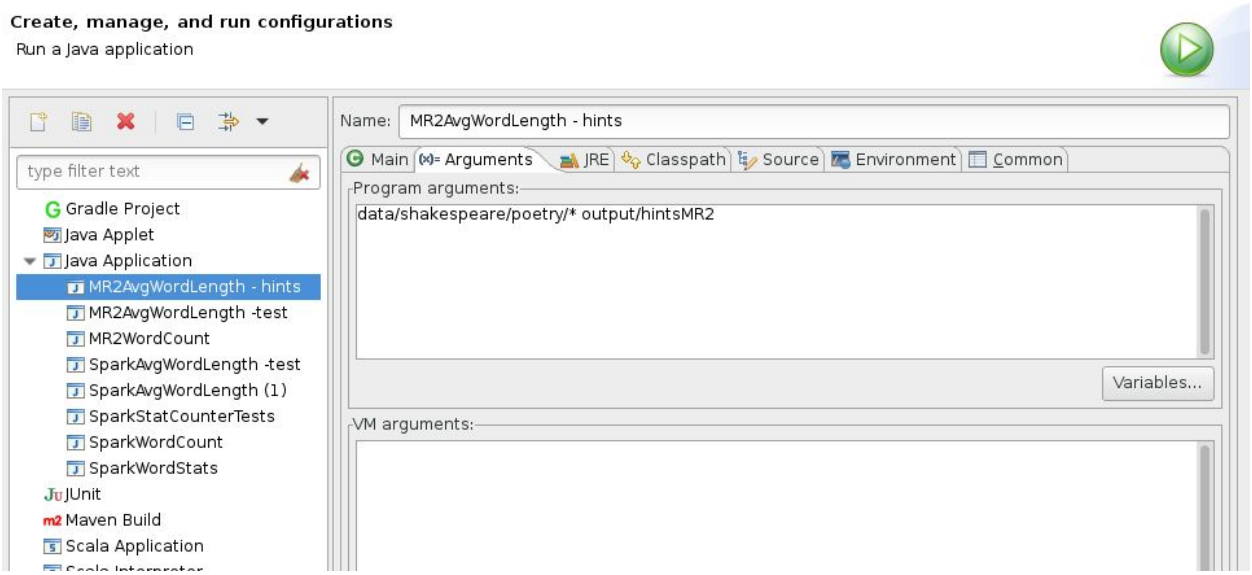
Use Eclipse to compile and test[1]

Name: MR2AvgWordLength - hints
Main class: hints.MR2AvgWordLength
Arguments:
Input data: shakespeare/poetry (in practice_3/data)
Output dir: <your choice>

Your `Run Configuration` should look like this:



and this:

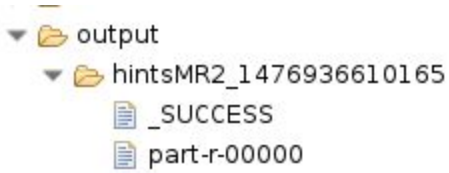


[1] If you forget how to run in Eclipse, the documentation is in practice_2 as Practice2:HowToRunHadoopJobs.

View the results

The output will be at the location you defined. *Don't forget to refresh your project!*

If you specified `output/hintsMR2` in the Run Configuration, your output will be here (with a slightly different timestamp[2])



The `part-r-00000` file should list all the numbers and letters in the data set, and the average length of the words starting with them. A portion of the result for shakespeare's poetry is shown below.

```
1 A   3.1134538152610443
2 B   3.6926952141057936
3 C   6.320224719101123
4 D   5.245614035087719
5 E   4.914893617021277
6 F   3.831168831168831
7 G   5.114754098360656
8 H   3.43768115942029
9 I   1.4577625570776256
10 J  5.3125
11 K  4.777777777777778
12 L  4.604026845637584
13 M  3.442982456140351
14 N  3.4827586206896552
15 O  1.9971014492753623
16 P  5.910714285714286
17 Q  5.5
18 R  5.739726027397261
19 S  4.30406852248394
20 T  3.7198529411764705
21 U  5.411764705882353
22 V  4.571428571428571
23 W  4.486803519061583
24 X  3.7903225806451615
25 Y  3.4634146341463414
26 a  3.335875436924055
```

[2] If you don't know where the timestamp came from, check your code out for the occurrence of `Calendar.getInstance().getTimeInMillis()`.

Writing an Spark program (SparkAvgWordLength.java)

Here are a few details to help you begin your `Spark` programming:

Define the Spark Configuration and JavaSparkContext

We are just starting out with applications, so the hints use a simple setup.

Reading the initial data

In this case, we are just reading a text file and counting the words, so we can load with textFile.

Define a simple map task composed of flatMap and mapToPair

- You can use `flatMap` to break input lines into words, assigning each word to its own output record. `flatMap` is often the first step in parsing a file, because it can be used to break lines into tokens.

- Then use `mapToPair` to generate a <key,value> pair using the tokens generated by `flatMap`.

- These string operations in Java may help when building the <key, value> pairs:

```
str.substring(0, 1)  // returns the first letter of str
str.length()         // returns the length of str
```

- In mapToPair, each input token *must* generate a <key,value> pair, so make sure that you handle empty input tokens.

Define an aggregating task using aggregateByKey and map

- To use **`aggregateByKey`**, provide two combining functions and a “zero value”.

- The combining functions are
  - seqFunc: `Function2`(U,V,U) for merging values within a partition
  - combFunc: `Function2`(U,U,U) for merging values between partitions
- This version of **`aggregateByKey`** will transform the <letter, length> pairs emitted by **`mapToPair`** and turn them into <letter,<sum of lengths, word count>> pairs.
- Next define **`map`**. `map` should take in a letter, a sum and a count. It should output the letter and an average.

If you get stuck… remember, you also have the solutions.

Use Eclipse to compile and test

Set up your Run Configuration like this:

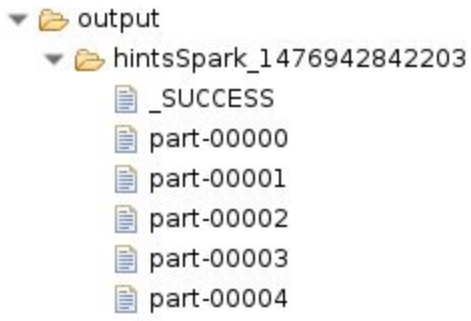
Name: SparkAvgWordLength - hints
Main class: hints.SparkAvgWordLength
Arguments:
Input data: shakespeare/poetry (in practice_3/data)
Output dir: <your choice>

If you need more information about how to run in Eclipse, the documentation is in practice_2 as **Practice2:HowToRunJobsOnHadoop.pdf**.

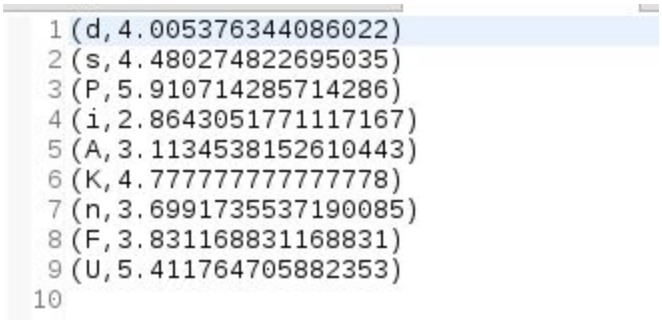
View the results

The output will be at the location you defined. *Don't forget to refresh your project!*

If you specified `output/hintsSpark` in the Run Configuration, your output will be here (with a different timestamp)



The `part-00000` file should list all the numbers and letters in the data set, and the average length of the words starting with them. A portion of the result for shakespeare's poetry is shown below.



We are not submitting our work to the cluster because practice_3 is using Java 1.8 and the Hadoop jar files on the cluster are compiled with Java 1.7 - there is a disconnect that causes unrecoverable errors when trying to run on the cluster.

So this is just an exercise in Maven, using Hadoop and Spark standalone.

End of Practice 3