**Word Count**

* Objective - count all words occurring in input data
* Source data - use any source data you have for this example

**Steps to Run the Word Count Program**

1. save file as WordCount.java
2. hdfs dfs -put <your input file with fully qualified path> /wordcount

* uploads your local input file (e.g., a text file for the word count program) to the Hadoop Distributed File System (HDFS)

1. mkdir -m 755 wordcount\_classes

* Creates a local directory named wordcount\_classes with permissions 755 and directory will be used to store the compiled Java classes before packaging them into a JAR file

1. javac -classpath $(hadoop classpath) -d wordcount\_classes WordCount.java

* the javac (Java compiler command), classpath $(hadoop classpath)- Specifies the class path needed for Hadoop libraries, -d wordcount\_classes (compiled class files will be placed) and WordCount.java (Java source file to compile)

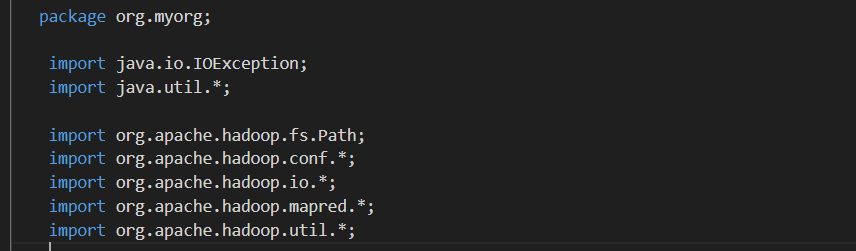
1. jar -cvf ${HOME}/scripts/wordcount.jar -C wordcount\_classes/ .

* jar: (create JAR file), ${HOME}/scripts/wordcount.jar: Path and name of the JAR file to be created and -C wordcount\_classes/ .: Changes to the wordcount\_classes directory and includes all files from it in the JAR.

1. hadoop jar ./wordcount.jar org.myorg.WordCount /wordcount /wordcount.res

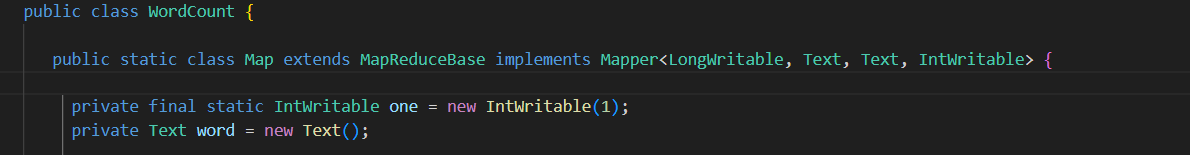
* Run the Hadoop MapReduce Job, the hadoop jar command runs the MapReduce job using the JAR file ./wordcount.jar, with org.myorg.WordCount as the main class. It processes input from /wordcount in HDFS and writes results to /wordcount.res.

1. Save the Below code with the name of WordCount.java.



package org.myorg: Defines the package for the Java class.

Imports: Includes necessary Hadoop libraries and classes for file handling, configuration, input/output operations, and MapReduce functionality.

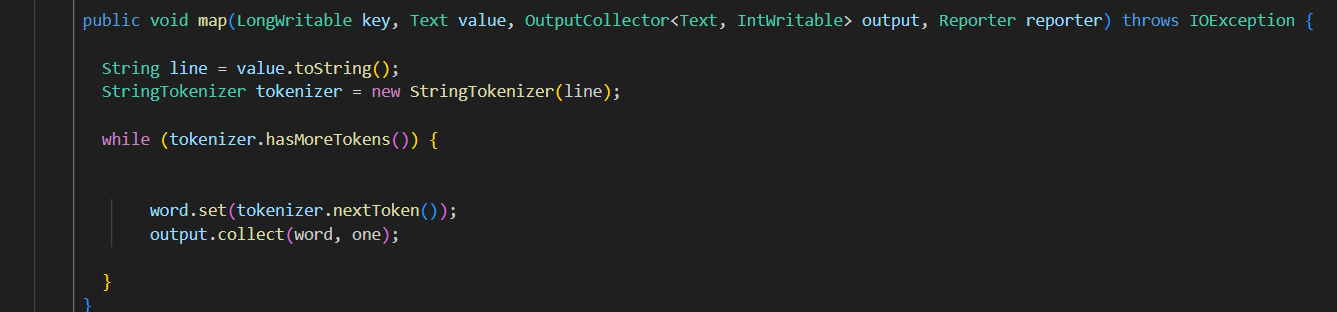


function: convert input into kay value pairs

LongWritable, Text, Text, IntWritable

- LongWritable, Text = 2 Inputs in this format

- Text, IntWritable = 2 outputs in this format



Inputs:

LongWritable key = binary position of every line read in the input file

Text value = complete line itself

Input ex:

My name is AA

Your name is BB

// break line into words

// put the below section in an if block which

// would only processes if the first letter of the input was

// a,e,i,o,u

Output - Round 1 - input to loop =[My, name, is, AA]:

(My,1)

(name,1)

(is,1)

(AA,1)

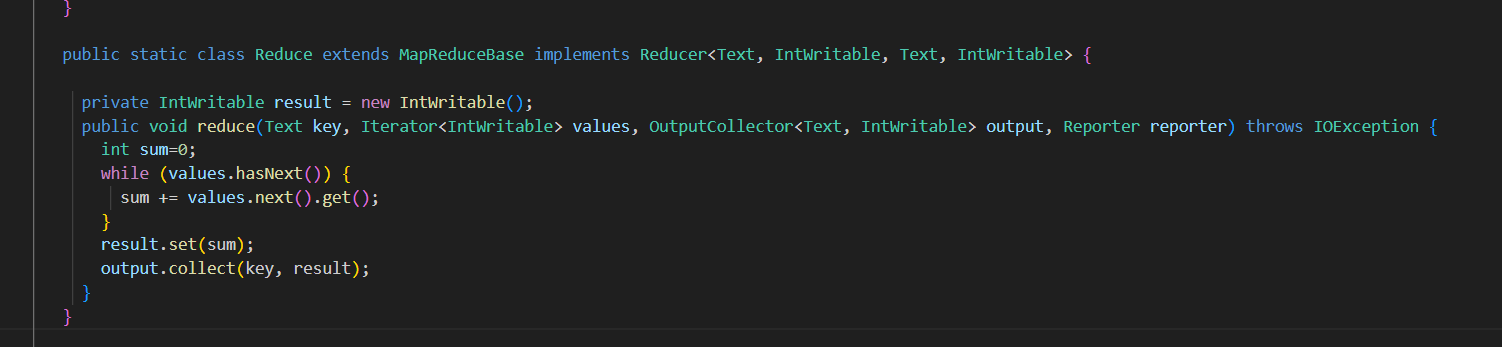
Output - Round 2 - input to loop =[You, name, is, BB]:

(Your,1)

(name,1)

(is,1)

(BB,1)



partitioning/shuffling/sorting before Reduce started

My, [(My,1)] => My, 1

name, [

1

1

] => name, 2

is, [

(is,1)

(is,1)

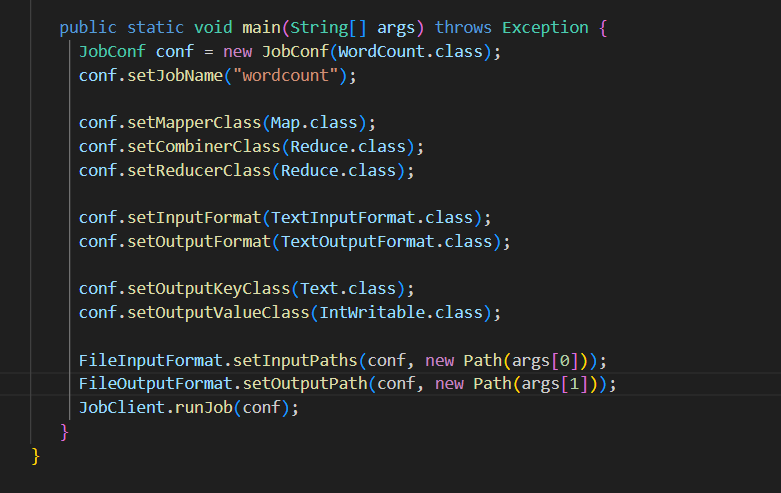
] => is, 2

AA, [(AA,1)] => AA, 1

Your,[(Your,1)] => Your, 1

BB,[(BB,1)] => BB, 1

* Reduce: A static inner class that implements the Reducer interface.
* Text, IntWritable, Text, IntWritable: Defines the input and output types for the reducer.
* Key & Values : The word being processed and an iterator over counts for that word.
* result.set(sum): Sets the total count for the word.
* output.collect(key, result): Emits the word with its total count.



setMapperClass: Specifies the mapper class.

setCombinerClass: Specifies the combiner class

setReducerClass: Specifies the reducer class.

setInputFormat: Specifies the input format (text file).

setOutputFormat: Specifies the output format (text file).

setOutputKeyClass and setOutputValueClass: Define the types of output key and value.

setInputPaths: Sets the input path for the job.

setOutputPath: Sets the output path for the job.

JobClient.runJob(conf): Submits and runs the MapReduce job using the configured settings.