YARN & MapReduce 1

Lucas BAKALIAN & Micha¨el HATOUM September 7, 2021

1

# YARN

In this tutorial, we will use MapReduce examples to learn how to work with Apache YARN.

## The MapReduce examples

The samples are located on the Hortonworks Data Platform cluster at:

/usr/odp/current/hadoop-mapreduce-client/hadoop-mapreduce-examples.jar

Source code for these samples is included on the ODP cluster at:

/usr/odp/current/hadoop-client/src/hadoop-mapreduce-project/hadoop-mapreduce-examples

The following samples are contained in this archive:

### Sample Description

aggregatewordcount Counts the words in the input files.

aggregatewordhist Computes the histogram of the words in the input files. bbp Uses Bailey-Borwein-Plouffe to compute exact digits of Pi.

dbcount Counts the pageview logs stored in a database.

distbbp Uses a BBP-type formula to compute exact bits of Pi.

grep Counts the matches of a regex in the input.

join Performs a join over sorted, equally partitioned datasets.

multifilewc Counts words from several files.

pentomino Tile laying program to find solutions to pentomino problems. pi Estimates Pi using a quasi-Monte Carlo method. randomtextwriter Writes 10 GB of random textual data per node. randomwriter Writes 10 GB of random data per node.

secondarysort Defines a secondary sort to the reduce phase. sort Sorts the data written by the random writer.

sudoku A sudoku solver.

teragen Generate data for the terasort.

terasort Run the terasort.

teravalidate Checking results of terasort.

wordcount Counts the words in the input files.

wordmean Counts the average length of the words in the input files. wordmedian Counts the median length of the words in the input files.

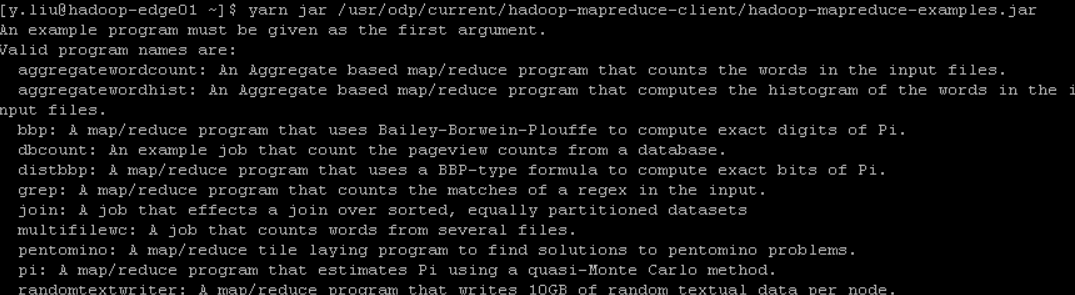
wordstandarddeviation Counts the standard deviation of the length of the words in the input files.

## Run the wordcount example

Connect to HADOOP cluster using SSH.

From the SSH session, use the following command to list the samples:

$ yarn j a r / usr / odp/ curre nt / hadoop-mapreduce-c l i e n t / hadoop-mapreduce-examples . j a r



This command generates the list of sample from the previous section of this document;

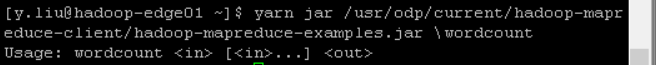
Use the following command to get help on a specific sample. In this case, the wordcount sample:

$ yarn j a r / usr / odp/ curre nt / hadoop-mapreduce-c l i e n t / hadoop-mapreduce-examples . j a r wordcount

*\*

You receive the following message:

Usage : wordcount *<*in *>* [*<* in *>* . . . ] *<*out*>*



This message indicates that you can provide several input paths for the source documents. The final path is where the output (count of words in the source documents) is stored.

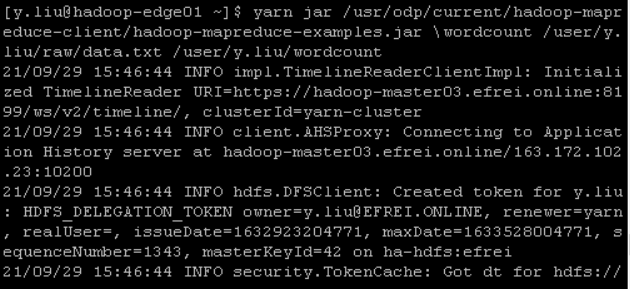
1）kinit

Use the following to count all words in downloaded e-book (in Plain Text UTF-8) from [Project Gutenberg](http://www.gutenberg.org/wiki/Main_Page):

$ yarn jar / usr/odp/current/ hadoop-mapreduce-client/hadoop-mapreduce-examples. jar \wordcount use r/*y.liu*/ d a vin c i . txt / use r/*y.liu*/wordcount

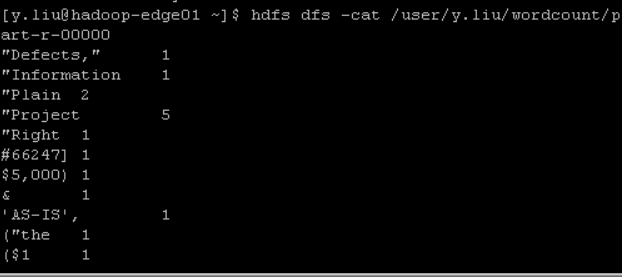
*\*

Input for this job is read from /user/*<*username*>*/davinci.txt. Output for this example is stored in /user/*<*username*>*/wordcount. Both paths are located on default storage for the cluster, not the local file system.



Once the job completes, use the following command to view the output:

$ hdfs d f s -cat / user/*y.liu*/wordcount/ part-r-00000



zum 1

zur 1

zwanzig 1

zwe i te 1

Each line represents a word and how many times it occurred in the input data.

## The Sudoku example

Sudoku is a logic puzzle made up of nine 3x3 grids. Some cells in the grid have numbers, while others are blank, and the goal is to solve for the blank cells.

The previous link has more information on the puzzle, but the purpose of this sample is to solve for the blank cells.

So our input should be a file that is in the following format:

Nine rows of nine columns;

Each column can contain either a number or ? (which indicates a blank cell);

Cells are separated by a space;

There is a certain way to construct Sudoku puzzles; you can’t repeat a number in a column or row.

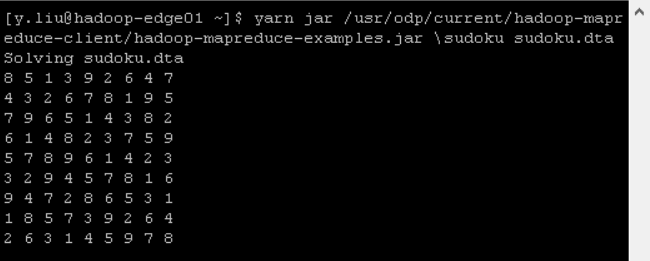
Create a file called *sudoku.dta* on your home local directory and contains the following text:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 8  ? | 5  ? | ?  2 | 3  ? | 9  ? | ?  ? | ?  ? | ?  ? | ?  ? |
| ? | ? | 6 | ? | 1 | ? | ? | ? | 2 |
| ? | ? | 4 | ? | ? | 3 | ? | 5 | 9 |
| ? | ? | 8 | 9 | ? | 1 | 4 | ? | ? |
| 3 | 2 | ? | 4 | ? | ? | 8 | ? | ? |
| 9 | ? | ? | ? | 8 | ? | 5 | ? | ? |
| ? | ? | ? | ? | ? | ? | 2 | ? | ? |
| ? | ? | ? | ? | 4 | 5 | ? | 7 | 8 |

To run this example problem through the Sudoku example, use the following command:

$ yarn j a r / usr / odp/ curre nt / hadoop-mapreduce-c l i e n t / hadoop-mapreduce-examples . j a r sudoku sudoku . dta

*\*



The results appear similar to the following text:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 8 | 5 | 1 | 3 | 9 | 2 | 6 | 4 | 7 |
| 4 | 3 | 2 | 6 | 7 | 8 | 1 | 9 | 5 |
| 7 | 9 | 6 | 5 | 1 | 4 | 3 | 8 | 2 |
| 6 | 1 | 4 | 8 | 2 | 3 | 7 | 5 | 9 |
| 5 | 7 | 8 | 9 | 6 | 1 | 4 | 2 | 3 |
| 3 | 2 | 9 | 4 | 5 | 7 | 8 | 1 | 6 |
| 9 | 4 | 7 | 2 | 8 | 6 | 5 | 3 | 1 |
| 1 | 8 | 5 | 7 | 3 | 9 | 2 | 6 | 4 |
| 2 | 6 | 3 | 1 | 4 | 5 | 9 | 7 | 8 |

## Pi example

The pi sample uses a statistical (quasi-Monte Carlo) method to estimate the value of pi. Points are placed at random in a unit square.

The square also contains a circle. The probability that the points fall within the circle is equal to the area of the circle, pi/4.

The value of pi can be estimated from the value of 4R. R is the ratio of the number of points that are inside the circle to the total number of points that are within the square.

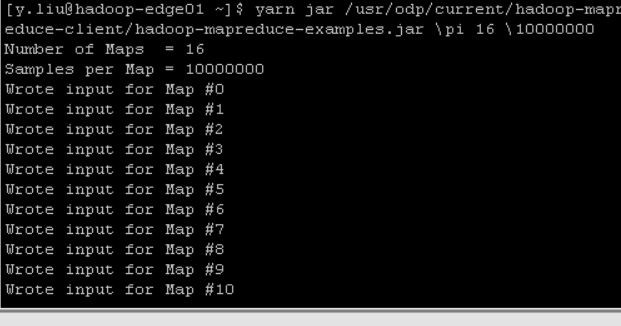
The larger the sample of points used, the better the estimate is.

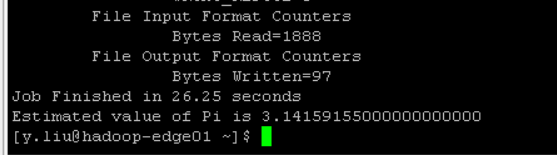
Use the following command to run this sample. This command uses 16 maps with 10,000,000 samples each to estimate the value of pi:

$ yarn jar /usr/odp/current/hadoop-mapreduce-client/hadoop-mapreduce-examples.jar \pi 16 10000000

*\*

The value returned by this command is similar to 3.14159155000000000000. For references, the first 10 decimal places of pi are 3.1415926535.





## 10 GB GraySort example

GraySort is a benchmark sort. The metric is the sort rate (TB/minute) that is achieved while sorting large amounts of data, usually a 100 TB minimum.

This sample uses a modest 10 GB of data so that it can be run relatively quickly. It uses the MapReduce applications developed by Owen O’Malley and Arun Murthy.

These applications won the annual general-purpose (”Daytona”) terabyte sort benchmark in 2009, with a rate of 0.578 TB/min (100 TB in 173 minutes). For more information on this and other sorting benchmarks, see the Sort Benchmark [site](https://sortbenchmark.org/).

This sample uses three sets of MapReduce programs:

TeraGen: A MapReduce program that generates rows of data to sort;

TeraSort: Samples the input data and uses MapReduce to sort the data into a total order;

TeraSort is a standard MapReduce sort, except for a custom partitioner. The partitioner uses a sorted list of N-1 sampled keys that define the key range for each reduce. In particular, all keys such that sample[i-1] *<*= key *<* sample[i] are sent to reduce i. This partitioner guarantees that the outputs of reduce i are all less than the output of reduce i+1.

TeraValidate: A MapReduce program that validates that the output is globally sorted;

It creates one map per file in the output directory, and each map en- sures that each key is less than or equal to the previous one. The map function generates records of the first and last keys of each file. The reduce function ensures that the first key of file i is greater than the last key of file i-1. Any problems are reported as an output of the reduce phase, with the keys that are out of order.

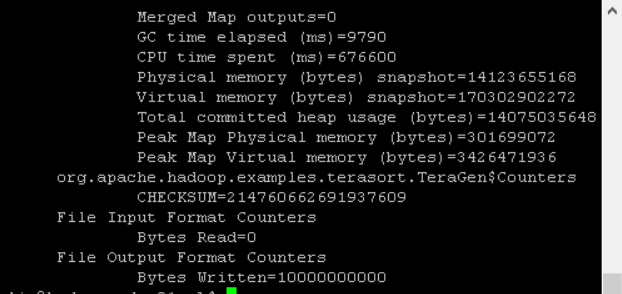
Use the following steps to generate data, sort, and then validate the output:

Generate 10 GB of data, which is stored to your home directory at /user/*<*username*>*/data/10GB- sort-input:

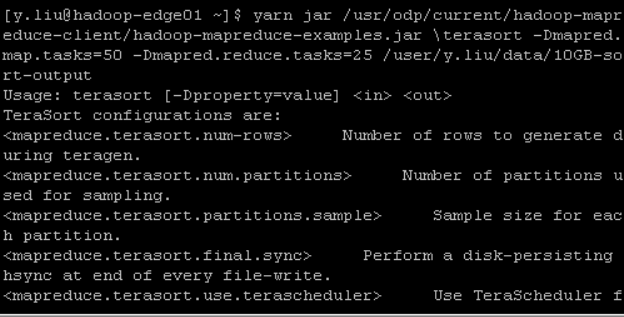
$ yarn j a r / usr / odp/ curre nt / hadoop=mapreduce=c l i e n t / hadoop=mapreduce=examples . j a r te rage n =Dmapred . map . ta s ks =50 100000000 / use r/*<*username *>*/data /10GB=s o rt =input

*\*

The -Dmapred.map.tasks tells Hadoop how many map tasks to use for this job. The final two parameters instruct the job to create 10 GB of data and to store it at /user/*<*username*>*/data/10GB-sort-input.



Use the following command to sort the data:

$ yarn j a r / usr / odp/ curre nt / hadoop=mapreduce=c l i e n t / hadoop=mapreduce=examples . j a r t e r a s o r t =Dmapred . map . ta s ks =50 =Dmapred . reduce . ta s ks =25

*\*

*\*

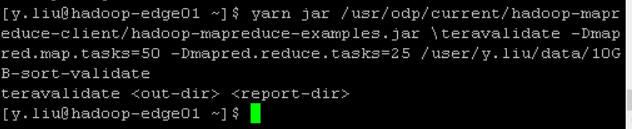
/ use r/*<*username *>*/data /10GB=s o rt =input / use r/*<*username *>*/data /10GB=s o rt =output

The -Dmapred.reduce.tasks tells Hadoop how many reduce tasks to use for the job. The final two parameters are just the input and output locations for data.

Use the following to validate the data generated by the sort:

$ yarn jar /usr/odp/current/hadoop-mapreduce-client/hadoop-mapreduce-examples.jar \teravalidate -Dmapred.map.tasks=50 -Dmapred.reduce.tasks=25 \

/user/j.qu/data/10GB-sort-output /user/j.qu/data/10GB-sort -validate



# MapReduce2

In this tutorial I will describe how to write a simple MapReduce program for Hadoop in the Python programming language.

## Motivation

Even though the Hadoop framework is written in Java, programs for Hadoop need not to be coded in Java but can also be developed in other languages like Python or C++ (the latter since version 0.14.1).

However, Hadoop’s documentation and the most prominent Python example on the Hadoop website could make you think that you must translate your Python code using Jython into a Java jar file.

Obviously, this is not very convenient and can even be problematic if you de- pend on Python features not provided by Jython.

Another issue of the Jython approach is the overhead of writing your Python program in such a way that it can interac twith Hadoop – just have a look at the example in $HADOOP HOME/src/examples/python/WordCount.py and you see what I mean.

That said, the ground is now prepared for the purpose of this tutorial: writing a Hadoop MapReduce program in a more Pythonic way, i.e. in a way you should be familiar with.

## What we want to do

We will write a simple MapReduce program (see also the MapReduce article on Wikipedia) for Hadoop in Python but without using Jython to translate our code to Java jar files.

Our program will mimick the WordCount, i.e. it reads text files and counts how often words occur. The input is text files and the output is text files, each line of which contains a word and the count ofhow often it occured, separated by a tab.

*Note: You can also use programming languages other than Python such as Perl or Ruby with the ”technique” described in this tutorial.*

## Python MapReduce Code

The “trick” behind the following Python code is that we will use the [Hadoop](https://cwiki.apache.org/confluence/display/HADOOP2/HadoopStreaming) [Streaming API](https://cwiki.apache.org/confluence/display/HADOOP2/HadoopStreaming) for helping us passing data between our Map and Reduce code via STDIN (standard input) and STDOUT (standard output).

We will simply use Python’s sys.stdin to read input data and print our own output to sys.stdout. That’s all weneed to do because Hadoop Streaming will take care of everything else!

### Map step: mapper.py

Save the following code in the file /home/*<*username*>*/mapper.py.

It will read data from STDIN, split it into words and output a list of lines mapping words to their (intermediate) counts to STDOUT.

The Map script will not compute an (intermediate) sum of a word’s occur- rences though. Instead, it will output *<*word*>* 1 tuples immediately – even though a specific word might occur multiple times in the input.

In our case we let the subsequent Reduce step do the final sum count. Of course, you can change this behavior in your own scripts as you please, but we will keep it like that in this tutorial because of didactic reasons. :-)

Make sure the file has execution permission or you will run into problems: chmod

+x /home/*<*username*>*/mapper.py.



You have to fill in the holes where the @TODO is written :

#!/ usr / bin / env python """ mapper . py """

import sys

# input comes from STDIN ( standard input ) for line in sys . stdin :

# remove leading and trailing whitespace # @TODO

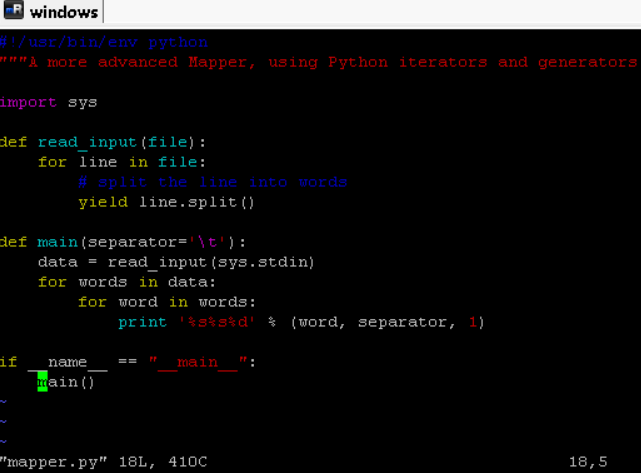
# split the line into words # @TODO

# increase counters for word in words :

# write the results to STDOUT ( standard output ); # what we output here will be the input for the # Reduce step , i. e. the input for reducer . py

#

# tab - delimited ; the trivial word count is 1 # @TODO

“

### Reduce step: reducer.py

Save the following code in the file /home/*<*username*>*/reducer.py.

It will read the results of mapper.py from STDIN (so the output format of mapper.py and the expected input format of reducer.py must match) and sum the occurrences of each word to a final count, and then output its results to STDOUT.

Make sure the file has execution permission or you will run into problems: Make sure the file has execution permission or you will run into problems: chmod

+x /home/*<*username*>*/reducer.py.

You have to fill in the holes where the @TODO is written :

#! / usr / bin / env python ””” re duce r . py”””

from o pe rato r import i te m g e tte r import sys

current word = None c urre nt c o unt = 0 word = None

# input comes from STDIN f o r l i n e in sys . s td in :

# remove l e a d in g and t r a i l i n g whitespace # @TODO

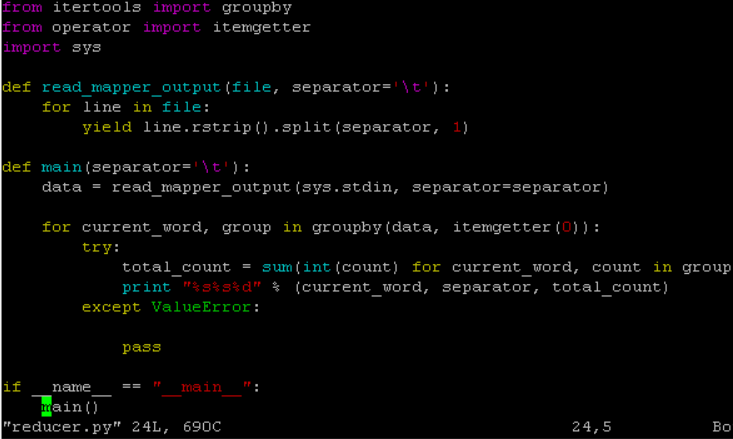
# parse the input we got from mapper . py # @TODO

# convert count ( c u r r e n t ly a s t r i n g ) to in t # @TODO

# t h i s IF= switch only works because Hadoop s o r t s map output # by key ( here : word ) b e f o re i t i s passed to the re duce r

# @TODO

# do not f o r g e t to output the l a s t word i f needed ! # @TODO



### Test your code (cat data *|* map *|* sort *|* reduce)

I recommend to test your mapper.py and reducer.py scripts locally before us- ing them in a MapReduce job.

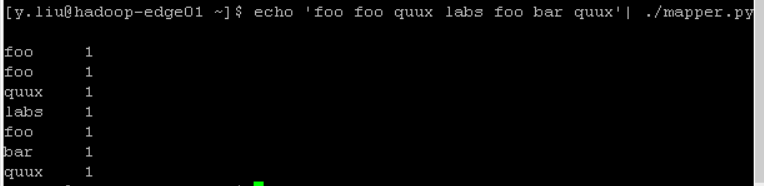
Otherwise your jobs might successfully complete but there will be no job re- sult data at all or not the results you would have expected.

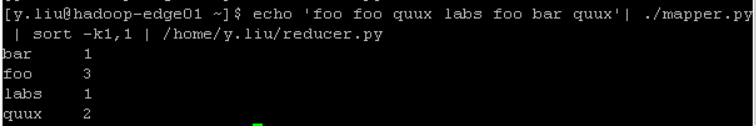
If that happens, most likely it was you (or me) who screwed up. Here are some ideas on how to test the functionality of the Map and Reduce scripts.

$ echo ” f o o f o o quux l a b s f o o bar quux” *|* /home/ username / mapper . py

$ echo ” f o o f o o quux l a b s f o o bar quux” *|* /home/ username / mapper . py *|* s o r t =k1 , 1 *|* /home/ username / r e d u c e r . py

You can use a downloaded e-book (in Plain Text UTF-8) from [Project Guten-](http://www.gutenberg.org/wiki/Main_Page) [berg](http://www.gutenberg.org/wiki/Main_Page).





## Running the Python Code on Hadoop

### Download example input data

We will use three ebooks from Project Gutenberg for this example:

The Outline of Science, Vol. 1 (of 4) by J. Arthur Thomson;

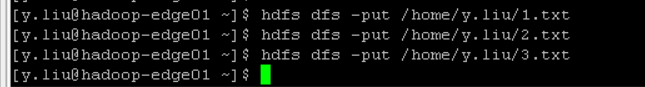
The Notebooks of Leonardo Da Vinci;

Ulysses by James Joyce;

Download each ebook as text files in Plain Text UTF-8 encoding and store the files in your home directory.

### Copy local example data to HDFS

Before we run the actual MapReduce job, we must first copy the files from our local file system to Hadoop’s HDFS.



### Run the MapReduce job

Now that everything is prepared, we can finally run our Python MapReduce job on the Hadoop cluster.

As I said above, we leverage the Hadoop Streaming API for helping us passing data between our Map and Reduce code via STDIN and STDOUT.

If you want to modify some Hadoop settings on the fly like increasing the num- ber of Reduce tasks, you can use the -D option.

The job will read all the files in the HDFS directory /user/*<*username*>*/gutenberg,

process it, and store the results in the HDFS directory /user/*<*username*>*/gutenberg-output.

In general Hadoop will create one output file per reducer; in our case how- ever it will only create a single file because the input files are very small.

Check if the result is successfully stored in HDFS directory /user/*<*username*>*/gutenberg-output.

### Improved Mapper and Reducer code: using Python iterators and generators

The Mapper and Reducer examples above should have given you an idea of how to create your first MapReduce application.

The focus was code simplicity and ease of understanding, particularly for be- ginners of the Python programming language.

In a real-world application however, you might want to optimize your code by using [Python iterators and generators](https://developer.ibm.com/articles/l-pycon/).

Generally speaking, iterators and generators (functions that create iterators, for example with Python’s yield statement) have the advantage that an ele- ment of a sequence is not produced until you actually need it.

This can help a lot in terms of computational expensiveness or memory con- sumption depending on the task at hand.

*Note: The following Map and Reduce scripts will only work ”correctly” when being run in the Hadoop context, i.e. as Mapper and Reducer in a MapReduce job. This means that running the naive test command ”cat DATA ./mapper.py sort -k1,1 ./reducer.py” will not work correctly anymore because some func-*

*| |*

*|*

*tionality is intentionally outsourced to Hadoop.*

Precisely, we compute the sum of a word’s occurrences, e.g. ("foo", 4), only if by chance the same word (foo) appears multiple times in succession.

In the majority of cases, however, we let the Hadoop group the (key, value) pairs between the Map and the Reduce step because Hadoop is more efficient in this regard than our simple Python scripts.

### mapper.py

You have to fill in the holes where the @TODO is written :

#! / usr / bin / env python

# A more advanced Mapper , using Python i t e r a t o r s and g e n e ra to rs . import sys

de f re ad input ( f i l e ) : f o r l i n e in f i l e :

# s p l i t the l i n e in to words # @TODO

de f main ( s e p a ra to r=’ *\* t ’ ) :

# input comes from STDIN ( standard input ) # @TODO

f o r words in data :

# w r i te the r e s u l t s to STDOUT ( standard output ) ; # what we output here w i l l be the input f o r the # Reduce step , i . e . the input f o r re ducer . py

#

# tab= d e l im i te d ; the t r i v i a l word count i s 1 # @TODO

i f name == ” m a in ” : main ()

### reducer.py

You have to fill in the holes where the @TODO is written :

#! / usr / bin / env python

# A more advanced Reducer , using Python i t e r a t o r s and g e n e ra to rs .

from i t e r t o o l s import groupby from o pe rato r import i te m g e tte r import sys

de f read mapper output ( f i l e , s e p a ra to r=’ *\* t ’ ) :

# @TODO

de f main ( s e p a ra to r=’ *\* t ’ ) :

# input comes from STDIN ( standard input )

data = read mapper output ( sys . stdin , s e p a ra to r=s e p a ra to r ) # groupby groups m u l t ip le word=count p a i r s by word ,

# and c r e a t e s an i t e r a t o r that re tu rn s c o n s e c u t ive keys and t h e i r group : # current word = s t r i n g c o n ta in in g a word ( the key )

# group = i t e r a t o r y i e l d i n g a l l [”*<* current word *>*”, ”*<*count *>*”] items # @TODO

i f name == ” m a in ” : main ()