Writing An Hadoop MapReduce Program In Python

1. Prerequisites
2. Python MapReduce Code
   * Map step: mapper.py
   * Reduce step: reducer.py
   * Test your code (cat data | map | sort | reduce)
3. Running the Python Code on Hadoop
   * Download example input data
   * Copy local example data to HDFS
   * Run the MapReduce job
4. Improved Mapper and Reducer code: using Python iterators and generators
   * mapper.py
   * reducer.py
5. Related Links

# 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 prominentPython 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 depend 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 interact with Hadoop – just have a look at the example in $HADOOP\_HOME/src/examples/python/WordCount.pyand 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 of how 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.

# Prerequisites

You should have an Hadoop cluster up and running because we will get our hands dirty. If you don’t have a cluster yet, my following tutorials might help you to build one. The tutorials are tailored to Ubuntu Linux but the information does also apply to other Linux/Unix variants.

* Running Hadoop On Ubuntu Linux (Single-Node Cluster) – How to set up a pseudo-distributed, single-node Hadoop cluster backed by the Hadoop Distributed File System (HDFS)
* Running Hadoop On Ubuntu Linux (Multi-Node Cluster) – How to set up a distributed, multi-nodeHadoop cluster backed by the Hadoop Distributed File System (HDFS)

# Python MapReduce Code

The “trick” behind the following Python code is that we will use the Hadoop Streaming API (see also the corresponding wiki entry) 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 we need to do because Hadoop Streaming will take care of everything else!

## Map step: mapper.py

Save the following code in the file /home/hduser/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 occurrences 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 (chmod +x /home/hduser/mapper.py should do the trick) or you will run into problems.

*#!/usr/bin/env python*

"""mapper.py"""

import sys

*# input comes from STDIN (standard input)*

**for** line **in** sys**.**stdin:

*# remove leading and trailing whitespace*

line **=** line**.**strip()

*# split the line into words*

words **=** line**.**split()

*# 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*

**print** '%s\t%s' **%** (word, 1)

## Reduce step: reducer.py

Save the following code in the file /home/hduser/reducer.py. It will read the results of mapper.pyfrom 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 (chmod +x /home/hduser/reducer.py should do the trick) or you will run into problems.

*#!/usr/bin/env python*

"""reducer.py"""

from operator import itemgetter

import sys

current\_word **=** None

current\_count **=** 0

word **=** None

*# input comes from STDIN*

**for** line **in** sys**.**stdin:

*# remove leading and trailing whitespace*

line **=** line**.**strip()

*# parse the input we got from mapper.py*

word, count **=** line**.**split('\t', 1)

*# convert count (currently a string) to int*

**try**:

count **=** int(count)

**except** ValueError:

*# count was not a number, so silently*

*# ignore/discard this line*

**continue**

*# this IF-switch only works because Hadoop sorts map output*

*# by key (here: word) before it is passed to the reducer*

**if** current\_word **==** word:

current\_count **+=** count

**else**:

**if** current\_word:

*# write result to STDOUT*

**print** '%s\t%s' **%** (current\_word, current\_count)

current\_count **=** count

current\_word **=** word

*# do not forget to output the last word if needed!*

**if** current\_word **==** word:

**print** '%s\t%s' **%** (current\_word, current\_count)

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

I recommend to test your mapper.py and reducer.py scripts locally before using them in a MapReduce job. Otherwise your jobs might successfully complete but there will be no job result 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.

*# Test mapper.py and reducer.py locally first*

*# very basic test*

hduser@ubuntu:~$ echo "foo foo quux labs foo bar quux" | /home/hduser/mapper.py

foo 1

foo 1

quux 1

labs 1

foo 1

bar 1

quux 1

hduser@ubuntu:~$ echo "foo foo quux labs foo bar quux" | /home/hduser/mapper.py | sort -k1,1 | /home/hduser/reducer.py

bar 1

foo 3

labs 1

quux 2

*# using one of the ebooks as example input*

*# (see below on where to get the ebooks)*

hduser@ubuntu:~$ cat /tmp/gutenberg/20417-8.txt | /home/hduser/mapper.py

The 1

Project 1

Gutenberg 1

EBook 1

of 1

**[**...]

**(**you get the idea**)**

# 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 a local temporary directory of choice, for example /tmp/gutenberg.

hduser@ubuntu:~$ ls -l /tmp/gutenberg/

total 3604

-rw-r--r-- 1 hduser hadoop 674566 Feb 3 10:17 pg20417.txt

-rw-r--r-- 1 hduser hadoop 1573112 Feb 3 10:18 pg4300.txt

-rw-r--r-- 1 hduser hadoop 1423801 Feb 3 10:18 pg5000.txt

hduser@ubuntu:~$

## 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.

hduser@ubuntu:/usr/local/hadoop$ bin/hadoop dfs -copyFromLocal /tmp/gutenberg /user/hduser/gutenberg

hduser@ubuntu:/usr/local/hadoop$ bin/hadoop dfs -ls

Found 1 items

drwxr-xr-x - hduser supergroup 0 2010-05-08 17:40 /user/hduser/gutenberg

hduser@ubuntu:/usr/local/hadoop$ bin/hadoop dfs -ls /user/hduser/gutenberg

Found 3 items

-rw-r--r-- 3 hduser supergroup 674566 2011-03-10 11:38 /user/hduser/gutenberg/pg20417.txt

-rw-r--r-- 3 hduser supergroup 1573112 2011-03-10 11:38 /user/hduser/gutenberg/pg4300.txt

-rw-r--r-- 3 hduser supergroup 1423801 2011-03-10 11:38 /user/hduser/gutenberg/pg5000.txt

hduser@ubuntu:/usr/local/hadoop$

## 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.

hduser@ubuntu:/usr/local/hadoop$ bin/hadoop jar contrib/streaming/hadoop-**\***streaming**\***.jar \

-file /home/hduser/mapper.py -mapper /home/hduser/mapper.py \

-file /home/hduser/reducer.py -reducer /home/hduser/reducer.py \

-input /user/hduser/gutenberg/**\*** -output /user/hduser/gutenberg-output

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

hduser@ubuntu:/usr/local/hadoop$ bin/hadoop jar contrib/streaming/hadoop-**\***streaming**\***.jar -D mapred.reduce.tasks**=**16 ...

Note about mapred.map.tasks: Hadoop does not honor mapred.map.tasks beyond considering it a hint. But it accepts the user specified mapred.reduce.tasks and doesn't manipulate that. You cannot force mapred.map.tasks but can specify mapred.reduce.tasks.

The job will read all the files in the HDFS directory /user/hduser/gutenberg, process it, and store the results in the HDFS directory /user/hduser/gutenberg-output. In general Hadoop will create one output file per reducer; in our case however it will only create a single file because the input files are very small.

Example output of the previous command in the console:

hduser@ubuntu:/usr/local/hadoop$ bin/hadoop jar contrib/streaming/hadoop-**\***streaming**\***.jar -mapper /home/hduser/mapper.py -reducer /home/hduser/reducer.py -input /user/hduser/gutenberg/**\*** -output /user/hduser/gutenberg-output

additionalConfSpec\_:null

null**=**@@@userJobConfProps\_.get**(**stream.shipped.hadoopstreaming

packageJobJar: **[**/app/hadoop/tmp/hadoop-unjar54543/]

**[]** /tmp/streamjob54544.jar tmpDir**=**null

**[**...] INFO mapred.FileInputFormat: Total input paths to process : 7

**[**...] INFO streaming.StreamJob: getLocalDirs**()**: **[**/app/hadoop/tmp/mapred/local]

**[**...] INFO streaming.StreamJob: Running job: job\_200803031615\_0021

**[**...]

**[**...] INFO streaming.StreamJob: map 0% reduce 0%

**[**...] INFO streaming.StreamJob: map 43% reduce 0%

**[**...] INFO streaming.StreamJob: map 86% reduce 0%

**[**...] INFO streaming.StreamJob: map 100% reduce 0%

**[**...] INFO streaming.StreamJob: map 100% reduce 33%

**[**...] INFO streaming.StreamJob: map 100% reduce 70%

**[**...] INFO streaming.StreamJob: map 100% reduce 77%

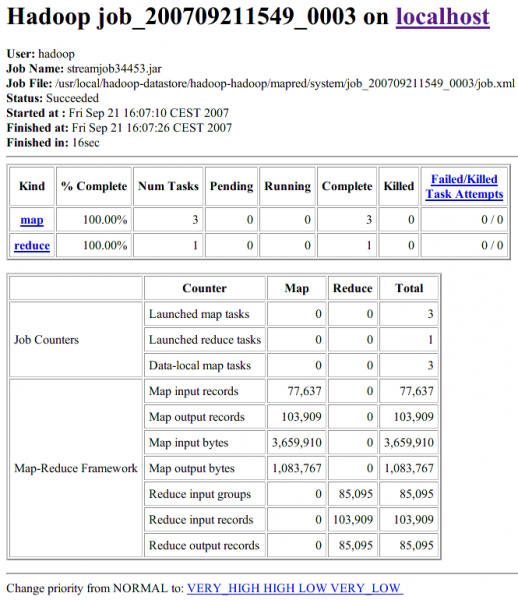
**[**...] INFO streaming.StreamJob: map 100% reduce 100%

**[**...] INFO streaming.StreamJob: Job complete: job\_200803031615\_0021

**[**...] INFO streaming.StreamJob: Output: /user/hduser/gutenberg-output

hduser@ubuntu:/usr/local/hadoop$

As you can see in the output above, Hadoop also provides a basic web interface for statistics and information. When the Hadoop cluster is running, open http://localhost:50030/ in a browser and have a look around. Here’s a screenshot of the Hadoop web interface for the job we just ran.



*Figure 1: A screenshot of Hadoop's JobTracker web interface, showing the details of the MapReduce job we just ran*

Check if the result is successfully stored in HDFS directory /user/hduser/gutenberg-output:

hduser@ubuntu:/usr/local/hadoop$ bin/hadoop dfs -ls /user/hduser/gutenberg-output

Found 1 items

/user/hduser/gutenberg-output/part-00000 &lt;r 1&gt; 903193 2007-09-21 13:00

hduser@ubuntu:/usr/local/hadoop$

You can then inspect the contents of the file with the dfs -cat command:

hduser@ubuntu:/usr/local/hadoop$ bin/hadoop dfs -cat /user/hduser/gutenberg-output/part-00000

"(Lo)cra" 1

"1490 1

"1498," 1

"35" 1

"40," 1

"A 2

"AS-IS". 2

"A\_ 1

"Absoluti 1

**[**...]

hduser@ubuntu:/usr/local/hadoop$

Note that in this specific output above the quote signs (") enclosing the words have not been inserted by Hadoop. They are the result of how our Python code splits words, and in this case it matched the beginning of a quote in the ebook texts. Just inspect the part-00000 file further to see it for yourself.

# 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 beginners of the Python programming language. In a real-world application however, you might want to optimize your code by using Python iterators and generators (an even better introduction in PDF).

Generally speaking, iterators and generators (functions that create iterators, for example with Python’s yield statement) have the advantage that an element of a sequence is not produced until you actually need it. This can help a lot in terms of computational expensiveness or memory consumption 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 functionality 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

*#!/usr/bin/env python*

"""A more advanced Mapper, using Python iterators and generators."""

import sys

**def** **read\_input**(file):

**for** line **in** file:

*# split the line into words*

**yield** line**.**split()

**def** **main**(separator**=**'\t'):

*# input comes from STDIN (standard input)*

data **=** read\_input(sys**.**stdin)

**for** words **in** data:

*# 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*

**for** word **in** words:

**print** '%s%s%d' **%** (word, separator, 1)

**if** \_\_name\_\_ **==** "\_\_main\_\_":

main()

## reducer.py

*#!/usr/bin/env python*

"""A more advanced Reducer, using Python iterators and generators."""

from itertools import groupby

from operator import itemgetter

import sys

**def** **read\_mapper\_output**(file, separator**=**'\t'):

**for** line **in** file:

**yield** line**.**rstrip()**.**split(separator, 1)

**def** **main**(separator**=**'\t'):

*# input comes from STDIN (standard input)*

data **=** read\_mapper\_output(sys**.**stdin, separator**=**separator)

*# groupby groups multiple word-count pairs by word,*

*# and creates an iterator that returns consecutive keys and their group:*

*# current\_word - string containing a word (the key)*

*# group - iterator yielding all ["&lt;current\_word&gt;", "&lt;count&gt;"] items*

**for** current\_word, group **in** groupby(data, itemgetter(0)):

**try**:

total\_count **=** sum(int(count) **for** current\_word, count **in** group)

**print** "%s%s%d" **%** (current\_word, separator, total\_count)

**except** ValueError:

*# count was not a number, so silently discard this item*

**pass**

**if** \_\_name\_\_ **==** "\_\_main\_\_":

main()

|  |
| --- |
|  |
| #!/usr/bin/env python26 |
|  |  |
|  | """Python HDFS use examples. |
|  |  |
|  | After reading this example you should have enough information to read and write |
|  | HDFS files from your programs. |
|  | """ |
|  |  |
|  | from hdfs.hfile import Hfile |
|  |  |
|  | hostname = 'hadoop.twitter.com' |
|  | port = 8020 |
|  | hdfs\_path = '/user/travis/example' |
|  | local\_path = '/etc/motd' |
|  |  |
|  | # Let's open local and HDFS files. |
|  |  |
|  | hfile = Hfile(hostname, port, hdfs\_path, mode='w') |
|  | fh = open(local\_path) |
|  |  |
|  | # Now we'll write lines from a local file into the HDFS file. |
|  | for line in fh: |
|  | hfile.write(line) |
|  |  |
|  | # And close them. |
|  | fh.close() |
|  | hfile.close() |
|  |  |
|  | # Let's read local\_path into memory for comparison. |
|  | motd = open(local\_path).read() |
|  |  |
|  | # Now let's read the data back |
|  | hfile = Hfile(hostname, port, hdfs\_path) |
|  |  |
|  | # With an iterator |
|  | data\_read\_from\_hdfs = '' |
|  | for line in hfile: |
|  | data\_read\_from\_hdfs += line |
|  | print motd == data\_read\_from\_hdfs |
|  |  |
|  | # All at once |
|  | data\_read\_from\_hdfs = hfile.read() |
|  | print motd == data\_read\_from\_hdfs |
|  |  |
|  | hfile.close() |
|  |  |
|  | # Hopefully you have enough info to get started! |
|  |  |
|  | from hdfs.hfilesystem import Hfilesystem |
|  | hfs = Hfilesystem(hostname, port) |
|  | print hfs.getHosts(hdfs\_path, 0, 1) |
|  |  |