Leveraging Unstructured Data - Lab 3: Submit Dataproc jobs for unstructured data v1.3

## Overview

Unstructured data includes data that is without a schema and data that has a structure, but which is not useful for the intended purpose.

In this lab you will learn about Spark and the framework of Resilient Distributed Datasets (RDDs) and operations for working with big data and unstructured data.

## Objectives

In this lab, you will perform the following tasks:

* Explore HDFS and Cloud Storage
* Use interactive PySpark to learn about RDDs, Operations, and Lambda functions

## Task 1: Preparation

A Dataproc cluster has been prepared for you. If you login to GCP before the progress bar reports that the "Lab is Running", you may have to wait several minutes for the cluster to transition from "Provisioning" to "Running" before the cluster completes setup.

You will be performing most of the lab steps from the Master Node of the cluster in an SSH terminal window.

1. In the Console, on the **Navigation menu** (7a91d354499ac9f1.png), click **Dataproc** > **Clusters**.
2. Locate the cluster named **dataproc-cluster**. Which region and zone is it located in? The region and zone have been selected automatically for you by Qwiklabs.
3. Notice the Cloud Storage staging bucket defined for this cluster. This bucket has the same name as the project ID, which is a convenient way to make the name globally unique.
4. Click on the name **dataproc-cluster** to go to the Cluster details page.
5. The Cluster details page opens to the **Monitoring** tab. Click on the tab labeled "VM Instances".
6. On the line for the VM named **dataproc-cluster-m** you will see that it has the Role of Master and there is an SSH link next to it. Click on **SSH** to open a terminal window to the Master Node.

## Task 2. Enable secure web access to the Dataproc cluster

### **Create a restrictive firewall rule using Target tags, IP address, and protocol**

Create a firewall rule that allows access only to the Master Node from your computer's IP address. Only ports 8088 (Hadoop Job Interface) and 9870 (Hadoop Admin interface) will be permitted.

### **Verify that the network tag is set on the Master Node**

Verify that the network tag "hadoopaccess" is set on the Master Node. That will apply the firewall rule to the Master Node, giving your laptop access to it.

1. In the Console, on the **Navigation menu** (7a91d354499ac9f1.png), click **Compute Engine** > **VM Instances**.
2. Click on the Master Node, **dataproc-cluster-m**.
3. Verify that under Network Tags it lists **hadoopaccess**.
4. If the tag is not there, click **EDIT**.
5. Under Network Tags add the tag: **hadoopaccess**
6. Click **Save**.

### **Identify the browser IP address**

You will use the browser IP address to allow your local browser to connect to the Dataproc cluster.

1. Find your computer's browser IP address by opening a browser window and viewing <http://ip4.me/> Copy the IP address.

### **Create the firewall rule**

1. In the Console, on the **Navigation menu** (7a91d354499ac9f1.png), click **VPC Network** > **Firewall rules**.
2. Click **Create Firewall Rule**.
3. Specify the following, and leave the remaining settings as their defaults:

|  |  |
| --- | --- |
| **Property** | **Value**  (type value or select option as specified) |
| **Name** | allow-hadoop |
| **Network** | default |
| **Priority** | 1000 |
| **Direction of traffic** | Ingress |
| **Action on match** | Allow |
| **Targets** | Specified target tags |
| **Target tags** | hadoopaccess |
| **Source IP ranges** | <your-IP>/32 |
| **Specified protocols and ports** | Check tcp and enter port number 9870,8088 |

1. Click **Create**.

It will take a few minutes for the firewall rule to become active.

## Task 3. Prepare the data

### **Copy sample files to the Master node home directory**

The sample files you need are have already been archived on the Master Node. You will need to copy them into your user directory with the following command.

1. In the Master Node SSH terminal window.

cd

cp -r /training .

ls

1. In the Master Node SSH terminal window. You should now have the **/training** directory in your home directory. And it should have files within it.

## Task 4. Explore HDFS and Cloud Storage

### **Use the Master Node HDFS client**

Dataproc requires a Cloud Storage bucket to stage its own files during initialization. Additionally, you can use this bucket to communicate with the cluster, for example, staging PySpark application programs or data in the bucket.

1. In the Console, on the **Navigation menu** (7a91d354499ac9f1.png), click **Compute Engine** > **VM Instances.**
2. On the line for the Master node, **dataproc-cluster-m**, click **SSH**.
3. In the **dataproc-cluster-m** terminal window, enter the following:

hadoop fs -ls /

Example output:

18/09/10 13:45:53 INFO gcs.GoogleHadoopFileSystemBase: GHFS version: 1.6.8-hadoop2

Found 3 items

drwxrwxrwt - mapred hadoop 0 2018-09-10 11:51 /hadoop

drwxrwxrwt - hdfs hadoop 0 2018-09-10 11:52 /tmp

drwxrwxrwt - hdfs hadoop 0 2018-09-10 11:51 /user

1. There are two sample text files in the **/training** folder. The file **road-not-taken.txt** is a Robert Frost poem of only 24 lines. The file **sherlock-holmes.txt** contains the collected works of Sherlock Holmes by Arthur Conan Doyle. (The first 277 lines of text were moved to this file to facilitate data processing: gs://cloud-training/gcpdei/sherlock-copyleft.txt)

cd ~/training

ls

1. Import the data files into HDFS in the cluster.

hadoop fs -mkdir /sampledata

hadoop fs -copyFromLocal road-not-taken.txt /sampledata/.

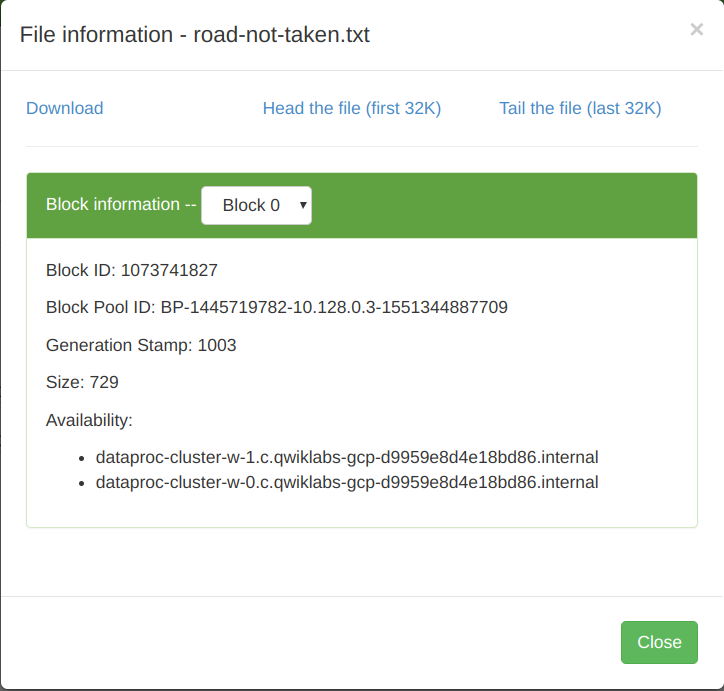
hadoop fs -copyFromLocal sherlock-holmes.txt /sampledata/.

1. Verify that the files exist in HDFS.

hadoop fs -ls /sampledata

1. Return to the browser for the Hadoop Administration Interface, or open a new window and browse to **<External\_IP>:9870**.
2. Under **Utilities**, select **Browse the file system**.
3. In the **Name** column click on **sampledata** (If you don't see sampledata, refresh the page).
4. In the **Name** column click on the **road-not-taken.txt**.

A pop-up window similar to the following should display.



This shows that the file fits into a single HDFS block. Notice from the **Block Information** pulldown, that the file is only located in **Block 0**. And that the block is duplicated on both worker node 0 and worker node 1.

1. Click **Close**.
2. Close the Hadoop Administration Interface window or tab.

## Task 5. Use interactive PySpark to learn about RDDs and Lambda functions

The interactive version of PySpark provides an excellent environment for learning Spark semantics and how to work with RDDs, transformations, and actions. In this section, you will be working with the copy of the sample files stored on the cluster in HDFS.

PySpark is a Read-Evaluate-Print-Loop (REPL) interpreter. Also known as a language shell. The REPL reads a single expression from the user, evaluates it, and prints the result. Then it loops and performs another single expression. The single-step immediate feedback of a REPL makes it useful for exploring and learning a language or system. The limitation is that the REPL holds no state context of its own, unlike more sophisticated shells.

<https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop>

1. In the Console, on the **Navigation menu** (7a91d354499ac9f1.png), click **Compute Engine** > **VM Instances.**
2. On the line for the Master node, **dataproc-cluster-m**, click **SSH**.
3. In the **dataproc-cluster-m** terminal window.
4. Start the interactive session by entering the following:

pyspark

It will take a few moments to initialize. When it is ready you will see something like this:

Python 2.7.13 (default, Sep 26 2018, 18:42:22)

[GCC 6.3.0 20170516] on linux2

Type "help", "copyright", "credits" or "license" for more information.

Setting default log level to "WARN".

To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).

ivysettings.xml file not found in HIVE\_HOME or HIVE\_CONF\_DIR,/etc/hive/conf.dist/ivysettings.xml will be used

Welcome to

\_\_\_\_ \_\_

/ \_\_/\_\_ \_\_\_ \_\_\_\_\_/ /\_\_

\_\ \/ \_ \/ \_ `/ \_\_/ '\_/

/\_\_ / .\_\_/\\_,\_/\_/ /\_/\\_\ version 2.3.2

/\_/

Using Python version 2.7.13 (default, Sept 26 2018 18:42:22)

SparkSession available as 'spark'.

1. Create a Spark RDD (Resilient Distributed Dataset) by reading the text file from HDFS. Use the python type() function to identify the object type. And use a built-in method for the object to count the number of lines.

A Resilient Distributed Dataset (RDD) is an abstraction over data in storage. The RDD is opaque to the location and replication of data it contains. For reliability, RDDs are resilient (fault-tolerant) to data loss due to node failures. An RDD lineage graph is used to recompute damaged or missing partitions. And for efficiency, Spark might choose to process one part in one location or another, based on availability of CPU at that location, or based on network latency or proximity to other resources.

The benefit of this abstraction is that it enables operations on an RDD to treat the RDD as a single object and ignore the complexity of how data is located, replicated, and migrated inside the RDD. All those details are left to Spark.

Enter:

lines = sc.textFile("/sampledata/sherlock-holmes.txt")

type(lines)

lines.count()

lines.take(15)

Example:

>>> lines = sc.textFile("/sampledata/sherlock-holmes.txt")

>>> type(lines)

<class 'pyspark.rdd.RDD'>

>>> lines.count()

12652

>>> lines.take(15)

[u'', u'THE ADVENTURES OF SHERLOCK HOLMES by ARTHUR CONAN DOYLE', u'', u'', u'', u'', u'A Scandal in Bohemia', u'Th

e Red-headed League', u'A Case of Identity', u'The Boscombe Valley Mystery', u'The Five Orange Pips', u'The Man wit

h the Twisted Lip', u'The Adventure of the Blue Carbuncle', u'The Adventure of the Speckled Band', u"The Adventure

of the Engineer's Thumb"]

>>>

1. Use a flatMap() spark transformation to split the sentences into words.

Enter:

words = lines.flatMap(lambda x: x.split(' '))

type(words)

words.count()

words.take(15)

Example:

>>> words = lines.flatMap(lambda x: x.split(' '))

>>> type(words)

<class 'pyspark.rdd.PipelinedRDD'>

>>> words.count()

107265

>>> words.take(15)

[u'', u'THE', u'ADVENTURES', u'OF', u'SHERLOCK', u'HOLMES', u'by', u'ARTHUR', u'CONAN', u'DOYLE', u'', u'', u'', u'

', u'A']

>>>

1. Use a spark map() transformation to create pairs. The first element in the pair is the word. The second element of the pair is the number of characters (length) of the word.

Enter:

pairs = words.map(lambda x: (x,len(x)))

type(pairs)

pairs.count()

pairs.take(5)

Example:

>>> pairs = words.map(lambda x: (x,len(x)))

>>> type(pairs)

<class 'pyspark.rdd.PipelinedRDD'>

>>> pairs.count()

107265

>>> pairs.take(5)

[(u'', 0), (u'THE', 3), (u'ADVENTURES', 10), (u'OF', 2), (u'SHERLOCK', 8)]

>>>

1. The objective is to count the number of words of various lengths -- so how many words with 10 characters are used in the books. Modify the map() so that it creates pairs of word length, and '1' for each instance. This is a common "column-oriented" method for counting instances in a parallelizable way.

Enter:

pairs = words.map(lambda x: (len(x),1))

pairs.take(5)

Example:

>>> pairs = words.map(lambda x: (len(x),1))

>>> pairs.take(5)

[(0, 1), (3, 1), (10, 1), (2, 1), (8, 1)]

>>>

1. Use a parallelizable method (add) to accumulate the instances. The add function will be called inside the Spark reduceByKey() transformation.

Enter:

from operator import add

wordsize = pairs.reduceByKey(add)

type(wordsize)

wordsize.count()

wordsize.take(5)

Example:

>>> from operator import add

>>> wordsize = pairs.reduceByKey(add)

>>> type(wordsize)

<class 'pyspark.rdd.PipelinedRDD'>

>>> wordsize.count()

22

>>> wordsize.take(5)

[(0, 2756), (2, 18052), (4, 19456), (6, 8622), (8, 4664)]

>>>

1. Convert the RDD into a Python object for easy output.

Enter:

output = wordsize.collect()

type(output)

for (size,count) in output: print(size, count)

Example:

>>> output = wordsize.collect()

>>> type(output)

<type 'list'>

>>> for (size,count) in output: print(size, count)

...

(0, 2756)

(2, 18052)

(4, 19456)

(6, 8622)

(8, 4664)

(10, 1730)

(12, 585)

(14, 159)

(16, 31)

(18, 8)

(20, 4)

(1, 5141)

(3, 22939)

(5, 12044)

(7, 6615)

(9, 2980)

(11, 1035)

(13, 352)

(15, 75)

(17, 12)

(19, 4)

(21, 1)

1. What happened? Why is the list out of order? It is because the collect() action was parallelized, and then the results were assembled. Use the Spark sortByKey() method to sort the pairs before collecting them into a list.

Enter:

output = wordsize.sortByKey().collect()

for (size,count) in output: print(size, count)

Example:

>>> output = wordsize.sortByKey().collect()

>>> for (size, count) in output: print(size, count)

...

(0, 2756)

(1, 5141)

(2, 18052)

(3, 22939)

(4, 19456)

(5, 12044)

(6, 8622)

(7, 6615)

(8, 4664)

(9, 2980)

(10, 1730)

(11, 1035)

(12, 585)

(13, 352)

(14, 159)

(15, 75)

(16, 31)

(17, 12)

(18, 8)

(19, 4)

(20, 4)

(21, 1)

>>>

1. The following program recreates exactly what you did in the previous steps. At each step you executed one Spark transformation on one RDD which returns results in a separate RDD.

Spark doesn't perform operations immediately. It uses an approach called "lazy evaluation".

There are two classes of operations: transformations and actions. A transformation takes one RDD as input and generates another RDD as output. You can think of a transformation as a request. It explains to Spark what you want done, but doesn't tell Spark how to do it.

Actions like "collect", "count", or "take" produce a result, such as a number or a list.

When Spark receives transformations, it stores them in a DAG (Directed Acyclic Graph) but doesn't perform them at that time. When Spark receives an action it examines the transformations in the DAG and the available resources (number of workers in the cluster), and creates pipelines to efficiently perform the work.

1. In the previous steps, you issued an action after each transformation to see the intermediate results. In the program below, the only action is "collect" at the end.

from operator import add

lines = sc.textFile("/sampledata/sherlock-holmes.txt")

words = lines.flatMap(lambda x: x.split(' '))

pairs = words.map(lambda x: (len(x),1))

wordsize = pairs.reduceByKey(add)

output = wordsize.sortByKey().collect()

1. To allow Spark to perform its magic, the program needs to build a chain of transformations using the dot operator. When it is passed to Spark in this way, Spark understands the multiple steps and that the results of one transformation are to be passed to the next transformation. This allows Spark to organize the processing in any way it decides based on the resources available in the cluster. So, while the above program is more readable, with "words", "pairs", and "wordsize" called out, the following program does the same thing without naming the intermediate results. They are functionally identical.

output2 = lines.flatMap(lambda x: x.split(' ')).map(lambda x: (len(x),1)).reduceByKey(add).sortByKey().collect()

for (size, count) in output2: print(size, count)

1. Enter **exit()** to quit PySpark. Leave the Master Node SSH terminal open. You will use it in the next task.

## Task 6: Submit a PySpark job from the Master Node

1. In the SSH terminal for the Master Node, use **nano** or **vi** to create the file **wordcount.py**.

**Note:** Before creating **wordcount.py**, confirm that you are in **training** directory. If not, run cd ~/training.

1. Copy and paste the following code into the file.

from pyspark.sql import SparkSession

from operator import add

import re

print("Okay Google.")

spark = SparkSession\

.builder\

.appName("CountUniqueWords")\

.getOrCreate()

lines = spark.read.text("/sampledata/road-not-taken.txt").rdd.map(lambda x: x[0])

counts = lines.flatMap(lambda x: x.split(' ')) \

.filter(lambda x: re.sub('[^a-zA-Z]+', '', x)) \

.filter(lambda x: len(x)>1 ) \

.map(lambda x: x.upper()) \

.map(lambda x: (x, 1)) \

.reduceByKey(add) \

.sortByKey()

output = counts.collect()

for (word, count) in output:

print("%s = %i" % (word, count))

spark.stop()

1. Run the job.

spark-submit wordcount.py

Example output

Okay Google.

"'ABOUT = 1

"'ABSOLUTE = 1

"'AH!' = 2

"'AH, = 2

"'AMPLE.' = 1

"'AND = 10

"'ARTHUR!' = 1

"'AS = 1

"'AT = 1

"'BECAUSE = 1

"'BRECKINRIDGE, = 1

### **View the job in the Hadoop JobTracker window**

1. While the application is running, return to the Hadoop JobTracker window. Examine the following pages in the Hadoop Applications interface at <External IP>:8088. You can find the Master node External IP in the Console, on the **Navigation menu** (7a91d354499ac9f1.png), click **Compute Engine** > **VM Instances**.
2. Explore these links in the interface. They map to the job lifecycle. SUBMITTED, ACCEPTED, RUNNING, FINISHED, FAILED, KILLED

### **Look for the job on the Dataproc Jobs page**

1. In the Console, on the **Navigation menu** (7a91d354499ac9f1.png), click **Dataproc** > **Jobs**. Why are there no jobs listed?

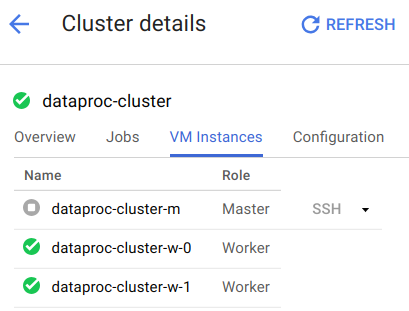
The Datalab applications that you ran previously do not show up in the Dataproc > Jobs page in Console. Only applications submitted from Console are tracked in Console.

## Task 7: Master Node timeout

The Master Node is configured with an automatic timeout feature. If no jobs are submitted for an extended period of time the Master Node will STOP to reduce costs. The other nodes continue to operate as needed to maintain HDFS. You can check for this state on the Cluster Details page. To restore the ability to submit jobs, you will need to restart the Master Node.

1. Check if the Master Node has timed out.
2. In the Console, on the **Navigation menu** (7a91d354499ac9f1.png), click **Dataproc** > **Clusters**.
3. Click on your cluster, **dataproc-cluster**. Then click on the **VM Instances** tab.

Example:



1. If the Master Node has timed out click on its name, **dataproc-cluster-m**.
2. At the top of the **VM instance details** page, click **START**.