Spark and PySpark

What Is Spark?

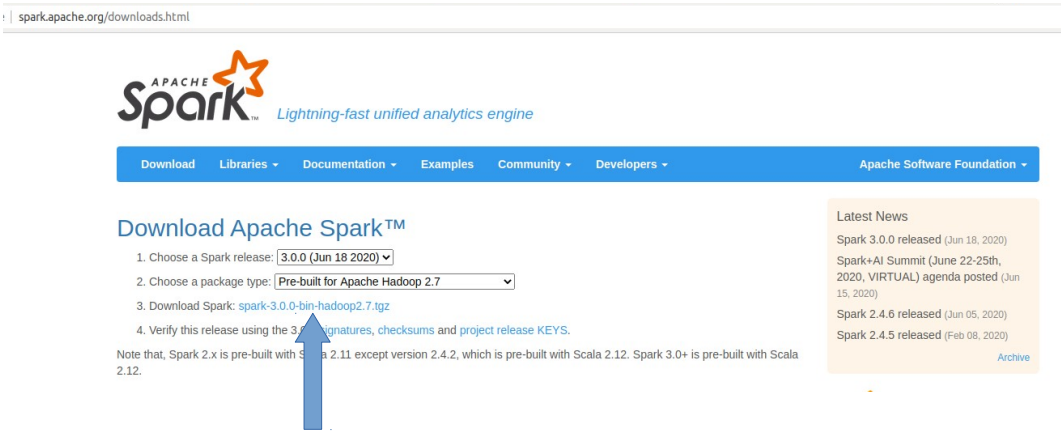
- Spark is an-open source computational framework that, like MapReduce, processes and Analyses huge amounts of data using commodity servers.
- Spark's API lets developers create distributed applications that make use of a cluster's resources without having to know all the low-level details about how to allocate the cluster's resources among the various applications.

What Is PySpark?

- PySpark is Spark's Python API.
- PySpark allows Spark applications to be created from an interactive shell or from Python programs.

Installing Spark

Step 1: Download the file from:
https://spark.apache.org/downloads.html



Step 2: unzip the file from terminal

faculty@glsica:~$ tar -zxvf spark-3.0.0-bin-hadoop2.7.tgz

Step 3: Rename the folder to spark_3.0

Step 4: In terminal open .bashrc file

faculty@glsica:~$ gedit ~/.bashrc

Step 5: Add this following code at the end of .bashrc file

```
#Spark Settings
export SPARK_HOME=/home/faculty/spark_3.0
export PATH=$PATH:$SPARK_HOME/bin
export PYSPARK_PYTHON=python3
export PATH=$PATH:$JAVA_HOME/jre/bin
```

Step 6: Execute the .bashrc file
faculty@glsica:~$ source ~/.bashrc

Step 7: Execute the following commands

faculty@glsica:~$ cd $SPARK_HOME

faculty@glsica:~/spark_3.0$ cd bin

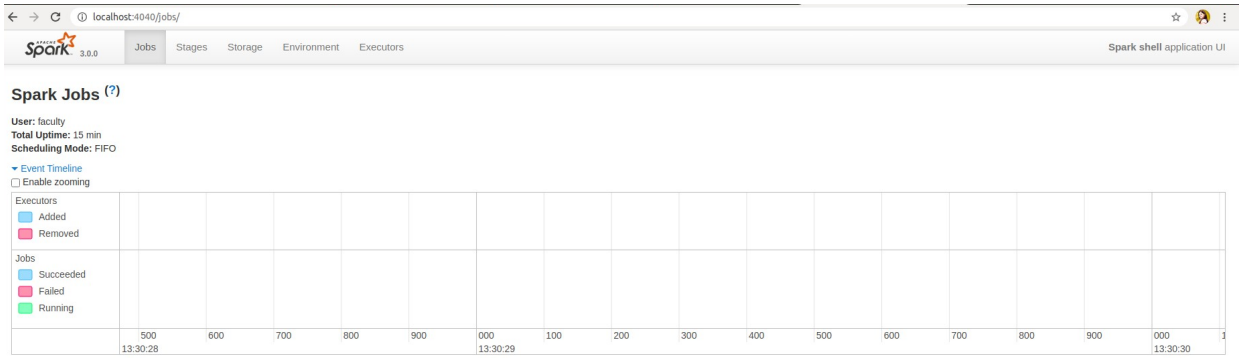
faculty@glsica:~/spark_3.0/bin$ spark-shell --version

```
faculty@glsica:~/spark_3.0/bin$ spark-shell --version
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.0.0
      /_/

Using Scala version 2.12.10, OpenJDK 64-Bit Server VM, 1.8.0_242
Branch HEAD
Compiled by user ubuntu on 2020-06-06T11:32:25Z
Revision 3fdfce3120f307147244e5eaf46d61419a723d50
Url https://gitbox.apache.org/repos/asf/spark.git
Type --help for more information.
```

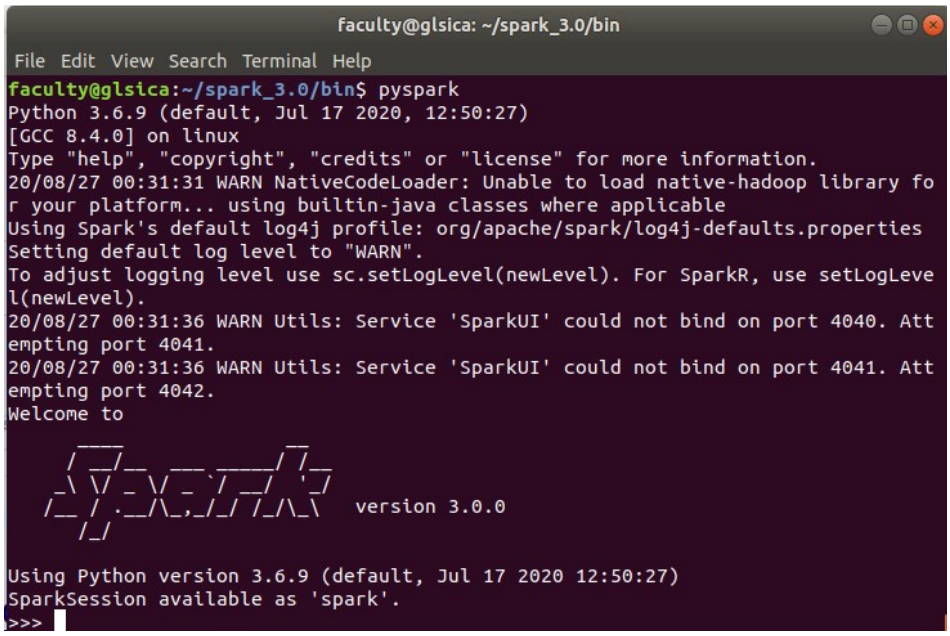
Spark UI

In browser type http://localhost:4040



Interacting with PySpark Shell

faculty@glsica:~/spark_3.0/bin$ pypsark or ./pyspark



Working with PySpark Shell

Resilient Distributed Datasets

- Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark.
- Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster.
- RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.
- Formally, an RDD is a read-only, partitioned collection of records.
- RDD is a fault-tolerant collection of elements that can be operated on in parallel.
- RDDs can be constructed in multiple ways: **by parallelizing existing Python collections, by referencing files in an external storage system such as HDFS, or by applying transformations to existing RDDs.**

Self-contained applications must first create a SparkContext object before using any Spark methods. The master can be set when the SparkContext() method is called:

For Python Programs

sc = SparkContext(master='local[4]')

For pyspark shell

faculty@glsica:~/spark_3.0/bin$ pyspark --master local[4]

The option --master local[4] means the spark context of this spark shell acts as a **master** on **local** node with **4** threads.

Note: Using PySpark Shell there is no need to create SparkContext() object. It will be automatically created. You just need to use the sc object.

Creating RDDs from Python Collections

- RDDs can be created from a Python collection by calling the Spark Context.parallelize() method
- The RDD.glom() method returns a list of all of the elements within each partition.
- RDD.collect() method brings all the elements to the driver node.

Creating a List and creating a RDD

```
>>> data = [1,2,3,4,5]
>>> rdd = sc.parallelize(data)
>>> rdd.glom().collect()
```

[[1], [2], [3], [4, 5]]

Creating RDD with Specified number of partitions

```
>>> rdd = sc.parallelize(data,3)
>>> rdd.glom().collect()
[[1], [2, 3], [4, 5]]
```

Operations on RDD

1) Count number of items in RDD with **count()**

```
>>> rdd.count()
5
```

2) Read the first item from RDD with **first()**

```
>>> rdd.first()
1
```

3) Read the first 3 items from RDD with **take()**

```
>>> rdd.take(3)
[1, 2, 3]
```

Creating a list of words and creating RDD

```
>>> word_list = ["hello this","is spark", "session"]
>>> rdd = sc.parallelize(word_list)
>>> rdd.glom().collect()
[[], ['hello this'], ['is spark'], ['session']]
```

Creating RDDs from External Sources

```
>>> inpfile = sc.textFile("/home/faculty/employee.txt")
>>> inpfile.glom().collect()
[['1 Ajay Programmer 20000', '2 Poonam Data_Scientest 40000', '3 Komal
Programmer 30000'], ['4 Kinjal Bussiness_Analyst 70000', '5 Chintan
Team_Lead 60000 ']]
```

Creating RDD with specified no of parameters

```
>>> inpfile = sc.textFile("/home/faculty/employee.txt",3)
>>> inpfile.glom().collect()
[['1 Ajay Programmer 20000', '2 Poonam Data_Scientest 40000'], ['3 Komal
Programmer 30000', '4 Kinjal Bussiness_Analyst 70000'], ['5 Chintan
Team_Lead 60000 ']]
```

Creating RDD with file from HDFS

```
>>> hdfsfile = sc.textFile("hdfs://localhost:9000/user/file.txt")
>>> hdfsfile.glom().collect()
[['hello hi ', 'how are you all', 'hi welcome to the'], ['datascience practicals ',
'session']]
```

Contents of file.txt

```
hello hi
how are you all
hi welcome to the
datascience practicals
session
```

RDD Operations

- RDDs support two types of operations: transformations and actions.
- Transformations create new datasets from existing ones, and actions run a computation on the dataset and return results to the driver program.
- Transformations are computed when an action requires a result to be returned to the driver program.
- This allows Spark to operate efficiently and only transfer the results of the transformations before an action.
- Spark is lazy, so nothing will be executed unless you call some transformation or action that will trigger job creation and execution

Python Lambda Functions

- Many of Spark's transformations and actions require function objects to be passed from the driver program to run on the cluster.
- The easiest way to define and pass a function is through the use of Python lambda functions.
- Lambda functions are anonymous functions that are created at runtime.
- They can be used wherever function objects are required and are syntactically restricted to a single expression.
- Lambdas are defined by the keyword lambda, followed by a comma separated list of arguments. A colon separates the function declaration from the function expression. The function expression is a single expression that produces a result for the provided arguments.

  e.g: lambda a, b: a + b

- This lambda has one argument and returns the length of the argument.

  e.g: lambda x: len(x)

RDD Workflow

The general workflow for working with RDDs is as follows:

1. Create an RDD from a data source.
2. Apply transformations to an RDD.
3. Apply actions to an RDD.

RDD Transformations

- Transformations create new datasets from existing ones.
- Lazy evaluation of transformation allows Spark to remember the set of transformations applied to the base RDD.

- This enables Spark to optimize the required calculations.

1) map
The **map(func)** function returns a new RDD by applying a function to each element of the source.

The following example performs operations on Python Collections computes operation of adding nos by 2 to the whole list and returns a new RDD.

```
>>> data=[1,2,3,4,5]
>>> rdd=sc.parallelize(data)
>>> mapdata=rdd.map(lambda x: x+2)
>>> mapdata.collect()
[3, 4, 5, 6, 7]
```

The following example loads file from HDFS and returns each line:

```
>>> hdfsfile = sc.textFile("hdfs://localhost:9000/user/file.txt")
>>> hdfsfile.glom().collect()

>>>filemap = hdfsfile.map(lambda line: line.split(" "))
>>> filemap.collect()
[['hello', 'hi'], ['how', 'are', 'you', 'all'], ['hi', 'welcome', 'to', 'the'], ['datascience', 'practicals'], ['session']]
```

2) FlatMap
With the help of **flatMap()** function, to each input element, we have many elements in an output RDD. The most simple use of flatMap() is to split each input string into words.
Map and flatMap are similar in the way that they take a line from input RDD and apply a function on that line. The key difference between map() and flatMap() is map() returns only one element, while flatMap() can return a list of elements.

```
>>>hdfsfile = sc.textFile("hdfs://localhost:9000/user/file.txt")
>>> filemap = hdfsfile.flatMap(lambda line: line.split(" "))
>>> filemap.collect()
['hello', 'hi', '', 'how', 'are', 'you', 'all', 'hi', 'welcome', 'to', 'the', 'datascience', 'practicals', '', 'session']
```

Difference between map() and flatMap()

```
>>> filemap = hdfsfile.map(lambda line: line.split(" "))
>>> filemap.collect()
[['hello', 'hi', ''], ['how', 'are', 'you', 'all'], ['hi', 'welcome', 'to', 'the'], ['datascience', 'practicals', ''], ['session']]
```

Map function returned each element consisting of a line.

```
>>> filemap = hdfsfile.flatMap(lambda line: line.split(" "))
>>> filemap.collect()
['hello', 'hi', '', 'how', 'are', 'you', 'all', 'hi', 'welcome', 'to', 'the', 'datascience', 'practicals', '', 'session']
```

FlatMap function return element consisting of each word from each line.

3) distinct

The **distinct()** method returns a new RDD containing only the distinct elements from the source RDD.

The following example returns the distinct elements from the RDD.

```
>>>hdfsfile = sc.textFile("hdfs://localhost:9000/user/file.txt")
>>> filemap = hdfsfile.flatMap(lambda line: line.split(" "))
>>> distinct_res = filemap.distinct()
>>> distinct_res.collect()
['are', 'practicals', 'hello', 'hi', 'how', 'you', 'all', 'welcome', 'to', 'the', 'datascience', 'session']
```

4) filter

The **filter(func)** function returns a new RDD containing only the elements of the source that the supplied function returns as true.

The following example filters the word hi and display its count from the generated RDD:

```
>>>hdfsfile = sc.textFile("hdfs://localhost:9000/user/file.txt")
>>> filemap = hdfsfile.flatMap(lambda line: line.split(" ")).filter(lambda value: value == "hi")
>>> filemap.collect()
['hi', 'hi']
```

5) reduceByKey(func, [num of Tasks])

When we use **reduceByKey** on a dataset (K, V), the pairs on the same machine with the same key are combined, before the data is shuffled.

```
>>> words = ["hi","hello","hi","welcome"]
>>> rdd=sc.parallelize(words)
>>> key_pair=rdd.map(lambda word: (word,1))
>>> key_pair.collect()
[('hi', 1), ('hello', 1), ('hi', 1), ('welcome', 1)]
>>>final_output=key_pair.reduceByKey(lambda key,pair:key+pair)
>>> final_output.collect()
[('hi', 2), ('hello', 1), ('welcome', 1)]
```

`

RDD Actions

1) reduce
The reduce() method aggregates elements in an RDD using a function, which takes two arguments and returns one. The function used in the reduce method is commutative and associative ensuring that it can be correctly computed in parallel. We can add the elements of RDD, count the number of words.

```
>>> data=[1,2,3,4,5,6]
>>> rdd = sc.parallelize(data)
>>> rdd.reduce(lambda a,b: a+b)
21
```

2)take
The take(n) method returns an array with the first n elements of the RDD. In the following example it returns first 5 elements from RDD.

```
>>>hdfsfile = sc.textFile("hdfs://localhost:9000/user/file.txt")
>>> filemap = hdfsfile.flatMap(lambda line: line.split(" "))
>>> filemap.take(5)
['hello', 'hi', '', 'how', 'are']
```

3)collect
The collect() method returns all of the elements of the RDD as an array. The following example returns all of the elements
from an RDD:

```
>>hdfsfile = sc.textFile("hdfs://localhost:9000/user/file.txt")
>>> filemap = hdfsfile.flatMap(lambda line: line.split(" "))
>>> filemap.collect()
['hello', 'hi', '', 'how', 'are', 'you', 'all', 'hi', 'welcome', 'to', 'the', 'datascience',
'practicals', '', 'session']
```

4)takeOrdered
The takeOrdered(n, key=func) method returns the first n elements of the RDD, in their natural order, or as specified by the function func. The following example returns the first four elements of the RDD in descending order:

```
>>> data = [6,7,3,5,4,8,9]
>>> rdd = sc.parallelize(data)
>>> rdd.takeOrdered(4,lambda s: -s)
[9, 8, 7, 6]
```

Word Count program in Pyspark Shell

```
>>> hdfsfile = sc.textFile("hdfs://localhost:9000/user/file.txt")
>>> filemap = hdfsfile.flatMap(lambda line: line.split(" "))
>>> key_pair = filemap.map(lambda word: (word, 1))
>>> final_count = key_pair.reduceByKey(lambda key,pair:key+pair)
>>> final_count.collect()
[('', 2), ('are', 1), ('practicals', 1), ('hello', 1), ('hi', 2), ('how', 1), ('you', 1), ('all',
1), ('welcome', 1), ('to', 1), ('the', 1), ('datascience', 1), ('session', 1)]
```

Working with PySpark Python Programs

Note: Take input of the file from local drive and save the output in local drive.

spark_wordcount.py (taking input and storing the output in local drive)

```
from pyspark import SparkContext
def main():
        sc = SparkContext(appName='SparkWordCount')
        hdfsfile = sc.textFile("/home/faculty/file.txt")
        filemap = hdfsfile.flatMap(lambda line: line.split(" "))
        key_pair = filemap.map(lambda word: (word, 1))
        final_count = key_pair.reduceByKey(lambda key,pair:key+pair)
        final_count.saveAsTextFile("/home/faculty/sparkoutput1")
        sc.stop()

if __name__ == '__main__':
        main()
```

Run the program

faculty@glsica:~/hadoop-3.1.3$ spark-submit --master local spark_wordcount.py

View the output

1) Go to the /home folder.

2) A folder named sparkoutput1 will be created.

3) Go inside folder and double click the part-00000 file to view output

Word Count program (Spark_wordcount.py)

Taking input and storing the output on HDFS

```
from pyspark import SparkContext
def main():
        sc = SparkContext(appName='SparkWordCount')
        hdfsfile = sc.textFile("hdfs://localhost:9000/user/file.txt")
        filemap = hdfsfile.flatMap(lambda line: line.split(" "))
        key_pair = filemap.map(lambda word: (word, 1))
        final_count = key_pair.reduceByKey(lambda key,pair:key+pair)
        final_count.saveAsTextFile("hdfs://localhost:9000/sparkoutput")
        sc.stop()

if __name__ == '__main__':
        main()
```

Executing the file on hadoop:

- To execute the Spark application, pass the name of the file to the spark-submit script.
- The master property is a cluster URL that determines where the Spark appliction will run.
- Values of master property:

local	Run Spark with one worker thread.
local[*n*]	Run Spark with *n* worker threads.
spark://HOST:PORT	Connect to a Spark standalone cluster.
mesos://HOST:PORT	Connect to a Mesos cluster.

faculty@glsica:~/hadoop-3.1.3$ **spark-submit --master local spark_wordcount.py**

Viewing output:

faculty@glsica:~/hadoop-3.1.3$ **hdfs dfs -cat /sparkoutput/part-00000**
2020-08-27 00:01:11,961 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
('hello', 1)
('hi', 2)
('', 2)
('how', 1)
('are', 1)
('you', 1)
('all', 1)
('welcome', 1)
('to', 1)
('the', 1)
('datascience', 1)
('practicals', 1)
('session', 1)

Reference:
https://data-flair.training/blogs/install-spark-ubuntu/
https://www.tutorialspoint.com/apache_spark/apache_spark_quick_guide.htm