What am I going to learn from this PySpark Tutorial?

This spark and python tutorial will help you understand how to use Python API bindings i.e. PySpark shell with Apache Spark for various analysis tasks.At the end of the PySpark tutorial, you will learn to use spark python together to perform basic data analysis operations.

Attractions of the PySpark Tutorial

- Basic Interaction with Spark Shell using Python API- PySpark
- Spark Resilient Distributed Datasets (Spark RDD's)
- Transformation and Actions in Spark
- Caching, Accumulators and UDF's

Software Prerequisites

- Apache Spark (Downloadable from http://spark.apache.org/downloads.html)
- Python Installed

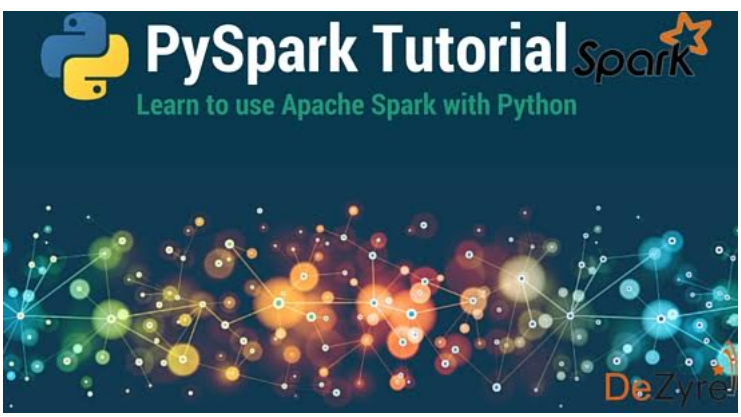
Python is a powerful programming language for handling complex data analysis and data munging tasks. It has several in-built libraries and frameworks to do data mining tasks efficiently. However, no programming language alone can handle big data processing efficiently. There is always need for a distributed computing framework like Hadoop or Spark.

Apache Spark supports three most powerful programming languages:

1. Scala
2. Java
3. Python

Taming Big Data with Apache Spark and Python

Apache Spark is written in Scala programming language that compiles the program code into byte code for the JVM for spark big data processing. The open source community has developed a wonderful utility for spark python big data processing known as PySpark. PySpark helps data scientists interface with Resilient Distributed Datasets in apache spark and python.Py4J is a popularly library integrated within PySpark that lets python interface dynamically with JVM objects (RDD's).



known as "Pyspark". To use Pyspark you will have to have python installed on your machine. As we know that each Linux machine comes preinstalled with python so you need not worry about python installation. To get started in a standalone mode you can download the pre-built version of spark from its official home page listed in the pre-requisites section of the PySpark tutorial. Decompress the downloaded file. On decompressing the spark downloadable, you will see the following structure:

bin
Holds all the binaries

conf
Holds all the necessary configuration files to run any spark application

ec2
Holds the scripts to launch a cluster on amazon cloud space with multiple ec2 instances

lib
Holds the prebuilt libraries which make up the spark APIS

licenses

python

python API

README.md

Holds important instructions to get started with spark

Holds important startup scripts that are required to setup distributed cluster

CHANGES.txt

Holds all the changes information for each version of apache spark

data

Holds data that is used in the examples

examples

Has examples which are a good place to learn the usage of spark functions.

LICENSE NOTICE

Important information

R

Holds API of R language

RELEASE

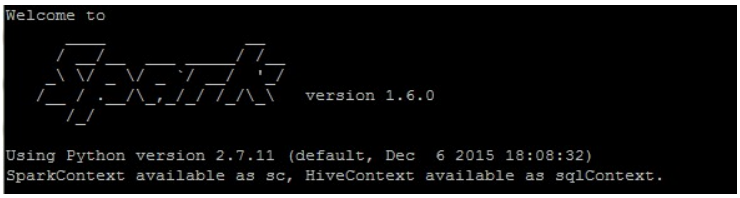
Holds make info of the downloaded version

Basic Interaction with PySpark shell

Spark comes with an interactive python shell. The PySpark shell is responsible for linking the python API to the spark core and initializing the spark context.

bin/PySpark command will launch the Python interpreter to run PySpark application. PySpark can be launched directly from the command line for interactive use.

You will get python shell with following screen:



Spark Context allows the users to handle the managed spark cluster resources so that users can read, tune and configure the spark cluster. Spark Content is used to initialize the driver program but since PySpark has Spark Context available as sc, PySpark itself acts as the driver program.

Reading a file in PySpark Shell

Let's read a file in the interactive session .We will read "CHANGES.txt" file from the spark folder here.

```
RDDread = sc.textFile ("file:///opt/spark/CHANGES.txt")
```

The above line of code has read the file CHANGES.txt in a RDD named as "RDDread".

How does it look like? Let's see the contents of the RDD using the collect () action-

```
RDDread. Collect ()
```

// Too much of output

So much of text and it's loaded in just a matter of few seconds-that's the power of Apace Spark. This action is not at all recommended on a huge file as it would overload the driver memory with too much of text on the console. When performing collect action on a larger file the data is pulled from multiples nodes and there is a probability that the driver node could run out of memory.

To display the content of Spark RDD's there in an organized format, actions like "first ()","take ()", and "takeSample (False, 10, 2)" can be used.

First () – This will return the first element from the dataset.

Example -

```
RDDread. First ()
```

// The above reads the first line of the RDD i.e. the first line from changes.txt file is displayed.

Take (n) - This will return the first n lines from the dataset and display them on the console.

Example-

```
RDDread. Take (5)
```

```
In [4]: RDDread.take(5)
Out[4]: [u'Spark Change Log', u'----------------', u'', u'Release 1.6.0', u'']
```

//The above line of code reads first 5 lines of the RDD

TakeSample (withReplacement, n, [seed]) - This action will return n elements from the dataset, with or without replacement (true or false). Seed is an optional parameter that is used as a random generator.

Example -

```
RDDread. TakeSample (False, 10, 2)
```

```
In [5]: RDDread.takeSample(False,10,2)
Out[5]:
[u'  2015-02-07 19:41:30 +0000',
 u'  [SPARK-5585] Flaky test in MLlib python',
 u'  [SPARK-10576] [BUILD] Move .java files out of src/main/scala',
 u'',
 u'',
 u'  [SPARK-8866][SQL] use 1us precision for timestamp type',
 u'',
 u'  2015-11-07 19:44:45 -0800',
 u'  [SPARK-5078] Optionally read from SPARK_LOCAL_HOSTNAME',
 u'  Reynold Xin <rxin@databricks.com>']
```

//This reads random 10 lines from the RDD. The first parameter says the random sample has been picked with replacement. The last parameter is simply the seed for the sample.

Count () – To know the number of lines in a RDD

Example -

```
RDDread. Count ()
```

```
In [6]: RDDread.count()
Out[6]: 34172
```


Using External Database

Let's look at how we can connect MySQL database through spark driver.

Apache Spark can load data into any RDBMS that supports JDBC connectivity like Postgres and MySQL. External databases can be accessed in Apache Spark either through hadoop connectors or custom spark connectors. Unlike other data sources, when using JDBCRDD, ensure that the database is capable of handling the load of parallel reads from apache spark.

Let's create a table in MySQL and insert data into it.

Launch MySQL

MySQL –u root -pXXXX

```
create table demotable(id int,name varchar(20),age int);
```



```
use demo;

insert into demotable values(1,"abhay",25),(2,"banti",25);

Select * from demotable;
```

```
+------+-------+------+

| id   | name  | age  |

+------+-------+------+

|    1 | abhay |   25 |

|    2 | banti |   25 |

+------+-------+------+
2 rows in set (0.00 sec)
```

Let's download the MySQL jar which would have all the classes to connect to a MySQL database.

You can download it from http://www.java2s.com/Code/JarDownload/mysql/mysql.jar.zip and decompress the zip file.

Launch PySpark with the jar file in the class path as shown below -

```
PySpark --jars
```

SqlContext is available to the PySpark shell by default which is used to load the table as a data frame.

```
dataframe_mysql = sqlContext.read.format("jdbc").options( url="jdbc:mysql://:3306/demo",driver =
"com.mysql.jdbc.Driver",dbtable = "demotable",user="root", password="XXXXX").load()
dataframe_mysql.show()
```

```
+---+-----+-------+

| id| name|age|

+---+--------+---+

|  1|abhay| 25|

|  2|banti| 25|

+---+------+---+
```

We have successfully fetched data from MySql in our driver.

Let's do a simple operation using the world database (you can download the world database from **http://downloads.mysql.com/docs/world.sql.zip) .**

World database can be imported into MySQL using the name world.

Let's load the two popular tables-Country and Country Language into the spark shell.

Loading Country Table using sqlContext

```
Country.persist()
```


Loading CountryLanguage Table using sqlContext

```
CountryLanguage = sqlContext.read.format("jdbc").options(
url="jdbc:mysql://localhost:3306/world",driver = "com.mysql.jdbc.Driver",dbtable =
"CountryLanguage",user="root", password="XXXXXX").load().persist()
```

Let's check the column headers for the loaded data frames-

```
Country.columns
```

```
['Code',
 'Name',
 'Continent',
 'Region',
 'SurfaceArea',
 'IndepYear',
 'Population',
 'LifeExpectancy',
 'GNP',
 'GNPOld',
 'LocalName',
 'GovernmentForm',
'HeadOfState',
 'Capital',
 'Code2']
```

```
CountryLanguage.columns
```

```
['CountryCode', 'Language', 'IsOfficial', 'Percentage']
```

We now have two data frames with information about countries across the world.

Suppose we want to find out the language corresponding to each then we should combine these two tables with a common key Code.

Let's do the same operation in spark:

```
country_name = Country.map(lambda row:(row[0],row[1]))
```

```
Countryname_language = country_name.join(country_lang)
```

Here's how you can check the joined data on both the tables-

```
Countryname_language.take(10)
```

```
[(u'AGO', (u'Angola', u'Ambo')),
 (u'AGO', (u'Angola', u'Chokwe')),
 (u'AGO', (u'Angola', u'Kongo')),
 (u'AGO', (u'Angola', u'Luchazi')),
 (u'AGO', (u'Angola', u'Luimbe-nganguela')),
(u'AGO', (u'Angola', u'Luvale')),
 (u'AGO', (u'Angola', u'Mbundu')),
 (u'AGO', (u'Angola', u'Nyaneka-nkhumbi')),
 (u'AGO', (u'Angola', u'Ovimbundu')),
 (u'DZA', (u'Algeria', u'Arabic'))]
```

You can get the distinct number of records from the above output as shown below -

```
Countryname_language.distinct().count()
```

```
Out[56]: 984
```

Transformation and Actions in Apache Spark

Spark Transformations

- **map()**
- **flatMap()**
- **filter()**
- **sample()**
- **union()**
- **intersection()**
- **distinct()**
- **join()**

Spark Actions

- reduce()
- collect()
- count()
- first()
- takeSample(withReplacement, num, [seed])

map() and flatMap() Transformations in Spark

map()

map() transformation applies changes on each line of the RDD and returns the transformed RDD as iterable of iterables i.e. each line is equivalent to a iterable and the entire RDD is itself a list

flatMap()

This transformation apply changes to each line same as map but the return is not a iterable of iterables but it is only an iterable holding entire RDD contents.

Confused !!! Ok. Let's clear this confusion with an example ...

We have a file which defines confusion and the content looks like below :

> Confusion is the inability to think as clearly or quickly as you normally do.
>
> You may have difficulty paying attention to anything , remembering anyone, and making decisions.
>
> Confusion may come to anyone early or late phase of the life, depending on the reason behind it .
>
> Many times, confusion lasts for a very short span and goes away.
>
> Other times, it may be permanent and has no cure. It may have association with delirium or dementia.
>
> Confusion is more common in people who are in late stages of the life and often occurs when you have stayed in hospital.
>
> Some confused people may have strange or unusual behavior or may act aggressively.
>
> A good way to find out if anyone is confused is to question the person their identity i.e. name, age, and the date.
>
> If they are little not sure or unable to answer correctly, they are confused

Load this file into an RDD "confusedRDD"

```
confusedRDD = sc.textFile("confused.txt")
```

Check its contents, say first 5 lines- check each line is one entity and the entire object is itearable of entities i.e. a list of strings

```
confusedRDD.take(5)
```

```
In [8]: confusedRDD.take(5)
Out[8]:
[u'Confusion is the inability to think as clearly or quickly as you normally do. ',
 u'You may feel disoriented and have difficulty paying attention, remembering, and making decisions.',
 u'Confusion may come on quickly or slowly over time, depending on the cause. ',
 u'Many times, confusion lasts for a short time and goes away. ',
 u'Other times, it is permanent and not curable. It may be associated with delirium or dementia.']
```

Now let's divide each string on spaces and analyze the structure of confused.txt file -

```
mappedconfusion.take(5)
```



```
In [9]: mappedconfusion = confusedRDD.map(lambda line : line.split(" "))

In [10]: mappedconfusion.take(5)
Out[10]:
[[u'Confusion',
  u'is',
  u'the',
  u'inability',
  u'to',
  u'think',
  u'as',
  u'clearly',
  u'or',
  u'quickly',
  u'as',
  u'you',
  u'normally',
  u'do.',
  u''],
```

From the output it is evident that each line is a separate iterable of words which itself is contained in another iterable i.e. iterable of iterables

Now let's check the flatMap operation and how is it different from Map transformations in Spark -

```
flatMappedConfusion = confusedRDD.flatMap(lambda line : line.split(" "))
flatMappedConfusion.take(5)
```

```
In [11]: flatMappedConfusion = confusedRDD.flatMap(lambda line : line.split(" "))

In [12]: flatMappedConfusion.take(5)
Out[12]: [u'Confusion', u'is', u'the', u'inability', u'to']
```

From the above output it is evident that each word is now acting as single line i.e. it is now iterable of strings.

Filter() Transformation in Spark

This transformation is used to reduce the old RDD based on some condition.

Let's try to find out the lines having confusion term in it in the confusedRDD-

```
onlyconfusion = confusedRDD.filter(lambda line : ("confus" in line.lower()))
onlyconfusion.count()
```

```
In [13]: onlyconfusion = confusedRDD.filter(lambda line : ("confus" in line.lower()))

In [14]: onlyconfusion.count()
Out[14]: 7
```

In the above output , we have found that there 7 lines that have the word confusion in them but to find out what are those lines we can use the collect() action in Spark as shown below 0

```
onlyconfusion.collect()
```

Let's look at another example using the changes.txt file from the spark directory structure.

The task is to include only those commits that are done by "Tathagata Das" in spark module.

```
changesRDD = sc.textFile("/opt/spark/CHANGES.txt")

Daschanges = changesRDD.filter (lambda line: "tathagata.das1565@gmail.com" in line)

Daschanges.count ()
```

```
In [15]: changesRDD = sc.textFile("/opt/spark/CHANGES.txt")

In [16]: Daschanges = changesRDD.filter(lambda line : "tathagata.das1565@gmail.com" in line)

In [17]: Daschanges.count()
Out[17]: 126
```

```
ankurchanges = changesRDD.filter(lambda line : "ankurdave@gmail.com" in line)

ankurchanges.count()
```

```
In [18]: ankurchanges = changesRDD.filter(lambda line : "ankurdave@gmail.com" in line)

In [19]: ankurchanges.count()
Out[19]: 8
```

Sample (withReplacement, fraction, seed)

This transformation is used to pick sample RDD from a larger RDD. It is frequently used in Machine learning operations where a sample of the dataset needs to be taken. The fraction means percentage of the total data you want to take the sample from.

Let's sample the confusedRDD with 50% of it allowing replacement-

```
sampledconfusion = confusedRDD.sample(True,0.5,3) //True implies withReplacement

sampledconfusion.collect()
```

```
In [20]: sampledconfusion = confusedRDD.sample(True,0.5,3)

In [21]: sampledconfusion.collect()
Out[21]:
[u'Confusion is the inability to think as clearly or quickly as you normally do. ',
 u'You may feel disoriented and have difficulty paying attention, remembering, and making decisions.',
 u'Confusion may come on quickly or slowly over time, depending on the cause. ',
 u'Many times, confusion lasts for a short time and goes away. ']
```

union() Transformation in Spark

Union()

Union is basically used to merge two RDDs together if they have the same structure.

Example: A class has two students Abhay and Ankur whose marks have to be combined to get the marks of the entire class. So, here's how you can do it -

```
abhay_marks = [("physics",85),("maths",75),("chemistry",95)]

ankur_marks = [("physics",65),("maths",45),("chemistry",85)]

abhay = sc.parallelize(abhay_marks)

ankur = sc.parallelize(ankur_marks)

abhay.union(ankur).collect()
```

```
In [22]: abhay_marks = [("physics",85),("maths",75),("chemistry",95)]

In [23]: ankur_marks = [("physics",65),("maths",45),("chemistry",85)]

In [24]: abhay = sc.parallelize(abhay_marks)

In [25]: ankur = sc.parallelize(ankur_marks)

In [26]: abhay.union(ankur).collect()
Out[26]:
[('physics', 85),
 ('maths', 75),
 ('chemistry', 95),
 ('physics', 65),
 ('maths', 45),
 ('chemistry', 85)]
```

join() Transformation in Spark

This transformation joins two RDDs based on a common key.

Example: In continuation to the above example of union, you can combine the marks of Abhay and Ankur based on each subject as follows -

```
Subject_wise_marks = abhay.join(ankur)
```

```
In [27]: Subject_wise_marks = abhay.join(ankur)

In [28]: Subject_wise_marks.collect()

Out[28]: [('maths', (75, 45)), ('chemistry', (95, 85)), ('physics', (85, 65))]
```

intersection() Transformation in Spark

Intersection gives you the common terms or objects from the two RDDS.

Example: Let's find out the players who are both good cricketers as well as toppers of the class.

```
Cricket_team =
["sachin","abhay","michael","rahane","david","ross","raj","rahul","hussy","steven","sourav"]

Toppers = ["rahul","abhay","laxman","bill","steve"]

cricketRDD = sc.parallelize(Cricket_team)

toppersRDD = sc.parallelize(Toppers)

toppercricketers = cricketRDD.intersection(toppersRDD)

toppercricketers.collect()
```

distinct() Transformation in Spark

This transformation is used to get rid of any ambiguities. As the name suggest it picks out the lines from the RDD that are unique.

Example: Suppose that there are various movie nominations in different categories. We want to find out, how many movies are nominated overall-

```
best_story = ["movie1","movie3","movie7","movie5","movie8"]

best_direction = ["movie11","movie1","movie5","movie10","movie7"]

best_screenplay = ["movie10","movie4","movie6","movie7","movie3"]

story_rdd = sc.parallelize(best_story)

direction_rdd = sc.parallelize(best_direction)

screen_rdd = sc.parallelize(best_screenplay)

total_nomination_rdd = story_rdd.union(direction_rdd).union(screen_rdd)

total_nomination_rdd.collect()
```

```
In [29]: Cricket_team = ["sachin","abhay","michael","rahane","david","ross","raj","rahul","hussy","steven","sourav"]

In [30]: Toppers = ["rahul","abhay","laxman","bill","steve"]

In [31]: cricketRDD = sc.parallelize(Cricket_team)

In [32]: toppersRDD = sc.parallelize(Toppers)

In [33]: toppercricketers = cricketRDD.intersection(toppersRDD)

In [34]: toppercricketers.collect()
Out[34]: ['abhay', 'rahul']
```

```
unique_movies_rdd = total_nomination_rdd.distinct()
unique_movies_rdd .collect()
```

```
In [38]: story_rdd = sc.parallelize(best_story)

In [39]: direction_rdd = sc.parallelize(best_direction)

In [40]: screen_rdd = sc.parallelize(best_screenplay)

In [41]: total_nomination_rdd = story_rdd.union(direction_rdd).union(screen_rdd)

In [42]: total_nomination_rdd.collect()
Out[42]:
['movie1',
 'movie3',
 'movie7',
 'movie5',
 'movie8',
 'movie11',
```

RDD Partitions

Parallelism is the key feature of any distributed system where operations are done by dividing the data into multiple parallel partitions. The same operation is performed on the partitions simultaneously which helps achieve fast data processing with spark. Map and Reduce operations can be effectively applied in parallel in apache spark by dividing the data into multiple partitions. A copy of each partition within an RDD is distributed across several workers running on different nodes of a cluster so that in case of failure of a single worker the RDD still remains available.

Degree of parallelism of each operation on RDD depends on the fixed number of partitions that an RDD has. We can specify the degree of parallelism or the number of partitions when creating it or later on using the repartition () and coalesce() methods.

```
partRDD = sc.textFile("/opt/spark/CHANGES.txt",4)
```

coalesce () is an optimized version of repartition() method that avoids data movement and is generally used to decrease the number of partitions after filtering a large dataset.

You can check the current number of partitions an RDD has by using the following methods- rdd.getNumPartitions()

```
partRDD.getNumPartitions()
```

```
In [45]: partRDD = sc.textFile("/opt/spark/CHANGES.txt",4)

In [46]: partRDD.getNumPartitions()
Out[46]: 4
```

When processing data with reduceByKey operation, Spark will form as many number of output partitions based on the default parallelism which depends on the numbers of nodes and cores available on each node.

Following are the two versions of the map transformation which work on each partition of RDD separately leveraging maximum cores and memory of the spark cluster-

partRDD.mapPartitions() : This runs a map operation individually on each partition unlike a normal map operation where map is used to operate on each line of the entire RDD.

mapPartitionsWithIndex() : This works same as partRDD.mapPartitions but we can additionally specify the partition number on which this operation has to be applied.

Caching, Accumulators and UDF

Caching

Caching is an important feature in apache spark that decreases the computation time by almost 100X when compared to other distributed computation frameworks like hadoop mapreduce.

Accumulators

from all other tasks. Under such circumstances, accumulators are used. They are write only variables which can be updated by each task and the aggregated result is propagated to the driver program.

UDF (User Defined Functions)

UDF's provide a simple way to add separate functions into Spark that can be used during various transformation stages. UDF's are generally used to perform multiple tasks on Spark RDD's.

Let's take a simple use case to understand the above concepts using movie dataset.

About the dataset:

u.user -- Demographic information about the users; this is a tab separated list of

user id | age | gender | occupation | zip code

```
1|24|M|technician|85711
2|53|F|other|94043
3|23|M|writer|32067
4|24|M|technician|43537
5|33|F|other|15213
6|42|M|executive|98101
7|57|M|administrator|91344
8|36|M|administrator|05201
9|29|M|student|01002
```

You can download the complete dataset from here.

Let's load the data into a spark RDD.

```
userRDD = sc.textFile("/usr/lib/data_backup/opt/spark_usecases/movie/ml-100k/u.user")
```

Let's count the number of users:

```
userRDD.count()
```

```
943
```

Creating a Spark Application

Let's create a user defined function to divide the users into age groups:

```
def parse_N_calculate_age(data):
            userid,age,gender,occupation,zip = data.split("|")
            return  userid, age_group(int(age)),gender,occupation,zip,int(age)
def  age_group(age):
        if age < 10 :
           return '0-10'
        elif age < 20:
           return '10-20'
        elif age < 30:
           return '20-30'
        elif age < 40:
           return '30-40'
        elif age < 50:
```

```
        elif age < 70:
            return '60-70'
        elif age < 80:
            return '70-80'
        else :
            return '80+'
```

```
data_with_age_bucket = userRDD.map(parse_N_calculate_age)
```

Now, let's analyze age group "20-30" for further analysis

```
RDD_20_30 = data_with_age_bucket.filter(lambda line : '20-30' in line)
```

As we are going to analyze the age group 20-30 for multiple things we can put it in-memory for those operations so that it takes less time to do the computation.

Let's count the number users by their profession in the given age_group 20-30

```
freq = RDD_20_30.map(lambda line : line[3]).countByValue()

dict(freq)
```

```
{u'administrator': 19,
 u'artist': 12,
 u'doctor': 2,
 u'educator': 12,
 u'engineer': 23,
 u'entertainment': 8,
 u'executive': 7,
 u'healthcare': 4,
 u'homemaker': 3,
 u'lawyer': 4,
 u'librarian': 11,
 u'marketing': 5,
 u'none': 2,
 u'other': 38,
 u'programmer': 30,
 u'salesman': 2,
 u'scientist': 8,
 u'student': 116,
 u'technician': 12,
 u'writer': 14}
```

Now let's count the number of movie users in the same age group based on gender -

```
age_wise = RDD_20_30.map (lambda line : line[2]).countByValue()

dict(age_wise)
```

```
{u'F': 85, u'M': 247}
```

```
RDD_20_30.unpersist()
```

Now, we will use Accumulators for outlier detection in the above movie dataset. Let's assume that anyone who falls into age group 80+ is outlier and marked as over_age and anyone falling into age group 0-10 is also an outlier and marked as under_age.

```
Under_age = sc.accumulator(0)

Over_age = sc.accumulator(0)
```

```
def outliers(data):
    global Over_age, Under_age
    age_grp = data[1]
    if(age_grp == "70-80"):
        Over_age +=1
    if(age_grp == "0-10"):
        Under_age +=1
    return data
```

Let's use the above method to pass the entire RDD through a function that is used to calculate outliers

```
df = data_with_age_bucket.map(outliers).collect()
```

Now we will check how many users are under age and how many are over aged-

Under_age.value

```
1
```

Over_age.value

```
4
```

Running a Spark application in Standalone Mode

You have learned how to implement various spark RDD concepts in interactive mode using PySpark. However, data engineers cannot perform all the data operations in interactive mode every time. Once the data pipeline and transformations are planned and execution is finalized, the entire code is put into a python script that would run the same spark application in standalone mode.

Here's how we can run our previous example in Spark Standalone Mode -

Remember every standalone spark application runs through a command called spark-submit.

The fundamental format to run spark application in standalone mode is:

Spark-submit

Let's create a demo.py file for our example:

```
from pyspark import SparkContext, SparkConf

conf = SparkConf().setAppName('MyFirstStandaloneApp')
sc = SparkContext(conf=conf)
```

```
            userid,age,gender,occupation,zip = data.split("|")
            return  userid, age_group(int(age)),gender,occupation,zip,int(age)

def  age_group(age):
        if age < 10 :
           return '0-10'
        elif age < 20:
           return '10-20'
        elif age < 30:
           return '20-30'
        elif age < 40:
           return '30-40'
        elif age < 50:
           return '40-50'
        elif age < 60:
           return '50-60'
        elif age < 70:
           return '60-70'
        elif age < 80:
           return '70-80'
        else :
           return '80+'

data_with_age_bucket = userRDD.map(parse_N_calculate_age)

RDD_20_30 = data_with_age_bucket.filter(lambda line : '20-30' in line)

freq = RDD_20_30.map(lambda line : line[3]).countByValue()

print "total user count is ",userRDD.count()

print "total movie users profession wise ",dict(freq)

Under_age = sc.accumulator(0)
Over_age = sc.accumulator(0)

def outliers(data):
    global Over_age, Under_age
    age_grp = data[1]
    if(age_grp == "70-80"):
        Over_age +=1
    if(age_grp == "0-10"):
        Under_age +=1
    return data

df = data_with_age_bucket.map(outliers).collect()

print "under age users of the movie are ",Under_age
print "over age users of the movie are ",Over_age
```

You can run the above application as follows-

spark-submit demo.py

The output looks would be -

```
total user count is  943
total movie users profession wise  {u'administrator': 19, u'lawyer': 4, u'healthcare': 4,
u'marketing': 5, u'executive': 7, u'doctor': 2, u'scientist': 8, u'student': 116, u'technician':
12, u'librarian': 11, u'programmer': 30, u'salesman': 2, u'homemaker': 3, u'engineer': 23,
u'none': 2, u'artist': 12, u'writer': 14, u'entertainment': 8, u'other': 38, u'educator': 12}
under age users of the movie are  1
over age users of the movie are  4
```

Launching Spark Application on a Cluster

applications can be run in with 3 different cluster managers-

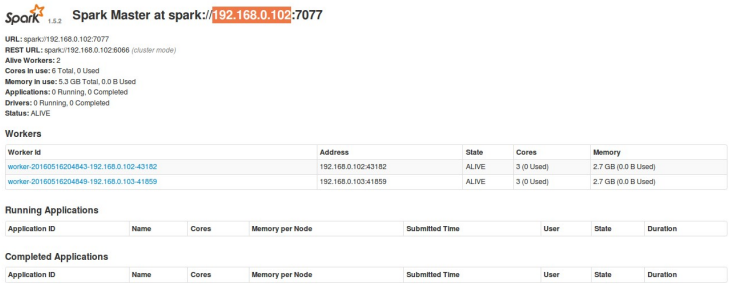
1. **Apache Hadoop YARN**: HDFS is the source storage and YARN is the resource manager in this scenario. All read or write operations in this mode are performed on HDFS.
2. **Apache Mesos**: A distributed mode where the resource management is handled by the cluster manager Apache Mesos developed by UC Berkeley.
3. **Standalone Mode:** In this mode the resource management is handled by the spark in-built resource manager.

Let's run the spark application in cluster mode where resource management is being handled by spark's own resource manager and the source of data is local file system.

In order to run the application in cluster mode you should have your distributed cluster set up already with all the workers listening to the master.

In our example the master is running on IP - 192.168.0.102 over default port 7077 with two worker nodes.

The cluster page gives a detailed information about the spark cluster -



From the image you can see that the spark cluster has two worker nodes one at 192.168.0.102 over port 43182 and another at 192.168.0.103 listening over port 41859. Each machine has been assigned 3 cores and 2.7 GB memory for task manipulations. There are no spark applications running in the above image, so let's fire a spark job in cluster mode and see the results.

We perform a log analysis of the spark jobs that have failed in the cluster to find out the number of errors that have occurred and of which how many I/O errors have been there. Here's how our log file looks like-

```
16/05/01 15:32:28 ERROR FileAppender: Error writing stream to file /opt/spark-1.5.2-b
hadoop2.4/work/app-20160501151716-0000/1/stderr

java.io.IOException: Stream closed

        at java.io.BufferedInputStream.getBuflfOpen(BufferedInputStream.java:170)

        at java.io.BufferedInputStream.read1(BufferedInputStream.java:283)

        at java.io.BufferedInputStream.read(BufferedInputStream.java:345)

        at java.io.FilterInputStream.read(FilterInputStream.java:107)

        at org.apache.spark.util.logging.FileAppender.appendStreamToFile(FileAppender.scala:70)


org.apache.spark.util.logging.FileAppender$$anon$1$$anonfun$run$1.apply$mcV$sp(FileAppender.scala:3


org.apache.spark.util.logging.FileAppender$$anon$1$$anonfun$run$1.apply(FileAppender.scala:39)


org.apache.spark.util.logging.FileAppender$$anon$1$$anonfun$run$1.apply(FileAppender.scala:39)
```

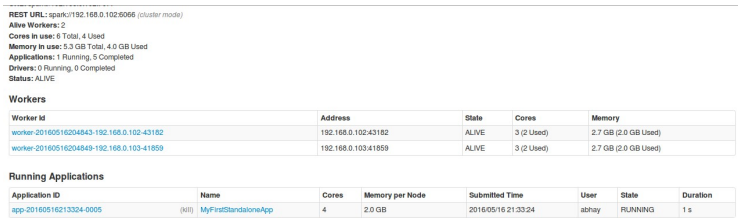
Here is the code to do log analysis in the python file named as “python_log_read.py”:

```
From pyspark import SparkContext, SparkConf

conf = SparkConf().setAppName('MyFirstStandaloneApp')
sc = SparkContext(conf=conf)

log_data = sc.textFile("file:///opt/spark/logs/spark-abhay-org.apache.spark.deploy.worker.Worker-1-Boss-Machine.out.3")

print("Total lines read  are \n")
print log_data.count()

print("Number of Error in Document are :: \n")

errors = log_data.filter(lambda row : "ERROR" in row)

print errors.count()

print("Number of IO Error in the  Document are :: \n")
IOException = log_data.filter(lambda row : "IOException" in row)
print IOException.count()
```

Let’s run it over our two node spark standalone cluster using the following command:

```
spark-submit --master spark://192.168.0.102:7077 --deploy-mode client python_log_read.py
```

We see on the cluster web page that the job has been submitted in the cluster:



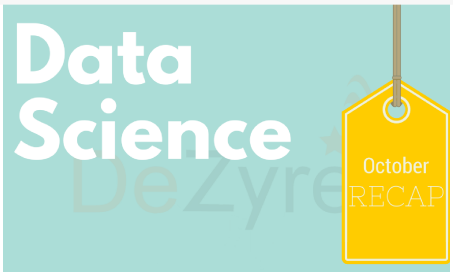
The output of the application is displayed as:

```
Total lines read  are

75

Number of Error in Document are ::

7

Number of IO Error in the  Document are ::

6
```

PREVIOUS NEXT

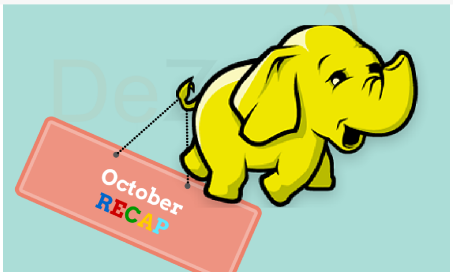
Sign In

PySpark Tutorial-Learn to use Apache Spark with Python Blog



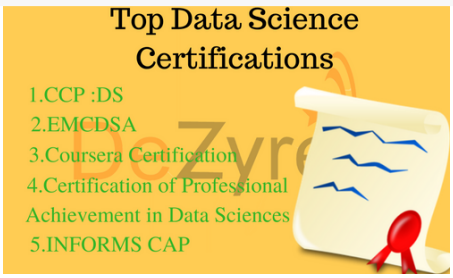
Recap of Data Science News for October

Recap of Apache Spark News for Octo

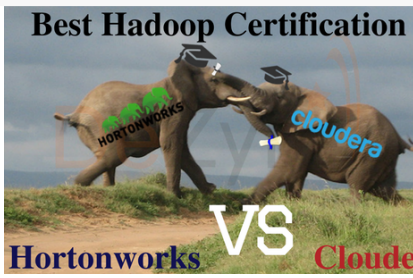


Recap of Hadoop News for October

Scenario-Based Hadoop Interview Questions to pr



Top Data Science Certifications to choose from in 2016



Best Hadoop Certification: Cloudera vs Hor

Other Tutorials

Apache Spark Tutorial–Run your First Spark Program

Step-by-Step Apache Spark Installation Tutorial

Introduction to Apache Spark Tutorial

Sign In

Courses

- Big Data and Hadoop Certification Training
- Hadoop Project based Training
- Apache Spark Certification Training
- Data Science Training Course
- Data Science in R Programming
- Salesforce Certifications - ADM 201 and DEV 401
- Hadoop Administration for Big Data
- Certificate in NoSQL Databases for Big Data
- Advanced MS Excel with Macro, VBA and Dashboards

EV SSL Certificate

About DeZyre

- About Us
- Contact Us
- DeZyre Reviews
- Blog
- Tutorials
- Webinar
- Online Hackathons
- Student Portfolios
- Privacy Policy
- Disclaimer

Connect with us

Online