# Comprehensive Introduction to Apache Spark, RDDs & Dataframes (using PySpark)

## Introduction

Industry estimates that we are creating more than 2.5 Quintillion bytes of data every year.

Think of it for a moment – 1 Qunitillion = 1 Million Billion! Can you imagine how many drives / CDs / Blue-ray DVDs would be required to store them? It is difficult to imagine this scale of data generation even as a data science professional. While this pace of data generation is very exciting, it has created entirely new set of challenges and has forced us to find new ways to handle Big Huge data effectively.

Big Data is not a new phenomena. It has been around for a while now. However, it has become really important with this pace of data generation. In past, several systems were developed for processing big data. Most of them were based on MapReduce framework. These frameworks typically rely on use of hard disk for saving and retrieving the results. However, this turns out to be very costly in terms of time and speed.

On the other hand, Organizations have never been more hungrier to add a competitive differentiation through understanding this data and offering its customer a much better experience. Imagine how valuable would be Facebook, if it did not understand your interests well? The traditional hard disk based MapReduce kind of frameworks do not help much to address this challenge.

In this article, I will introduce you to one such framework, which has made querying and analysing data at a large scale much more efficient than previous systems / frameworks – Read on!

P.S. This article is meant for complete beginners on the topic and presumes minimal prior knowledge in Big Data

## Table of Contents

1. Challenges while working with Big Data
2. Introduction to Distributed Computing Framework
3. What is Apache Spark?
   * History of Spark
   * Common terms used
   * Benefits of Spark over traditional big data frameworks
4. Installation of Apache Spark (with Python)
5. Python vs Scala
6. Getting up to speed with RDD / Dataframe / Dataset
7. Solving a machine learning problem

## Challenges while working with big data

Challenges associated with big data can be classified in following categories:

* Challenges in data capturing: Capturing huge data could be a tough task because of large volume and high velocity. There are millions of sources emanating data at high speed. To deal with this challenge, we have created devices which can capture the data effectively and efficiently. For example, sensors which not only sense data like temperature of a room, steps count, weather parameters in real time, but send this information directly over to cloud for storage.
* Challenges with data storage: Given the increase in data generation, we need more efficient ways to store data. This challenge is typically dealt by combination of various methods including increasing disk sizes, compressing the data and using multiple machines, which are connected to each other and can share data efficiently.
* Challenges with Querying and Analysing data: This is probably the most difficult task at hand. The task is to not only to retrieve the past data, but also coming out with insights in real time (or as little time as possible). To handle this challenge, we can look at several options. One options is to increase the processing speed. However, this normally comes with increase in cost and can not scale as much. Alternately, we can build a network of machines or nodes known as “Cluster”. In this scenario, we first break a task to sub-tasks and distribute them to different nodes. At the end, we aggregate the output of each node to have final output. This distribution of task is known as “Distributed Computing”

Now that I have spoken of Distributed computing, let us get a bit deeper into it!

## What is Distributing Computing Framework?

In simple terms, distributed computing is just a distributed system, where multiple machines are doing certain work at the same time. While doing the work, machines will communicate with each other by passing messages between them. Distributed computing is useful, when there is requirement of fast processing (computation) on huge data.

Let us take a simple analogy to explain the concept. Let us say, you had to count the number of books in various sections of a really large library. And you have to finish it in less than an hour. This number has to be exact and can not be approximated. What would you do? If I was in this position, I would call up as many friends as I can and divide areas / rooms among them. I’ll divide the work in non-overlapping manner and ask them to report back to be in 55 minutes. Once they come back, I’ll simply add up the numbers to come up with a solution. This is exactly how distributed computing works.

Apache Hadoop and Apache Spark are well-known examples of Big data processing systems. Hadoop and Spark are designed for distributed processing of large data sets across clusters of computers. Although, Hadoop is widely used for fast distributed computing, it has several disadvantages. For example, it does not use “In-memory computation“, which is nothing but keeping the data in RAM instead of Hard Disk for fast processing. In-memory computation enables faster processing of Big data. When Apache Spark was developed, it overcame this problem by using In-memory computation for fast computing.

MapReduce is also used widely, when the task is to process huge amounts of data, in parallel (more than one machines are doing a certain task at the same time), on large clusters. You can learn more about MapReduce from this [link](https://www.analyticsvidhya.com/blog/2014/05/introduction-mapreduce/).

## What is Apache Spark?

Apache Spark is a fast cluster computing framework which is used for processing, querying and analyzing Big data. It is based on In-memory computation, which is a big advantage of Apache Spark over several other big data Frameworks. Apache Spark is open source and one of the most famous Big data framework. It can run tasks up to 100 times faster, when it utilizes the in-memory computations and 10 times faster when it uses disk than traditional map-reduce tasks.

Please note that Apache Spark is not a replacement of Hadoop. It is actually designed to run on top of Hadoop.

### History of Apache Spark

Apache Spark was originally created at University of California, Berkeley’s AMPLab in 2009. The Spark code base was later donated to the Apache Software Foundation. Subsequently, it was open sourced in 2010. Spark is mostly written in Scala language. It has some code written in Java, Python and R. Apache Spark provides several APIs for programmers which include Java, Scala, R and Python.

### Key terms used in Apache Spark:

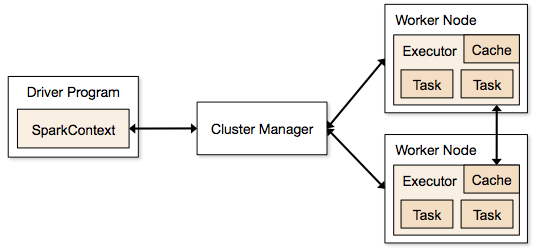


Image source: https://spark.apache.org/docs/1.1.1/img/cluster-overview.png

Spark Context: It holds a connection with Spark cluster manager. All Spark applications run as independent set of processes, coordinated by a SparkContext in a program.

Driver and Worker: A driver is incharge of the process of running the main() function of an application and creating the SparkContext. A worker, on the other hand, is any node that can run program in the cluster. If a process is launched for an application, then this application acquires executors at worker node.

Cluster Manager: Cluster manager allocates resources to each application in driver program. There are three types of cluster managers supported by Apache Spark – Standalone, Mesos and YARN. Apache Spark is agnostic to the underlying cluster manager, so we can install any cluster manager, each has its own unique advantages depending upon the goal. They all are different in terms of scheduling, security and monitoring. Once SparkContext connects to the cluster manager, it acquires executors on a cluster node, these executors are worker nodes on cluster which work independently on each tasks and interact with each other.

### How Apache Spark is better than traditional big data framework?

In-memory computation: The biggest advantage of Apache Spark comes from the fact that it saves and loads the data in and from the RAM rather than from the disk (Hard Drive). If we talk about memory hierarchy, RAM has much higher processing speed than Hard Drive (illustrated in figure below). Since the prices of memory has come down significantly in last few years, in-memory computations have gained a lot of momentum.

Spark uses in-memory computations to speed up 100 times faster than Hadoop framework.

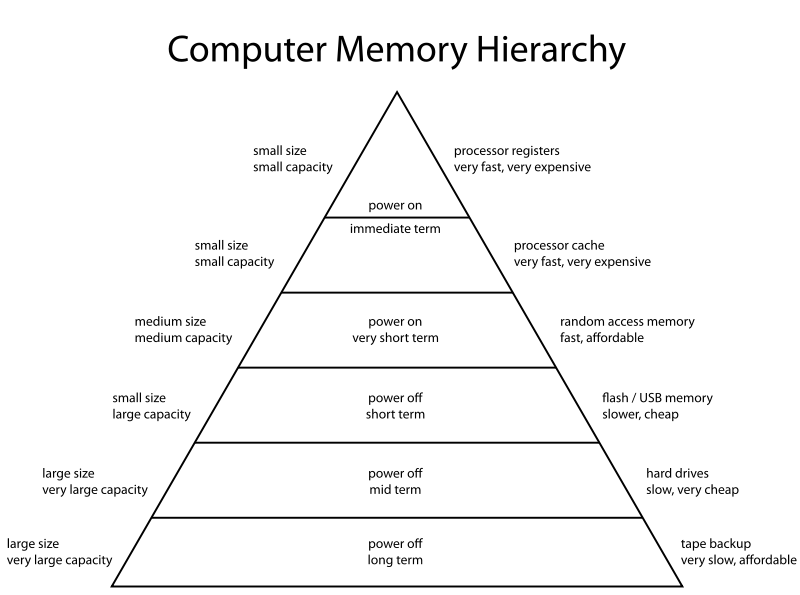
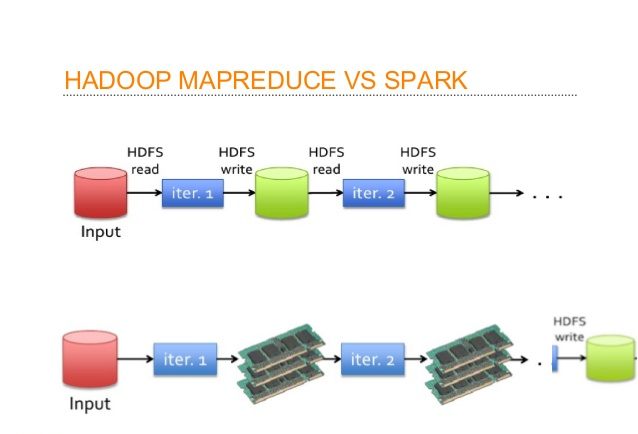


Image Source: https://en.wikipedia.org/wiki/Memory\_hierarchy

In Hadoop, tasks are distributed among the nodes of a cluster, which in turn save data on disk. When that data is required for processing, each node has to load the data from the disk and save the data into disk after performing operation. This process ends up adding cost in terms of speed and time, because disk operations are far slower than RAM operations. It also requires time to convert the data in a particular format when writing the data from RAM to disk. This conversion is known as Serialization and reverse is Deserialization.

Let’s look at the MapReduce process to understand the advantage of in-memory computation better. Suppose, there are several map-reduce tasks happening one after another. At the start of the computations, both technologies (Hadoop and Spark), read the data from disk for mapping. Hadoop performs the map operation and saves the results back to hard drive. However, in case of Apache Spark, the results are stored in RAM.

In the next step (Reduce operation), Hadoop reads the saved data from the hard drive, where as Apache Spark reads it from RAM. This creates a difference in a single MapReduce operation. Now imagine, if there were multiple map-reduce operations, how much time difference would you see at the end of task completion.



Language Support: Apache Spark has API support for popular data science languages like Python, R, Scala and Java.

Supports Real time and Batch processing: Apache Spark supports “Batch data” processing where a group of transactions is collected over a period of time. It also supports real time data processing, where data is continuously flowing from the source. For example, weather information coming in from sensors can be processed by Apache Spark directly.

Lazy operation: Lazy operations are used to optimize solutions in Apache Spark. I will discuss about lazy evaluation in later part of this article. For now, we can think that there are some operations which do not execute until we require results.

Support for multiple transformations and actions: Another advantage of Apache Spark over Hadoop is that Hadoop supports only MapReduce but Apache Spark support many transformations and actions including MapReduce.

There are further advantages of Apache Spark in comparison to Hadoop. For example, Apache Spark is much faster while doing Map side shuffling and Reduce side shuffling. However, shuffling is a complex topic in itself and requires an entire article in itself. Hence, I am not talking about it in more details here.

## Python vs Scala:

One of the common question people ask is whether it is necessary to learn Scala to learn Spark? If you are some one who already knows Python to some extent or are just exploring Spark as of now, you can stick to Python to start with. However, if you want to process some serious data across several machines and clusters, it is strongly recommended that you learn Scala. Computation speed in Python is much slower than Scala in Apache Spark.

* Scala is native language for Spark (because Spark itself written in Scala).
* Scala is a compiled language where as Python is an interpreted language.
* Python has process based executors where as Scala has thread based executors.
* Python is not a JVM (java virtual machine) language.

## Apache Spark data representations: RDD / Dataframe / Dataset

Spark has three data representations viz RDD, Dataframe, Dataset. For each data representation, Spark has a different API. For example, later in this article I am going to use ml (a library), which currently supports only Dataframe API. Dataframe is much faster than RDD because it has metadata (some information about data) associated with it, which allows Spark to optimize query plan. Refer to this [link](https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html) to know more about optimization. The Dataframe feature in Apache Spark was added in Spark 1.3. If you want to know more in depth about when to use RDD, Dataframe and Dataset you can refer this [link](https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html).

In this article, I will first spend some time on RDD, to get you started with Apache Spark. Later, I will spend some time on Dataframes. Dataframes share some common characteristics with RDD (transformations and actions). In this article, I am not going to talk about Dataset as this functionality is not included in PySpark.

### RDD:

After installing and configuring PySpark, we can start programming using Spark in Python. But to use Spark functionality, we must use RDD. RDD (Resilient Distributed Database) is a collection of elements, that can be divided across multiple nodes in a cluster to run parallel processing. It is also fault tolerant collection of elements, which means it can automatically recover from failures. RDD is immutable, we can create RDD once but can’t change it. We can apply any number of operation on it and can create another RDD by applying some transformations. Here are a few things to keep in mind about RDD:

We can apply 2 types of operations on RDDs:

Transformation: Transformation refers to the operation applied on a RDD to create new RDD.  
Action: Actions refer to an operation which also apply on RDD that perform computation and send the result back to driver.

Example: Map (Transformation) performs operation on each element of RDD and returns a new RDD. But, in case of Reduce (Action), it reduces / aggregates the output of a map by applying some functions (Reduce by key). There are many transformations and actions are defined in Apache Spark documentation, I will discuss these in a later article.

RDDs use Shared Variable:  
The parallel operations in Apache Spark use shared variable. It means that whenever a task is sent by a driver to executors program in a cluster, a copy of shared variable is sent to each node in a cluster, so that they can use this variable while performing task. Accumulator and Broadcast are the two types of shared variables supported by Apache Spark.  
Broadcast: We can use the Broadcast variable to save the copy of data across all node.  
Accumulator: In Accumulator variables are used for aggregating the information.

How to Create RDD in Apache Spark

Existing storage: When we want to create a RDD though existing storage in driver program (which we would like to be parallelized). For example, converting a list to RDD, which is already created in a driver program.

External sources: When we want to create a RDD though external sources such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.

Writing first program in Apache Spark

I have already discussed that RDD supports two type of operations, which are transformation and action. Let us get down to writing our first program:

Step1: Create SparkContext

First step in any Apache programming is to create a SparkContext. SparkContext is needed when we want to execute operations in a cluster. SparkContext tells Spark how and where to access a cluster. It is first step to connect with Apache Cluster. If you are using Spark Shell, we will find that this is already created. Otherwise, we can create the Spark Context by importing, initializing and providing the configuration settings. For example:

from pyspark import SparkContext

sc = SparkContext()

Step2: Create a RDD

I have already discussed that we can create RDD in two ways: Either from an existing storage or from an external storage. Let’s create our first RDD. SparkContext has parallelize method, which is used for creating the Spark RDD from an iterable (like list, tuple..) already present in driver program.

We can also provide the number of partitions as a parameter to parallelize method. If we do not give number of partition parameter, then Spark will automatically set the number of partition in a cluster. The number of partition can be set manually by passing second parameter to parallelize method. For example, sc.parallelize(data, 10)), where data is an existing data in driver program and 10 is the number of partitions.  
Lets create the first Spark RDD called rdd.

data = range(1,1000)

rdd = sc.parallelize(data)

We have a collect method to see the content of RDD.

rdd.collect()

To see the first n element of a RDD we have a method take:

rdd.take(2) # It will print first 2 elements of rdd

We have 2 parallel operations in RDD which are Transformation and Action. Transformation and Action were already discussed briefly earlier. So let’s see how transformation works. Remember that RDDs are immutable – so we can’t change our RDD, but we can apply transformation on it. Let’s see an example of map transformation to demonstrate how transformation works.

Step 3: Map transformation.

Map transformation returns a Mapped RDD by applying function to each element of the base RDD. Let’s repeat the first step of creating a RDD from existing source, For example,

data = ['Hello' , 'I' , 'AM', 'Ankit ', 'Gupta']

Rdd = sc.parallelize(data)

Now a RDD (name is ‘Rdd’) is created from the existing source, which is a list of string in a driver program. We will now apply lambda function to each element of Rdd and return the mapped (transformed) RDD (word,1) pair in the Rdd1.

Rdd1 = Rdd.map(lambda x: (x,1))

Let’s see the out of this map operation.

Rdd1.collect()

output: [('Hello', 1), ('I', 1), ('AM', 1), ('Ankit ', 1), ('Gupta', 1)]

If you noticed, nothing happened after applying the lambda function on Rdd1 (we won’t see any computation happening in a cluster). This is called the lazy operation. All transformation operations in Spark are lazy, which means that we will not see any computations on RDD, until we need them for further action.

Spark remembers which transformation is applied to which RDD with the help of DAG (Directed a Cyclic Graph). The lazy evaluation helps Spark to optimize the solution because Spark will get time to see the DAG before actually executing the operations on RDD. This enables Spark to run operations more efficiently.

In the code above, collect() and take() are the examples of an action.

There are many number of transformation defined in Apache Spark. We will talk more about them in a future post.

## Solving a machine learning problem:

We have covered a lot of ground already. We started with understanding what Spark brings to the table, its data representations, installed Spark and have already played with our first RDD. Now, I’ll demonstrate solution to [“Practice Problem: Black Friday”](https://datahack.analyticsvidhya.com/contest/black-friday/) using Apache Spark. Even if you don’t understand these commands completely as of now, it is fine. Just follow along, we will take them up again in a future tutorial.

Let’s look at the steps:

Reading a data file (csv)

For reading the csv file in Apache Spark, we need to specify the library in python shell. Lets read the the data from a csv files to create the Dataframe and apply some data science skills on this Dataframe like we do in Pandas.

For reading the csv file, first we need to download Spark-csv package ([Latest](https://spark-packages.org/package/databricks/spark-csv)) and extract this package into the home directory of Spark. Then, we need to open a PySpark shell and include the package (I am using “spark-csv\_2.10:1.3.0”).

$ ./bin/pyspark --packages com.databricks:spark-csv\_2.10:1.3.0

In Apache Spark, we can read the csv file and create a Dataframe with the help of SQLContext. Dataframe is a distributed collection of observations (rows) with column name, just like a table. Let’s see how can we do that.

Please note that since I am using pyspark shell, there is already a sparkContext and sqlContext available for me to use. In case, you are not using pyspark shell, you might need to type in the following commands as well:

sc = sparkContext()

sqlContext = SQLContext(sc)

First download the train and test file and load these with the help of SparkContext

train = sqlContext.load(source="com.databricks.spark.csv", path = 'PATH/train.csv', header = True,inferSchema = True)

test = sqlContext.load(source="com.databricks.spark.csv", path = 'PATH/test-comb.csv', header = True,inferSchema = True)

PATH is the location of folder, where your train and test csv files are located. Header is True, it means that the csv files contains the header. We are using inferSchema is True for telling sqlContext to automatically detect the data type of each column in data frame. If we do not set inferSchema to true, all columns will be read as string.

Analyze the data type

To see the types of columns in Dataframe, we can use the method printSchema(). Lets apply printSchema() on train which will Print the schema in a tree format.

train.printSchema()

Previewing the data set

To see the first n rows of a Dataframe, we have head() method in PySpark, just like pandas in python. We need to provide an argument (number of rows) inside the head method. Lets see first 10 rows of train:

train.head(10)

To see the number of rows in a data frame we need to call a method count(). Lets check the number of rows in train. The count method in pandas and Spark are different.

train.count()

Impute Missing values

We can check number of not null observations in train and test by calling drop() method. By default, drop() method will drop a row if it contains any null value. We can also pass ‘all” to drop a row only if all its values are null.

train.na.drop().count(),test.na.drop('any').count()

Here, I am imputing null values in train and test file with -1. Imputing the values with -1 is not an elegant solution. We have several algorithms / techniques to impute null values, but for the simplicity I am imputing null with constant value (-1). We can transform our base train, test Dataframes after applying this imputation. For imputing constant value, we have fillna method. Lets fill the -1 in-place of null in all columns.

train = train.fillna(-1)

test = test.fillna(-1)

Analyzing numerical features

We can also see the various summary Statistics of a Dataframe columns using describe() method, which shows statistics for numerical variables. To show the results we need to call show() method.

train.describe().show()

Sub-setting Columns

Let’s select a column called ‘User\_ID’ from a train, we need to call a method ‘select’ and pass the column name which we want to select. The select method will show result for selected column. We can also select more than one column from a data frame by providing columns name separated by comma.

train.select('User\_ID').show()

Analyzing categorical features

To start building a model, we need to see the distribution of categorical features in train and test. Here I am showing this for only Product\_ID but we can also do the same for any categorical feature. Let’s see the number of distinct categories of “Product\_ID” in train and test. Which we can do by applying methods distinct() and count().

train.select('Product\_ID').distinct().count(), test.select('Product\_ID').distinct().count()

Output:(3631, 3491)

After counting the number of distinct values for train and test we can see the train has more categories than test. Let us check what are the categories for Product\_ID, which are in test but not in train by applying subtract method.We can also do the same for all categorical feature.

diff\_cat\_in\_train\_test=test.select('Product\_ID').subtract(train.select('Product\_ID'))

diff\_cat\_in\_train\_test.distinct().count()# For distict count

Output: 46

Above you can see that 46 different categories are in test not in train. In this case, either we collect more data about them or skip the rows in test for those categories(invalid category) which are not in train.

Transforming categorical variables to labels

We also need to transform categorical columns to label by applying StringIndexer Transformation on Product\_ID which will encode the Product\_ID column of labels to a column of label indices. You can see more about this from the [link](https://spark.apache.org/docs/latest/ml-features.html" \l "stringindexer)

from pyspark.ml.feature import StringIndexer

plan\_indexer = StringIndexer(inputCol = 'Product\_ID', outputCol = 'product\_ID')

labeller = plan\_indexer.fit(train)

Above, we build a ‘labeller’ by applying fit() method on train Dataframe. Later we will use this ‘labeller’ to transform our train and test. Let us transform our train and test Dataframe with the help of labeller. We need to call transform method for doing that. We will store the transformation result in Train1 and Test1.

Train1 = labeller.transform(train)

Test1 = labeller.transform(test)

Lets check the resulting Train1 Dataframe.

Train1.show()

The show method on Train1 Dataframe will show that we successfully added one transformed column product\_ID in our previous train Dataframe.

Selecting Features to Build a Machine Learning Model

Let’s try to create a formula for Machine learning model like we do in R. First, we need to import RFormula from the pyspark.ml.feature. Then we need to specify the dependent and independent column inside this formula. We also have to specify the names for features column and label column.

from pyspark.ml.feature import RFormula

formula = RFormula(formula="Purchase ~ Age+ Occupation +City\_Category+Stay\_In\_Current\_City\_Years+Product\_Category\_1+Product\_Category\_2+ Gender",featuresCol="features",labelCol="label")

After creating the formula we need to fit this formula on our Train1 and transform Train1,Test1 through this formula. Lets see how to do this and after fitting transform train1,Test1 in train1,test1.

t1 = formula.fit(Train1)

train1 = t1.transform(Train1)

test1 = t1.transform(Test1)

We can see the transformed train1, test1.

train1.show()

After applying the formula we can see that train1 and test1 have 2 extra columns called features and label those we have specified in the formula (featuresCol=”features” and labelCol=”label”). The intuition is that all categorical variables in the features column in train1 and test1 are transformed to the numerical and the numerical variables are same as before for applying ML. Purchase variable will transom to label column. We can also look at the column features and label in train1 and test1.

train1.select('features').show()

train1.select('label').show()

Building a Machine Learning Model: Random Forest

After applying the RFormula and transforming the Dataframe, we now need to develop the machine learning model on this data. I want to apply a random forest regressor for this task. Let us import a random forest regressor, which is defined in pyspark.ml.regression and then create a model called rf. I am going to use default parameters for randomforest algorithm.

from pyspark.ml.regression import RandomForestRegressor

rf = RandomForestRegressor()

After creating a model rf we need to divide our train1 data to train\_cv and test\_cv for cross validation.

Here we are dividing train1 Dataframe in 70% for train\_cv and 30% test\_cv.

(train\_cv, test\_cv) = train1.randomSplit([0.7, 0.3])

Now build the model on train\_cv and predict on test\_cv. The results will save in predictions.

model1 = rf.fit(train\_cv)

predictions = model1.transform(test\_cv)

If you check the columns in predictions Dataframe, there is one column called prediction which has prediction result for test\_cv.

model1 = rf.fit(train\_cv)

predictions = model1.transform(test\_cv)

Lets evaluate our predictions on test\_cv and see what is the mean squae error.

To evaluate model we need to import RegressionEvaluator from the pyspark.ml.evaluation. We have to create an object for this. There is a method called evaluate for evaluator which will evaluate the model. We need to specify the metrics for that.

from pyspark.ml.evaluation import RegressionEvaluator

evaluator = RegressionEvaluator()

mse = evaluator.evaluate(predictions,{evaluator.metricName:"mse" })

import numpy as np

np.sqrt(mse), mse

After evaluation we can see that our root mean square error is 3773.1460883883865 which is a square root of mse.

Now, we will implement the same process on full train1 dataset.

model = rf.fit(train1)

predictions1 = model.transform(test1)

After prediction, we need to select those columns which are required in Black Friday competition submission.

df = predictions1.selectExpr("User\_ID as User\_ID", "Product\_ID as Product\_ID", 'prediction as Purchase')

Now we need to write the df in csv format for submission.

df.toPandas().to\_csv('submission.csv')

After writing into the csv file(submission.csv). We can upload our first solution to see the score, I got the score 3822.121053 which is not very bad for first model out of Spark!

# Using PySpark to perform Transformations and Actions on RDD

## Introduction

In my previous [article](https://www.analyticsvidhya.com/blog/2016/09/comprehensive-introduction-to-apache-spark-rdds-dataframes-using-pyspark/), I introduced you to the basics of Apache Spark, different data representations (RDD / DataFrame / Dataset) and basics of operations (Transformation and Action). We even solved a machine learning problem from one of our past [hackathons](https://datahack.analyticsvidhya.com/contest/black-friday/). In this article, I will continue from the place I left in my previous article. I will focus on manipulating RDD in PySpark by applying operations (Transformation and Actions).

As you would remember, a RDD (Resilient Distributed Database) is a collection of elements, that can be divided across multiple nodes in a cluster to run parallel processing. It is also a fault tolerant collection of elements, which means it can automatically recover from failures. RDD is immutable, i.e. once created, we can not change a RDD. So, then how do I apply operations on a RDD? Well, we apply an operation and store results in another RDD

For this article, one must have some understanding about Apache Spark and hands on experience in python programming.

## Table of Contents

1. Recap
2. What is Transformation and Action?
   * Transformation and Action
   * Major Categories
3. Applying Transformation and Action
   * General
   * Mathematical and Statistical
   * Set Theory and Relational
   * Data-structure and IO



## Recap

Let’s recall concepts about RDD from our [previous](https://www.analyticsvidhya.com/blog/2016/09/comprehensive-introduction-to-apache-spark-rdds-dataframes-using-pyspark/) article:

* We can create a RDD in two different ways, from existing source and external source.
* We can apply two types of operations on RDD, namely “transformation” and “action”. All transformations on RDD are lazy in nature, which means that computations on RDD are not done until we apply an action.
* RDDs are immutable in nature i.e. we cannot change the RDD, we need to transform it by applying transformation(s). There are various transformations and actions, which can be applied on RDD.

Before applying transformations and actions on RDD, we need to first open the PySpark shell (please refer to my previous [article to setup PySpark](https://www.analyticsvidhya.com/blog/2016/09/comprehensive-introduction-to-apache-spark-rdds-dataframes-using-pyspark/) ).

$ ./bin/pyspark

## What is Transformation and Action?

Spark has certain operations which can be performed on RDD. An operation is a method, which can be applied on a RDD to accomplish certain task. RDD supports two types of operations, which are Action and Transformation. An operation can be something as simple as sorting, filtering and summarizing data.

Let’s take few examples to understand the concept of transformation and action better. Let’s assume, we want to develop a machine learning model on a data set. Before applying a machine learning model, we will need to perform certain tasks:

1. Understand the data ( List out the number of columns in data and their type)
2. Preprocess the data (Remove null value observations on data).
3. Filter the data (Let’s say, we want to filter the observations corresponding to males data)
4. Fill the null values in data ( Filling the null values in data by constant, mean, median, etc)
5. Calculate the features in data

All the above mentioned tasks are examples of an operation. In Spark, operations are divided into 2 parts – one is transformation and second is action. Find below a brief descriptions of these operations.

Transformation: Transformation refers to the operation applied on a RDD to create new RDD. Filter, groupBy and map are the examples of transformations.

Actions: Actions refer to an operation which also applies on RDD, that instructs Spark to perform computation and send the result back to driver. This is an example of action.

The Transformations and Actions in Apache Spark are divided into 4 major categories:

* General
* Mathematical and Statistical
* Set Theory and Relational
* Data-structure and IO

## Applying Transformation and Action

To understand the operations, I am going to use the text file from my previous article. Let’s begin, I have already copied and pasted all text from my blog in a textfile called blogtexts. To download this file you can refer to this [link](https://drive.google.com/open?id=0B3GihyBQdu_PZ05MM2x3VTdvQ2M). Before applying operations on blogtexts, we need to first load this file with the help of SparkContext.

rdd = sc.textFile("PATH/blogtexts")

In above code, ‘PATH’ is the location of blogtexts. Let’s see first 5 elements of RDD.

rdd.take(5)

Output:

[u'Think of it for a moment \u2013 1 Qunitillion = 1 Million Billion! Can you imagine how many drives / CDs / Blue-ray DVDs would be required to store them? It is difficult to imagine this scale of data generation even as a data science professional. While this pace of data generation is very exciting, it has created entirely new set of challenges and has forced us to find new ways to handle Big Huge data effectively.',

u'',

u'Big Data is not a new phenomena. It has been around for a while now. However, it has become really important with this pace of data generation. In past, several systems were developed for processing big data. Most of them were based on MapReduce framework. These frameworks typically rely on use of hard disk for saving and retrieving the results. However, this turns out to be very costly in terms of time and speed.',

u'',

u'On the other hand, Organizations have never been more hungrier to add a competitive differentiation through understanding this data and offering its customer a much better experience. Imagine how valuable would be Facebook, if it did not understand your interests well? The traditional hard disk based MapReduce kind of frameworks do not help much to address this challenge.'

]

Now lets see one by one how transformations and actions work on RDDs.

## General transformations

For each transformation, I have first laid out the need of the transformation in the form of a question and then answered it in the subsequent section.

### Transformation: map and flatMap

Q1: Convert all words in a rdd to lowercase and split the lines of a document using space.

To lower the case of each word of a document, we can use the map transformation. A map transformation is useful when we need to transform a RDD by applying a function to each element. So how can we use map transformation on ‘rdd’ in our case?

Solution: Let’s see through the example, Apply a function called “Func” on each words of a document ( blogtexts ). “Func” will do two things:

1. It will take a corpus, lower the each words in this corpus.
2. After that it splits the words in each line by space.

To do this first we need to write “Func” and then apply this function using map.

def Func(lines):

lines = lines.lower()

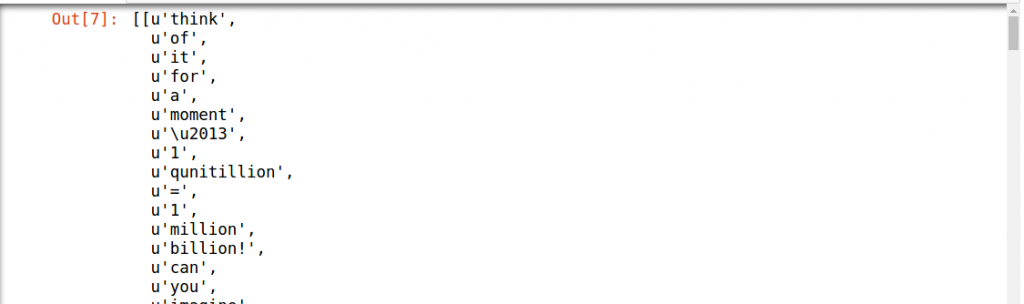
lines = lines.split()

return lines

rdd1 = rdd.map(Func)

After applying the function (Func) on “rdd”, we have transformed this “rdd” into “rdd1”, we can see the first 5 elements of “rdd1” by applying take operation (which is an action).

rdd1.take(5)



Output is too long so, I have just attached a snippet of it. We can also see that our output is not flat (it’s a nested list). So for getting the flat output, we need to apply a transformation which will flatten the output, The transformation “flatMap” will help here:

The “flatMap” transformation will return a new RDD by first applying a function to all elements of this RDD, and then flattening the results. This is the main difference between the “flatMap” and map transformations. Let’s apply a “flatMap” transformation on “rdd” , then take the result of this transformation in “rdd2” and print the result after applying this transformation.

rdd2 = rdd.flatMap(Func)

rdd2.take(5)

Output: [u'think', u'of', u'it', u'for', u'a']

You can now observe that the new output is flattened out.

### Transformation: filter

Q2: Next, I want to remove the words, which are not necessary to analyze this text. We call these words as “stop words”; Stop words do not add much value in a text. For example, “is”, “am”, “are” and “the” are few examples of stop words.

Solution: To remove the stop words, we can use a “filter” transformation which will return a new RDD containing only the elements that satisfy given condition(s). Lets apply “filter” transformation on “rdd2” and get words which are not stop words and get the result in “rdd3”. To do that:

1. We need to define the list of stop words in a variable called “stopwords” ( Here, I am selecting only a few words in stop words list instead of all the words).
2. Apply “filter” on “rdd2” (Check if individual words of “rdd2” are in the “stopwords” list or not ).

We can check first 10 elements of “rdd3” by applying take action.

stopwords = ['is','am','are','the','for','a']

rdd3 = rdd2.filter(lambda x: x not in stopwords)

rdd3.take(10)

Output:

[u'think',

u'of',

u'it',

u'moment',

u'\u2013',

u'1',

u'qunitillion',

u'=',

u'1',

u'million']

After seeing the result of a filter transformation, we can check now we don’t have specified stop words in rdd3 (there are no for and a).

### Transformation: groupBy

Q3: After getting the results into rdd3, we want to group the words in rdd3 based on which letters they start with. For example, suppose I want to group each word of rdd3 based on first 3 characters.

Solution: The “groupBy” transformation will group the data in the original RDD. It creates a set of key value pairs, where the key is output of a user function, and the value is all items for which the function yields this key.

1. We have to pass a function (in this case, I am using a lambda function) inside the “groupBy” which will take the first 3 characters of each word in “rdd3”.
2. The key is the first 3 characters and value is all the words which start with these 3 characters.

After applying “groupBy” function, we store the transformed result in “rdd4” (RDDs are immutable – remember!). To view “rdd4”, we can print first (key, value) elements in “rdd4”.

rdd4 = rdd3.groupBy(lambda w: w[0:3])

print [(k, list(v)) for (k, v) in rdd4.take(1)]

Output: [(u'all', [u'all', u'allocates', u'all', u'all', u'allows', u'all', u'all', u'all', u'all', u'all', u'all', u'all'])]

### Transformation: groupByKey / reduceByKey

Q4: What if we want to calculate how many times each word is coming in corpus ?

Solution: We can apply the “groupByKey” / “reduceByKey” transformations on (key,val) pair RDD. The “groupByKey” will group the values for each key in the original RDD. It will create a new pair, where the original key corresponds to this collected group of values.

To use “groupbyKey” / “reduceByKey” transformation to find the frequencies of each words, you can follow the steps below:

1. A (key,val) pair RDD is required; In this (key,val) pair RDD, key is the word and val is 1 for each word in RDD (1 represents the number for the each word in “rdd3”).
2. To apply “groupbyKey” / “reduceByKey” on “rdd3”, we need to first convert “rdd3” to (key,val) pair RDD.

Let’s see, how to convert “rdd3” to new mapped (key,val) RDD. And then we can apply “groupbyKey” / “reduceByKey” transformation on this RDD.

rdd3\_mapped = rdd3.map(lambda x: (x,1))

rdd3\_grouped = rdd3\_mapped.groupByKey()

In the above code I am first converting “rdd3” into “rdd3\_mapped”. The “rdd3\_mapped” is nothing but a mapped (key,val) pair RDD. Then I am applying “groupByKey” transformation on “rdd3\_mapped” to group the all elements based on the keys (words). Next, I am saving the result into “rdd3\_grouped”. Let’s see the first 5 elements in “rdd3\_grouped”.

print(list((j[0], list(j[1])) for j in rdd3\_grouped.take(5)))

Output: [(u'all', [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]), (u'elements,', [1, 1]), (u'step2:', [1]), (u'manager', [1]), (u'(if', [1])]

After seeing the result of the above code, I rechecked the corpus to know, how many times the word ‘manager’ is there, so I found that ‘manager’ is written more then once. I figure out that there are more words like ‘manager.’ , ‘manager,’ and ”manager:’. Let’s filter ‘manager,’ in “rdd3”.

rdd3.filter(lambda x: x == 'manager,').collect()

Output: [u'manager,', u'manager,', u'manager,']

We can see that in above output, we have multiple words with ‘manager’ in our corpus. To overcome this situation we can do several things. We could apply a regular expression to remove unnecessary punctuation from the words. For the purpose of this article, I am skipping that part.

Until now we have not calculated the frequencies / counts of each words. Let’s proceed further :

rdd3\_freq\_of\_words = rdd3\_grouped.mapValues(sum).map(lambda x: (x[1],x[0])).sortByKey(False)

In the above code, I first applied “mapValues” transformation on “rdd3\_grouped”. The “mapValues” (only applicable on pair RDD) transformation is like a map (can be applied on any RDD) transform but it has one difference that when we apply map transform on pair RDD we can access the key and value both of this RDD but in case of “mapValues” transformation, it will transform the values by applying some function and key will not be affected. So for example, in above code I applied sum, which will calculate the sum (counts) for the each word.

After applying “mapValues” transformation I want to sort the words based on their frequencies so for doing that I am first converting a ( word, frequency ) pair to ( frequency,word ) so that our key and values will be interchanged then, I will apply a sorting based on key and then get a result in “rdd3\_freq\_of\_words”. We can see that 10 most frequent words I used in my previous blog by applying “take” action.

rdd3\_freq\_of\_words.take(10)

output:

[(164, u'to'),

(143, u'in'),

(122, u'of'),

(106, u'and'),

(103, u'we'),

(69, u'spark'),

(64, u'this'),

(63, u'data'),

(55, u'can'),

(52, u'apache')]

Above output shows that I used words spark 69 times and Apache 52 times in my previous blog.

We can also use “reduceByKey” transformation for counting the frequencies of each word in (key,value) pair RDD. Lets see how will we do this.

rdd3\_mapped.reduceByKey(lambda x,y: x+y).map(lambda x:(x[1],x[0])).sortByKey(False).take(10)

output:

[(164, u'to'),

(143, u'in'),

(122, u'of'),

(106, u'and'),

(103, u'we'),

(69, u'spark'),

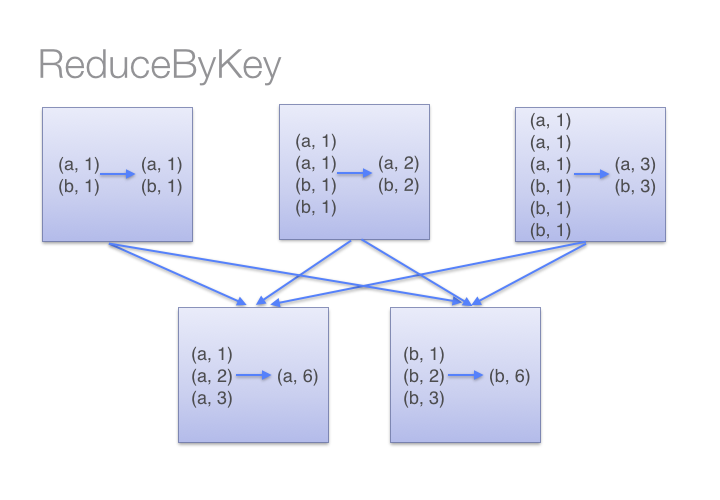
(64, u'this'),

(63, u'data'),

(55, u'can'),

(52, u'apache')]

If we compare the result of both ( “groupByKey” and “reduceByKey”) transformations, we have got the same results. I am sure you must be wondering what is the difference in both transformations. The “reduceByKey” transformations first combined the values for each key in all partition, so each partition will have only one value for a key then after shuffling, in reduce phase executors will apply operation for example, in my case sum(lambda x: x+y).

  
Source: Databricks

But in case of “groupByKey” transformation, it will not combine the values in each key in all partition it directly shuffle the data then merge the values for each key. Here in “groupByKey” transformation lot of shuffling in the data is required to get the answer, so it is better to use “reduceByKey” in case of large shuffling of data.

### 

Source: Databricks

### Transformation: mapPartitions

Q5: How do I perform a task (say count the words ‘spark’ and ‘apache’ in rdd3) separatly on each partition and get the output of the task performed in these partition ?  
Soltion: We can do this by applying “mapPartitions” transformation. The “mapPartitions” is like a map transformation but runs separately on different partitions of a RDD. So, for counting the frequencies of words ‘spark’ and ‘apache’ in each partition of RDD, you can follow the steps:

1. Create a function called “func” which will count the frequencies for these words
2. Then, pass the function defined in step1 to the “mapPartitions” transformation.

def func(iterator):

count\_spark = 0

count\_apache = 0

for i in iterator:

if i =='spark':

count\_spark = count\_spark + 1

if i == 'apache':

count\_apache = count\_apache + 1

return (count\_spark,count\_apache)

Lets apply above function called ‘func’ on each partition of rdd3.

rdd3.mapPartitions(func).glom().collect()

Output: [[49, 39], [20, 13]]

I have used the “glom” function which is very useful when we want to see the data insights for each partition of a RDD. So above result shows that 49,39 are the counts of ‘spark’, ‘apache’ in partition1 and 20,13 are the counts of ‘spark’, ‘apache’ in partition2. If we won’t use the “glom” function we won’t we able to see the results of each partition.

rdd3.mapPartitions(f).collect()

Output: [49, 39, 20, 13]

## Math / Statistical Transformation

### Transformation: sample

Q6: What if I want to work with samples instead of full data ?  
Soltion: “sample” transformation helps us in taking samples instead of working on full data. The sample method will return a new RDD, containing a statistical sample of the original RDD.  
We can pass the arguments insights as the sample operation:

1. “withReplacement = True” or False (to choose the sample with or without replacement)
2. “fraction = x” ( x= .4 means we want to choose 40% of data in “rdd” ) and “seed” for reproduce the results.

rdd3\_sampled = rdd3.sample(False, 0.4, 42)

print len(rdd3.collect()),len(rdd3\_sampled.collect())

Output: 4768 1895

We can see the above output, we have total 4768,1895 words in “rdd3” and “rdd3\_sampled”.

## Set Theory / Relational Transformation

### Transformation: union

Q 7: What if I want to create a RDD which contains all the elements (a.k.a. union) of two RDDs ?  
Solution: To do so, we can use “union” transformation on two RDDs. In Spark “union” transformation will return a new RDD by taking the union of two RDDs. Please note that duplicate items will not be removed in the new RDD. To illustrate this:

1. I am first going to create a two sample RDD ( say sample1, sample2 ) from the “rdd3” by taking 20% sample for each.
2. Apply a union transformation on sample1, sample2.

sample1 = rdd3.sample(False,0.2,42)

sample2 =rdd3.sample(False,0.2,42)

union\_of\_sample1\_sample2 = sample1.union(sample2)

print len(sample1.collect()), len(sample2.collect()),len(union\_of\_sample1\_sample2.collect())

Output: 914 914 1828

From the above output, we can see that the “sample1”, “sample2” both have 914 elements each. And in the “union\_of\_sample1\_sample2”, we have 1828 elements which shows that union operation didn’t remove the duplicate elements.

### Transformation: join

Q 8: If we want to join the two pair RDDs based on their key.  
Solution: The “join” transformation can help us join two pairs of RDDs based on their key. To show that:

1. First create the two sample (key,value) pair RDDs (“sample1”, “sample2”) from the “rdd3\_mapped” same as I did for “union” transformation
2. Apply a “join” transformation on “sample1”, “sample2”.

sample1 = rdd3\_mapped.sample(False,.2,42)

sample2 = rdd3\_mapped.sample(False,.2,42)

join\_on\_sample1\_sample2 = sample1.join(sample2)

join\_on\_sample1\_sample2.take(2)

Output: [(u'operations', (1, 1)), (u'operations', (1, 1))]

### Transformation: distinct

Q 9: How to calculate distinct elements in a RDD ?  
Solution: We can apply “distinct” transformation on RDD to get the distinct elements. Let’s see how many distinct words do we have in the “rdd3”.

rdd3\_distinct = rdd3.distinct()

len(rdd3\_distinct.collect())

Output: 1485

“rdd3\_distinct” will contain all the unique words / elements present in “rdd3”. We can also check that we have 1485 unique words in the “rdd3”.

## Data Structure / I/O Transformation

### Transformation: coalesce

Q 10: What if I want to reduce the number of partition of a RDD and get the result in a new RDD?  
Solution: We will use “coalesce” transformation here. To demonstrate that:

1. Let’s first check the number of partition in rdd3.

rdd3.getNumPartitions()

Output: 2

2. And now apply coalesce transformation on “rdd3” , get the results in “rdd3\_coalesce” and see the number of partitions.

rdd3\_coalesce = rdd3.coalesce(1)

rdd3\_coalesce.getNumPartitions()

Output: 1

In some previous examples of transformation I already used some of the actions on different RDDs for printing the result. For example,”take” to print the first n elements of a RDD , “getNumPartitions” to know how many partition a RDD has and “collect” to print all elements of RDD.

Now, I will take few more actions to demonstrate how we can get the results.

## General Actions

### Action: getNumPartitions

Q 11: How do I find out number of parition in RDD ?

Solution: With “getNumPartitions”, we can find out that how many partitions exist in our RDD. Let’s see how many partition our initial RDD ("rdd3") has.

rdd3.getNumPartitions() Output: 2

### Action: Reduce

Q 12: If I want to find out the sum the all numbers in a RDD.

Solution: To demonstrate this, I will:

1. First create a RDD from a list of number from (1,1000) called “num\_rdd”.
2. Use a reduce action and pass a function through it (lambda x,y: x+y).

A reduce action is use for aggregating all the elements of RDD by applying pairwise user function.

num\_rdd = sc.parallelize(range(1,1000))

num\_rdd.reduce(lambda x,y: x+y)

Output: 499500

In the code above, I first created a RDD(“num\_rdd”) from the list and then I applied a reduce action on it to sum all the numbers in “num\_rdd”.

## Mathematical / Statistical Actions

### Action: count

Q 13: Count the number of elements in RDD.

Solution: The count action will count the number of elements in RDD. To see that, let’s apply count action on “rdd3” to count the number of words in "rdd3".

rdd3.count() Output: 4768

### Action: max, min, sum, variance and stdev

To take the maximum, minimum, sum, variance and standard deviation of a RDD, we can apply “max”, “min”, “sum”, “variance” and “stdev” actions. Let’s take the maximum, minimum, sum, variance and standard deviation of “num\_rdd”.

num\_rdd.max(),num\_rdd.min(), num\_rdd.sum(),num\_rdd.variance(),num\_rdd.stdev()

Output: (999, 1, 499500, 83166.66666666667, 288.38631497813253)

# Complete Guide on DataFrame Operations in PySpark

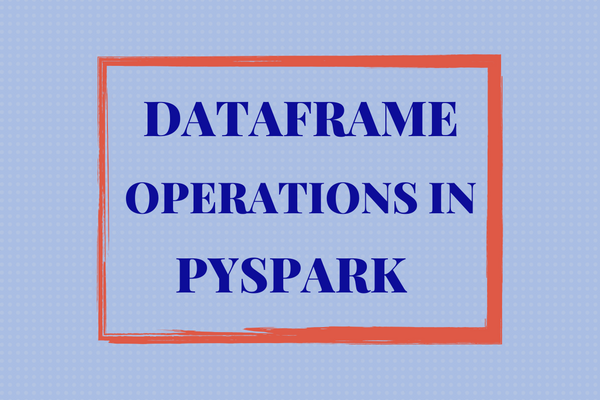
## Introduction

In my first [article](https://www.analyticsvidhya.com/blog/2016/09/comprehensive-introduction-to-apache-spark-rdds-dataframes-using-pyspark/), I introduced you to basic concepts of Apache Spark like how does it work, different cluster modes in Spark and What are the different data representation in Apache Spark. To provide you with a hands-on-experience, I also used a real world machine learning problem and then I solved it using PySpark.

In my second [article](https://www.analyticsvidhya.com/blog/2016/10/using-pyspark-to-perform-transformations-and-actions-on-rdd/), I introduced you on how to create RDD from different sources ( External, Existing ) and briefed you on basic operations ( Transformation and Action) on RDD.

In this article, I will be talking about DataFrame and its features in detail. Then, we will see how to create DataFrame from different sources and how to perform various operations in DataFrame.

P.S. – If you have not read the previous 2 articles, I strongly recommend that you go through them before going further.



## Table of Contents

1. Dataframes in PySpark: Overview
2. Why DataFrames are Useful
3. Setup of Apache Spark
4. How to create a DataFrame
   1. Creating DataFrame from RDD
   2. Creating DataFrame from CSV File
5. Dataframe Manipulations
6. Apply SQL queries on DataFrame
7. Pandas vs PySpark DataFrame

## 1. Dataframe in PySpark: Overview

In Apache Spark, a DataFrame is a distributed collection of rows under named columns. In simple terms, it is same as a table in relational database or an Excel sheet with Column headers. It also shares some common characteristics with RDD:

* Immutable in nature : We can create DataFrame / RDD once but can’t change it. And we can transform a DataFrame / RDD after applying transformations.
* Lazy Evaluations: Which means that a task is not executed until an action is performed.
* Distributed: RDD and DataFrame both are distributed in nature.

My first exposure to DataFrames was when I learnt about Pandas. Today, it is difficult for me to run my data science workflow with out Pandas DataFrames. So, when I saw similar functionality in Apache Spark, I was excited about the possibilities it opens up!

## 2. Why DataFrames are Useful ?

I am sure this question must be lingering in your mind. To make things simpler for you, I’m listing down few advantages of DataFrames:

* DataFrames are designed for processing large collection of structured or semi-structured data.
* Observations in Spark DataFrame are organised under named columns, which helps Apache Spark to understand the schema of a DataFrame. This helps Spark optimize execution plan on these queries.
* DataFrame in Apache Spark has the ability to handle petabytes of data.
* DataFrame has a support for wide range of data format and sources.
* It has API support for different languages like Python, R, Scala, Java.

## 3. Setup Apache Spark

In order to understand the operations of DataFrame, you need to first setup the Apache Spark in your machine. Follow the step by step approach mentioned in my previous [article](https://www.analyticsvidhya.com/blog/2016/09/comprehensive-introduction-to-apache-spark-rdds-dataframes-using-pyspark/), which will guide you to setup Apache Spark in Ubuntu.

DataFrame supports wide range of operations which are very useful while working with data. In this section, I will take you through some of the common operations on DataFrame.

First step, in any Apache programming is to create a SparkContext. SparkContext is required when we want to execute operations in a cluster. SparkContext tells Spark how and where to access a cluster. And the first step is to connect with Apache Cluster. If you are using Spark Shell, you will notice that it is already created. Otherwise, we can create the SparkContext by importing, initializing and providing the configuration settings. For example,

from pyspark import SparkContext

sc = SparkContext()

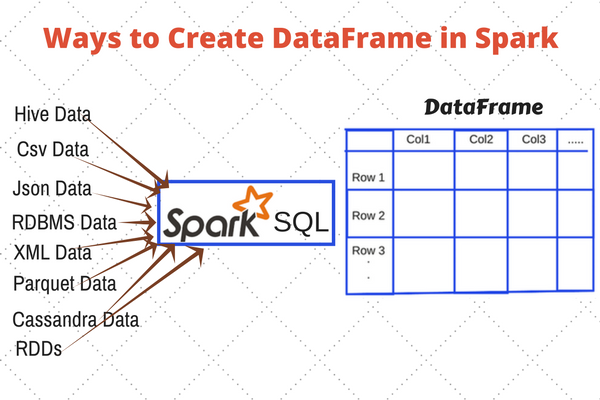
Again we need to do same with the SQLContext, if it is not loaded.

sqlContext = SQLContext(sc)

## 4. How to create a DataFrame ?

A DataFrame in Apache Spark can be created in multiple ways:

* It can be created using different data formats. For example, loading the data from JSON, CSV.
* Loading data from Existing RDD.
* Programmatically specifying schema



### Creating DataFrame from RDD

I am following these steps for creating a DataFrame from list of tuples:

* Create a list of tuples. Each tuple contains name of a person with age.
* Create a RDD from the list above.
* Convert each tuple to a row.
* Create a DataFrame by applying createDataFrame on RDD with the help of sqlContext.

from pyspark.sql import Row

l = [('Ankit',25),('Jalfaizy',22),('saurabh',20),('Bala',26)]

rdd = sc.parallelize(l)

people = rdd.map(lambda x: Row(name=x[0], age=int(x[1])))

schemaPeople = sqlContext.createDataFrame(people)

Lets check the type of schemaPeople.

type(schemaPeople)

Output:

pyspark.sql.dataframe.DataFrame

### Creating the DataFrame from CSV file

For reading a csv file in Apache Spark, we need to specify a new library in our python shell. To perform this action, first we need to download Spark-csv package (Latest version) and extract this package into the home directory of Spark. Then, we need to open a PySpark shell and include the package (I am using “spark-csv\_2.10:1.3.0”).

$ ./bin/pyspark --packages com.databricks:spark-csv\_2.10:1.3.0

Let’s read the data from csv file and create the DataFrame. To demonstrate this I’m to using the train and test datasets from the [Black Friday Practice Problem](https://datahack.analyticsvidhya.com/contest/black-friday/), which you can download [here](https://datahack.analyticsvidhya.com/contest/black-friday/).

train = sqlContext.load(source="com.databricks.spark.csv", path = 'PATH/train.csv', header = True,inferSchema = True)

test = sqlContext.load(source="com.databricks.spark.csv", path = 'PATH/test-comb.csv', header = True,inferSchema = True)

PATH is the location of folder, where your train and test csv files are located. Header is True, which means that the csv files contains the header. We are using inferSchema = True option for telling sqlContext to automatically detect the data type of each column in data frame. If we do not set inferSchema to be true, all columns will be read as string.

## 5. DataFrame Manipulations

Now comes the fun part. You have loaded the dataset by now. Let us start playing with it now.

### How to see datatype of columns?

To see the types of columns in DataFrame, we can use the printSchema, dtypes. Let’s apply printSchema() on train which will Print the schema in a tree format.

train.printSchema()

Output:

root

|-- User\_ID: integer (nullable = true)

|-- Product\_ID: string (nullable = true)

|-- Gender: string (nullable = true)

|-- Age: string (nullable = true)

|-- Occupation: integer (nullable = true)

|-- City\_Category: string (nullable = true)

|-- Stay\_In\_Current\_City\_Years: string (nullable = true)

|-- Marital\_Status: integer (nullable = true)

|-- Product\_Category\_1: integer (nullable = true)

|-- Product\_Category\_2: integer (nullable = true)

|-- Product\_Category\_3: integer (nullable = true)

|-- Purchase: integer (nullable = true)

From above output, we can see that, we have perfectly captured the schema / data types of each columns while reading from csv.

### How to Show first n observation?

We can use head operation to see first n observation (say, 5 observation). Head operation in PySpark is similar to head operation in Pandas.

train.head(5)

Output:

[Row(User\_ID=1000001, Product\_ID=u'P00069042', Gender=u'F', Age=u'0-17', Occupation=10, City\_Category=u'A', Stay\_In\_Current\_City\_Years=u'2', Marital\_Status=0, Product\_Category\_1=3, Product\_Category\_2=None, Product\_Category\_3=None, Purchase=8370),

Row(User\_ID=1000001, Product\_ID=u'P00248942', Gender=u'F', Age=u'0-17', Occupation=10, City\_Category=u'A', Stay\_In\_Current\_City\_Years=u'2', Marital\_Status=0, Product\_Category\_1=1, Product\_Category\_2=6, Product\_Category\_3=14, Purchase=15200),

Row(User\_ID=1000001, Product\_ID=u'P00087842', Gender=u'F', Age=u'0-17', Occupation=10, City\_Category=u'A', Stay\_In\_Current\_City\_Years=u'2', Marital\_Status=0, Product\_Category\_1=12, Product\_Category\_2=None, Product\_Category\_3=None, Purchase=1422),

Row(User\_ID=1000001, Product\_ID=u'P00085442', Gender=u'F', Age=u'0-17', Occupation=10, City\_Category=u'A', Stay\_In\_Current\_City\_Years=u'2', Marital\_Status=0, Product\_Category\_1=12, Product\_Category\_2=14, Product\_Category\_3=None, Purchase=1057),

Row(User\_ID=1000002, Product\_ID=u'P00285442', Gender=u'M', Age=u'55+', Occupation=16, City\_Category=u'C', Stay\_In\_Current\_City\_Years=u'4+', Marital\_Status=0, Product\_Category\_1=8, Product\_Category\_2=None, Product\_Category\_3=None, Purchase=7969)]

Above results are comprised of row like format. To see the result in more interactive manner (rows under the columns), we can use the show operation. Let’s apply show operation on train and take first 2 rows of it. We can pass the argument truncate = True to truncate the result.

train.show(2,truncate= True)

Output:

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

|User\_ID|Product\_ID|Gender| Age|Occupation|City\_Category|Stay\_In\_Current\_City\_Years|Marital\_Status|Product\_Category\_1|Product\_Category\_2|Product\_Category\_3|Purchase|

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

|1000001| P00069042| F|0-17| 10| A| 2| 0| 3| null| null| 8370|

|1000001| P00248942| F|0-17| 10| A| 2| 0| 1| 6| 14| 15200|

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

only showing top 2 rows

### How to Count the number of rows in DataFrame?

We can use count operation to count the number of rows in DataFrame. Let’s apply count operation on train & test files to count the number of rows.

train.count(),test.count()

Output:

(550068, 233599)

We have 550068, 233599 rows in train and test respectively.

### How many columns do we have in train and test files along with their names?

For getting the columns name we can use columns on DataFrame, similar to what we do for getting the columns in pandas DataFrame. Let’s first print the number of columns and columns name in train file then in test file.

len(train.columns), train.columns

OutPut:

12 ['User\_ID', 'Product\_ID', 'Gender', 'Age', 'Occupation', 'City\_Category', 'Stay\_In\_Current\_City\_Years', 'Marital\_Status', 'Product\_Category\_1', 'Product\_Category\_2', 'Product\_Category\_3', 'Purchase']

Lets do same for the test.

len(test.columns), test.columns

Output:

13 ['', 'User\_ID', 'Product\_ID', 'Gender', 'Age', 'Occupation', 'City\_Category', 'Stay\_In\_Current\_City\_Years', 'Marital\_Status', 'Product\_Category\_1', 'Product\_Category\_2', 'Product\_Category\_3', 'Comb']

From the above output we can check that we have 13 columns in test file and 12 in train file. “Purchase” not present in test file where as “Comb” is only in test file. We can also see that, we have one column (”) in test file which doesn’t have a name.

### How to get the summary statistics (mean, standard deviance, min ,max , count) of numerical columns in a DataFrame?

describe operation is use to calculate the summary statistics of numerical column(s) in DataFrame. If we don’t specify the name of columns it will calculate summary statistics for all numerical columns present in DataFrame.

train.describe().show()

Output:

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

|summary| User\_ID| Occupation| Marital\_Status|Product\_Category\_1|Product\_Category\_2|Product\_Category\_3| Purchase|

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

| count| 550068| 550068| 550068| 550068| 376430| 166821| 550068|

| mean|1003028.8424013031|8.076706879876669|0.40965298835780306| 5.404270017525106| 9.842329251122386|12.668243206790512| 9263.968712959126|

| stddev|1727.5915855308265|6.522660487341778| 0.4917701263173273|3.9362113692014082| 5.086589648693526| 4.125337631575267|5023.0653938206015|

| min| 1000001| 0| 0| 1| 2| 3| 12|

| max| 1006040| 20| 1| 20| 18| 18| 23961|

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

Let’s check what happens when we specify the name of a categorical / String columns in describe operation.

train.describe('Product\_ID').show()

Output:

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

|summary|Product\_ID|

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

| count| 550068|

| mean| null|

| stddev| null|

| min| P00000142|

| max| P0099942|

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

As we can see that, describe operation is working for String type column but the output for mean, stddev are null and min & max values are calculated based on ASCII value of categories.

### How to select column(s) from the DataFrame?

To subset the columns, we need to use select operation on DataFrame and we need to pass the columns names separated by commas inside select Operation. Let’s select first 5 rows of ‘User\_ID’ and ‘Age’ from the train.

train.select('User\_ID','Age').show(5)

Output:

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

|User\_ID| Age|

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

|1000001|0-17|

|1000001|0-17|

|1000001|0-17|

|1000001|0-17|

|1000002| 55+|

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

### How to find the number of distinct product in train and test files?

The distinct operation can be used here, to calculate the number of distinct rows in a DataFrame. Let’s apply distinct operation to calculate the number of distinct product in train and test file each.

train.select('Product\_ID').distinct().count(),test.select('Product\_ID').distinct().count()

Output:

(3631, 3491)

We have 3631 & 3491 distinct product in train & test file respectively. After counting the number of distinct values for train and test files, we can see the train file has more categories than test file. Let us check what are the categories for Product\_ID, which are in test file but not in train file by applying subtract operation.We can do the same for all categorical features.

diff\_cat\_in\_train\_test=test.select('Product\_ID').subtract(train.select('Product\_ID'))

diff\_cat\_in\_train\_test.distinct().count()# For distict count

Output:

46

Above, you can see that 46 different categories are in test file but not in train. In this case, either we collect more data about them or skip the rows in test file for those categories (invalid category) which are not in train file.

### What if I want to calculate pair wise frequency of categorical columns?

We can use crosstab operation on DataFrame to calculate the pair wise frequency of columns. Let’s apply crosstab operation on ‘Age’ and ‘Gender’ columns of train DataFrame.

train.crosstab('Age', 'Gender').show()

Output:

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

|Age\_Gender| F| M|

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

| 0-17| 5083| 10019|

| 46-50|13199| 32502|

| 18-25|24628| 75032|

| 36-45|27170| 82843|

| 55+| 5083| 16421|

| 51-55| 9894| 28607|

| 26-35|50752|168835|

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

In the above output, the first column of each row will be the distinct values of Age and the column names will be the distinct values of Gender. The name of the first column will be Age\_Gender. Pair with no occurrences will have zero count in contingency table.

### What If I want to get the DataFrame which won’t have duplicate rows of given DataFrame?

We can use dropDuplicates operation to drop the duplicate rows of a DataFrame and get the DataFrame which won’t have duplicate rows. To demonstrate that I am performing this on two columns Age and Gender of train and get the all unique rows for these columns.

train.select('Age','Gender').dropDuplicates().show()

Output:

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

| Age|Gender|

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

|51-55| F|

|51-55| M|

|26-35| F|

|26-35| M|

|36-45| F|

|36-45| M|

|46-50| F|

|46-50| M|

| 55+| F|

| 55+| M|

|18-25| F|

| 0-17| F|

|18-25| M|

| 0-17| M|

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

### What if I want to drop the all rows with null value?

The dropna operation can be use here. To drop row from the DataFrame it consider three options.

* how– ‘any’ or ‘all’. If ‘any’, drop a row if it contains any nulls. If ‘all’, drop a row only if all its values are null.
* thresh – int, default None If specified, drop rows that have less than thresh non-null values. This overwrites the how parameter.
* subset – optional list of column names to consider.

Let’t drop null rows in train with default parameters and count the rows in output DataFrame. Default options are any, None, None for how, thresh, subset respectively.

train.dropna().count()

Output:

166821

### What if I want to fill the null values in DataFrame with constant number?

Use fillna operation here. The fillna will take two parameters to fill the null values.

* value:
  + It will take a dictionary to specify which column will replace with which value.
  + A value (int , float, string) for all columns.
* subset: Specify some selected columns.

Let’s fill ‘-1’ inplace of null values in train DataFrame.

train.fillna(-1).show(2)

Output:

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

|User\_ID|Product\_ID|Gender| Age|Occupation|City\_Category|Stay\_In\_Current\_City\_Years|Marital\_Status|Product\_Category\_1|Product\_Category\_2|Product\_Category\_3|Purchase|

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

|1000001| P00069042| F|0-17| 10| A| 2| 0| 3| -1| -1| 8370|

|1000001| P00248942| F|0-17| 10| A| 2| 0| 1| 6| 14| 15200|

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

only showing top 2 rows

### If I want to filter the rows in train which has Purchase more than 15000?

We can apply the filter operation on Purchase column in train DataFrame to filter out the rows with values more than 15000. We need to pass a condition. Let’s apply filter on Purchase column in train DataFrame and print the number of rows which has more purchase than 15000.

train.filter(train.Purchase > 15000).count()

Output:

110523

### How to find the mean of each age group in train?

The groupby operation can be used here to find the mean of Purchase for each age group in train. Let’s see how can we get the mean purchase for the ‘Age’ column train.

train.groupby('Age').agg({'Purchase': 'mean'}).show()

Output:

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

| Age| avg(Purchase)|

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

|51-55|9534.808030960236|

|46-50|9208.625697468327|

| 0-17|8933.464640444974|

|36-45|9331.350694917874|

|26-35|9252.690632869888|

| 55+|9336.280459449405|

|18-25|9169.663606261289|

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

We can also apply sum, min, max, count with groupby when we want to get different summary insight each group. Let’s take one more example of groupby to count the number of rows in each Age group.

train.groupby('Age').count().show()

Output:

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

| Age| count|

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

|51-55| 38501|

|46-50| 45701|

| 0-17| 15102|

|36-45|110013|

|26-35|219587|

| 55+| 21504|

|18-25| 99660|

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

### How to create a sample DataFrame from the base DataFrame?

We can use sample operation to take sample of a DataFrame. The sample method on DataFrame will return a DataFrame containing the sample of base DataFrame. The sample method will take 3 parameters.

* withReplacement = True or False to select a observation with or without replacement.
* fraction = x, where x = .5 shows that we want to have 50% data in sample DataFrame.
* seed for reproduce the result

Let’s create the two DataFrame t1 and t2 from train, both will have 20% sample of train and count the number of rows in each.

t1 = train.sample(False, 0.2, 42)

t2 = train.sample(False, 0.2, 43)

t1.count(),t2.count()

Output:

(109812, 109745)

### How to apply map operation on DataFrame columns?

We can apply a function on each row of DataFrame using map operation. After applying this function, we get the result in the form of RDD. Let’s apply a map operation on User\_ID column of train and print the first 5 elements of mapped RDD(x,1) after applying the function (I am applying lambda function).

train.select('User\_ID').map(lambda x:(x,1)).take(5)

Output:

[(Row(User\_ID=1000001), 1),

(Row(User\_ID=1000001), 1),

(Row(User\_ID=1000001), 1),

(Row(User\_ID=1000001), 1),

(Row(User\_ID=1000002), 1)]

In above code we have passed lambda function in the map operation which will take each row / element of ‘User\_ID’ one by one and return pair for them (‘User\_ID’,1).

### How to sort the DataFrame based on column(s)?

We can use orderBy operation on DataFrame to get sorted output based on some column. The orderBy operation take two arguments.

* List of columns.
* ascending = True or False for getting the results in ascending or descending order(list in case of more than two columns )

Let’s sort the train DataFrame based on ‘Purchase’.

train.orderBy(train.Purchase.desc()).show(5)

Output:

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

|User\_ID|Product\_ID|Gender| Age|Occupation|City\_Category|Stay\_In\_Current\_City\_Years|Marital\_Status|Product\_Category\_1|Product\_Category\_2|Product\_Category\_3|Purchase|

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

|1003160| P00052842| M|26-35| 17| C| 3| 0| 10| 15| null| 23961|

|1002272| P00052842| M|26-35| 0| C| 1| 0| 10| 15| null| 23961|

|1001474| P00052842| M|26-35| 4| A| 2| 1| 10| 15| null| 23961|

|1005848| P00119342| M|51-55| 20| A| 0| 1| 10| 13| null| 23960|

|1005596| P00117642| M|36-45| 12| B| 1| 0| 10| 16| null| 23960|

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

only showing top 5 rows

### How to add the new column in DataFrame?

We can use withColumn operation to add new column (we can also replace) in base DataFrame and return a new DataFrame. The withColumn operation will take 2 parameters.

* Column name which we want add /replace.
* Expression on column.

Let’s see how withColumn works. I am calculating new column name ‘Purchase\_new’ in train which is calculated by dviding Purchase column by 2.

train.withColumn('Purchase\_new', train.Purchase /2.0).select('Purchase','Purchase\_new').show(5)

Output:

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

|Purchase|Purchase\_new|

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

| 8370| 4185.0|

| 15200| 7600.0|

| 1422| 711.0|

| 1057| 528.5|

| 7969| 3984.5|

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

only showing top 5 rows

### How to drop a column in DataFrame?

To drop a column from the DataFrame we can use drop operation. Let’s drop the column called ‘Comb’ from the test and get the remaining columns in test.

test.drop('Comb').columns

Output:

['',

'User\_ID',

'Product\_ID',

'Gender',

'Age',

'Occupation',

'City\_Category',

'Stay\_In\_Current\_City\_Years',

'Marital\_Status',

'Product\_Category\_1',

'Product\_Category\_2',

'Product\_Category\_3']

### What if I want to remove some categories of Product\_ID column in test that are not present in Product\_ID column in train?

Here, we can use a user defined function ( udf ) to remove the categories of a column which are in test but not in train. Let’s again calculate the categories in Product\_ID column which are in test but not in train.

diff\_cat\_in\_train\_test=test.select('Product\_ID').subtract(train.select('Product\_ID'))

diff\_cat\_in\_train\_test.distinct().count()# For distict count

Output:

46

We have got 46 different categories in test. For removing these categories from the test ‘Product\_ID’ column. I am applying these steps.

* Create the distinct list of categories called ‘not\_found\_cat’ from the diff\_cat\_in\_train\_test using map operation.
* Register a udf(user define function).
* User defined function will take each element of test column and search this in not\_found\_cat list and it will put -1 if it finds in this list otherwise it will do nothing.

Let’s see how it works. First create ‘not\_found\_cat’

not\_found\_cat = diff\_cat\_in\_train\_test.distinct().rdd.map(lambda x: x[0]).collect()

len(not\_found\_cat)

Output:

46

Now resister the udf, we need to import StringType from the pyspark.sql and udf from the pyspark.sql.functions. The udf function takes 2 parameters as arguments:

* Function (I am using lambda function)
* Return type (in my case StringType())

from pyspark.sql.types import StringType

from pyspark.sql.functions import udf

F1 = udf(lambda x: '-1' if x in not\_found\_cat else x, StringType())

In the above code function name is ‘F1’ and we are putting ‘-1’ for not found catagories in test ‘Product\_ID’. Finally apply above ‘F1’ function on test ‘Product\_ID’ and take result in k1 for new column calles “NEW\_Product\_ID”.

k = test.withColumn("NEW\_Product\_ID",F1(test["Product\_ID"])).select('NEW\_Product\_ID')

Now, let’s see the results by again calculating the different categories in k and train subtract operation.

diff\_cat\_in\_train\_test=k.select('NEW\_Product\_ID').subtract(train.select('Product\_ID'))

diff\_cat\_in\_train\_test.distinct().count()# For distinct count

Output:

1

The output 1 means we have now only 1 different category k and train.

diff\_cat\_in\_train\_test.distinct().collect()

Output:

Row(NEW\_Product\_ID=u'-1')

## 6. How to Apply SQL Queries on DataFrame?

We have already discussed in the above section that DataFrame has additional information about datatypes and names of columns associated with it. Unlike RDD, this additional information allows Spark to run SQL queries on DataFrame. To apply SQL queries on DataFrame first we need to register DataFrame as table. Let’s first register train DataFrame as table.

train.registerAsTable('train\_table')

In the above code, we have registered ‘train’ as table(‘train\_table’) with the help of registerAsTable operation. Let’s apply SQL queries on ‘train\_table’ to select Product\_ID the result of SQL query will be a DataFrame. We need to apply a action to get the result.

sqlContext.sql('select Product\_ID from train\_table').show(5)

Output:

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

|Product\_ID|

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

| P00069042|

| P00248942|

| P00087842|

| P00085442|

| P00285442|

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

In the above code, I am using sqlContext.sql for specifying SQL query.

Let’s get maximum purchase of each Age group in train\_table.

sqlContext.sql('select Age, max(Purchase) from train\_table group by Age').show()

Output:

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

| Age| \_c1|

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

|51-55|23960|

|46-50|23960|

| 0-17|23955|

|36-45|23960|

|26-35|23961|

| 55+|23960|

|18-25|23958|

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

## 7. Pandas vs PySpark DataFrame

Pandas and Spark DataFrame are designed for structural and semistructral data processing. Both share some similar properties (which I have discussed above). The few differences between Pandas and PySpark DataFrame are:

* Operation on Pyspark DataFrame run parallel on different nodes in cluster but, in case of pandas it is not possible.
* Operations in PySpark DataFrame are lazy in nature but, in case of pandas we get the result as soon as we apply any operation.
* In PySpark DataFrame, we can’t change the DataFrame due to it’s immutable property, we need to transform it. But in pandas it is not the case.
* Pandas API support more operations than PySpark DataFrame. Still pandas API is more powerful than Spark.
* Complex operations in pandas are easier to perform than Pyspark DataFrame

In addition to above points, Pandas and Pyspark DataFrame have some basic differences like columns selection, filtering, adding the columns, etc. which I am not covering here.