**CSE 4262 Data Analytics Lab**

**Lab 1: Familiarization with Spark frameworks and learning the fundamentals of PySpark**

Outcomes: After this lab students will be able to:

* Explain the fundamentals concepts of PySpark including Spark Session vs SparkContext, Dataframe, and Resilient Distributed Dataset (RDD).
* Apply Dataframe and RDD to solve practical problems.

**What is Apache Spark?**

Apache Spark is a distributed processing system used to perform big data and machine learning tasks on large datasets. Distributed processing is a setup in which multiple processors are used to run an application. Instead of trying to process large datasets on a single computer, the task can be divided between multiple devices that communicate with each other. **With Apache Spark, users can run queries and machine learning workflows on petabytes of data, which is impossible to do on your local device.** The other good thing about Spark is that the programs that we write are much smaller than the typical MapReduce classes that we write for Hadoop. So, not only do our programs run faster but it also takes less time to write them.

Spark has **four major higher-level tools built on top of the Spark Core: Spark Streaming, Spark MLlib (machine learning), Spark SQL (an SQL interface for accessing the data), and GraphX (for graph processing). The Spark Core is the heart of Spark**. Spark provides higher-level abstractions in Scala, Java, and Python for data representation, serialization, scheduling, metrics, and so on. Figure 1 shows the details of Spark architecture.

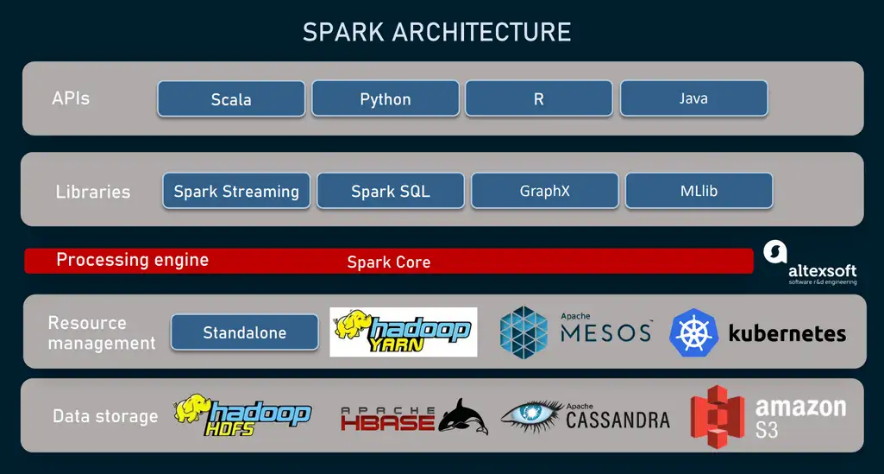


Figure 1: Spark Architecture

**What is PySpark?**

PySpark is an interface for Apache Spark in Python. With PySpark, you can write Python and SQL-like commands to manipulate and analyze data in a distributed processing environment. Most data scientists and analysts are familiar with Python and use it to implement machine learning workflows. PySpark allows them to work with a familiar language on large-scale distributed datasets.

Companies that collect terabytes of data will have a big data framework like Apache Spark in place. To work with these large-scale datasets, knowledge of Python and R frameworks alone will not suffice. The reason companies choose to use a framework like PySpark is because of how quickly it can process big data**. It is faster than libraries like Pandas and Dask and can handle larger amounts of data than these frameworks.** **If you had over petabytes of data to process, for instance, Pandas and Dask would fail but PySpark would be able to handle it easily.**

**Task1: Understand and run a simple spark program**

Step1: Create a SparkSession

SparkSession is the unified entry point to use all the features of Apache Spark, including Spark SQL, DataFrame API, and Dataset API. It was introduced in Spark 2.0 as a higher-level API than SparkContext and SQLContext and provides a simplified, user-friendly interface for interacting with Spark.

Add the following codes into the cell and execute:

from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Datacamp Pyspark Tutorial").master("local[\*]").config("spark.memory.offHeap.enabled","true").config("spark.memory.offHeap.size","10g").getOrCreate()

The above code creates a SparkSession object using the builder design pattern from pyspark. The spark-shell/pyspark creates the SparkSession object automatically and assigns it to the spark variable. The SparkSession object has the SparkContext object, which you can access with spark.sparkContext. A SparkSession’s metadata are

* The appName parameter sets the name of the application to "Datacamp Pyspark Tutorial".
* This value is the name of the master. Using local as the default value for the master makes it easy to launch your application in a test environment locally. 'local[\*]' in the case of a fully local setup or 'yarn' in the case of a proper Hadoop cluster.
* The two config parameters enable off-heap memory and set the size of the off-heap memory to 10 gigabytes.
* Finally, the getOrCreate() method returns an existing SparkSession or creates a new one if none exists.

**Q/A-1: Write the Difference Between SparkSession and SparkContext.**

**SparkContext**, introduced in Spark 1.x, is the primary entry point for Spark functionality and is used for lower-level operations like creating RDDs. On the other hand, **SparkSession**, introduced in Spark 2.0, is a new entry point that simplifies and unifies the Spark user interface, particularly for structured data processing. While SparkContext is used for RDD manipulation, SparkSession provides a simplified interface for structured data processing.

Step2: Create a Dataframe

Spark abstractions: The main features of Apache Spark are distributed data representation and computation, thus achieving massive scaling of data operations. Spark's primary unit for representation of data is RDD, which allows for easy parallel operations on the data. Until 2.0.0, everyone worked with RDDs. However, they are low-level raw structures, which can be optimized for performance and scalability.

This is where Datasets/DataFrames come into the picture. Datasets/DataFrames are API-level abstractions, that is, the main programming interface. They provide most of the RDD operations but are layered over RDDs via optimized query plans. So, the underlying representation is still an RDD but accessed via the Dataset/DataFrame APIs. RDDs can be viewed as arrays of arrays with primitive data types, such as integers, floats, and strings. Datasets/DataFrames, on the other hand, are similar to a table or a spreadsheet with column headings such as name, title, order number, order date, and movie rating and the associated data types.

Datasets/DataFrames and RDDs can be converted back and forth using dataset.rdd() and SparkSession.createDataset(rdd)/SparkSession.createDataFrame(rdd).

Add the following codes into the cell and execute:

df = spark.read.csv('c:\spark\Datacamp\_Ecommerce.csv',header=True,escape="\"")

df.show(5,0)

* The above code reads a CSV file named "datacamp\_ecommerce.csv" into a Spark DataFrame called "df".
* The header=True argument indicates that the first row of the CSV file contains the column names.
* The escape="\"" argument specifies that the backslash character should be used as the escape character for any special characters in the CSV file.
* The spark object is assumed to be a SparkSession object that has been previously created.
* the show () method to display the first 5 rows of a DataFrame df.
* The second argument 0 specifies that the method should not truncate the displayed columns.
* This is useful when working with large datasets where the default behavior is to truncate the columns to fit the screen.
* By setting the second argument to 0, all columns will be displayed without truncation.

**Q/A-2: Write the column names of the "datacamp\_ecommerce.csv". How many rows are in the csv file?**

**Columns :**

**InvoiceNo, StockCode, Description, Quantity, InvoiceDate, UnitPrice, CustomerID, Country**

**Total Rows : 541909**

**Task2: Create Dataframes from different data sources**

**Create Dataframes**

1. my\_list = [['a', 1, 2], ['b', 2, 3],['c', 3, 4]]

col\_name = ['A', 'B', 'C']

spark.createDataFrame(my\_list, col\_name).show()

1. import pandas as pd

df=pd.DataFrame(my\_list,columns= col\_name)

print(df.to\_string()+"\n")

Now change the pd.DataFrame() by the following

1. df=pd.DataFrame(my\_list,col\_name)

**Q/A-3: Write the difference between rdd.DataFrame vs pd.DataFrame and compare their outputs.**

When **spark.createDataFrame(my\_list, col\_name)** is used, the DataFrame is presented in a tabular layout with specified column names, resembling how data is organized in a database or spreadsheet.

Similarly, **pd.DataFrame(my\_list, columns=col\_name)** in Pandas also arranges the DataFrame in a tabular form with the provided column names, following the usual behavior of Pandas DataFrames.

**Create Dataframe from Pandas Dataframe**

import pandas as pd

# df = spark.createDataFrame(pandas\_df.toPandas())

# Creating pandas dataframe first

l = [["2015-06-23", 5]

,["2016-07-20", 7]] #List with data elements

df = spark.createDataFrame(pd.DataFrame(l),['data\_date','months\_to\_add'])

df.show()

**Now change the above Panda’s dataframe (df) to spark’s dataframe.**

**Create Dataframe from RDD**

# Import spark libraries

from pyspark.sql import Row, DataFrame

from pyspark.sql.types import StringType, StructType, StructField, IntegerType

# create RDD to load into spark dataframe

l = [["2015-06-23", 5] , ["2016-07-20", 7]] #List with data elements

rdd1 = spark.sparkContext.parallelize(l)

row\_rdd = rdd1.map(lambda x: Row(x[0], x[1]))

df = spark.createDataFrame(row\_rdd, ['data\_date', 'months\_to\_add'])

df.show()

**Create Dataframe from RDD and Schema**

# Import spark libraries

from pyspark.sql import Row, DataFrame

from pyspark.sql.types import StringType, StructType, StructField, IntegerType

# create RDD to load into spark dataframe

l = [["2015-06-23", 5],["2016-07-20", 7]] #List with data elements

rdd1 = spark.sparkContext.parallelize(l)

schema = StructType([ StructField("data\_date", StringType(), True),

StructField("months\_to\_add", IntegerType(), True)]) # Col, Type, Nullable

df = spark.createDataFrame(rdd1, schema)

df.show()

**Q/A-4: Explain the procedure to define the schema.**

**1. Import Necessary Libraries:**

Import the required libraries from PySpark:

*from pyspark.sql.types import StringType, StructType, StructField, IntegerType*

**2. Define the Schema:**

Create a schema using the `StructType` class, which represents the schema of a DataFrame. You can specify the schema by defining the structure of each column using `StructField`.

From above example:

- `StructField` is used to define each column of the schema.

- The first argument specifies the name of the column.

- The second argument specifies the data type of the column (`StringType()` for string data, `IntegerType()` for integer data, etc.).

- The third argument (`True`) indicates whether the column allows null values or not. If set to `True`, the column is nullable; if set to `False`, it's not nullable.

**3. Create DataFrame with Schema:**

Once the schema is defined, you can create a DataFrame using the `createDataFrame` method, passing the RDD containing the data and the schema as parameters.

***df = spark.createDataFrame(rdd1, schema)***

This will create a DataFrame (`df`) with the specified schema (`schema`) and the data from the RDD (`rdd1`).

By following these steps, a schema can be defined for Spark DataFrame, which allows for more structured and efficient data processing and analysis.

RDD is the primary low-level abstraction in Spark. The main programming APIs will be Datasets/DataFrames. However, underneath it all, the data will be represented as RDDs. So, understanding and working with RDDs is important. From a structural view, RDDs are just a bunch of elements that can be operated in parallel. RDD stands for Resilient Distributed Dataset, that is, it is distributed over a set of machines, and the transformations are captured so that an RDD can be recreated in case there is a machine failure or memory corruption. One important aspect of the distributed parallel data representation scheme is that RDDs are immutable, which means when you do an operation, it generates a new RDD.

From a modality perspective, all data can be grouped into three categories: structured, semi-structured, and unstructured. The modality is independent of the data source, organization, or storage technologies. Before 2.0.0, things were conceptually simpler-we only needed to read data into RDDs and use map () to transform the data as required. However, data wrangling was harder. With Dataset/DataFrame, we can read directly into a table with headings, associate data types with domain semantics, and start working with data more effectively.

As a general rule of thumb, perform the following steps:

1. Use SparkContext and RDDs to handle unstructured data.
2. Use SparkSession and Datasets/DataFrames for semi-structured and structured data. As you will see in the later chapters, SparkSession has unified the read from various formats, such as the .csv, .json, .parquet, .jdbc, .orc, and .text files. Moreover, there is a pluggable architecture called DataSource API to access any type of structured data.

**Creation of RDD**

* **using parallelize():** parallelize() method is used to create an RDD from a list.

dataList = [("Java", 20000), ("Python", 100000), ("Scala", 3000)]

rdd=spark.sparkContext.parallelize(dataList)

rdd.collect()

* **using textFile():** RDD can also be created from a text file using textFile() function of the SparkContext.

rdd2 = spark.sparkContext.textFile("C:\spark\text1.txt")

df1=spark.read.text("/text1.txt")

df1.show()

**Task3: Convert PySpark RDD to DataFrame**

First, create an RDD by the following code snippet.

# importing necessary libraries

from pyspark.sql import SparkSession

# function to create new SparkSession

def create\_session():

spk = SparkSession.builder \

.appName("Corona\_cases\_statewise.com") \

.getOrCreate()

return spk

# function to create RDD

def create\_RDD(sc\_obj, data):

df = sc.parallelize(data)

return df

# function to convert RDD to dataframe

def RDD\_to\_df(df,schema):

# converting RDD to dataframe using toDF() in which we are passing schema of df

df3 = rd\_df.toDF(schema)

return df3

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

input\_data = [("Dhaka", 122000, 89600, 12238), ("Khulna", 454000, 380000, 67985),

("Rangpur", 115000, 102000, 13933),("Bogra", 147000, 111000, 15306),

("Rajshahi", 153000, 124000, 5259)]

# calling function to create SparkSession

spark = create\_session()

# creating spark context object

sc = spark.sparkContext

# calling function to create RDD

rd\_df = create\_RDD(sc, input\_data)

# printing the type

print(type(rd\_df))

**Q/A-5: Execute the above code and write the outputs.**

**Output:** <class 'pyspark.rdd.RDD'>

**There are two approaches to converting RDD to dataframe.**

* Using createDataframe(rdd, schema)

Modify the above RDD codes by adding the following statements:

1. Add the following function

# function to convert RDD to dataframe

def RDD\_to\_df(spark,df,schema):

# converting RDD to df using createDataframe()

# in which we are passing RDD and schema of df

df1 = spark.createDataFrame(df,schema)

return df1

1. Add the following statements after the RDD codes

schema\_lst = ["State","Cases","Recovered","Deaths"]

# calling function to convert RDD to dataframe

converted\_df = RDD\_to\_df(rd\_df,schema\_lst)

# visualizing the schema and dataframe

converted\_df.printSchema()

converted\_df.show()

* Using toDF(schema)

Modify the above RDD codes by adding the following statements:

1. Add the following function

# function to convert RDD to dataframe

def RDD\_to\_df(df,schema):

# converting RDD to dataframe using toDF()

# in which we are passing schema of df

df = rd\_df.toDF(schema)

return df

1. Add the following statements after the RDD codes

schema\_lst = ["State","Cases","Recovered","Deaths"]

# calling function to convert RDD to dataframe

converted\_df = RDD\_to\_df(rd\_df,schema\_lst)

# visualizing the schema and dataframe

converted\_df.printSchema()

converted\_df.show()

**Q/A-6: Write the difference between the above two approaches.**

**1. createDataFrame() Method:**

- In this approach, `createDataFrame()` method of the `SparkSession` object, passing the RDD and the schema as parameters.

- This method creates a DataFrame directly from the RDD and the specified schema.

- It requires passing both the RDD and the schema as arguments.

**2. toDF() Method:**

- In this approach, you call the `toDF()` method directly on the RDD, passing only the schema as a parameter.

- This method implicitly converts the RDD to a DataFrame using the specified schema.

- It requires passing only the schema as an argument, as the RDD itself is used to infer the structure of the DataFrame.

Once you have an RDD, you can perform transformation and action operations. Any operation you perform on RDD runs in parallel.

RDD Transformations: Transformations on Spark RDD return another RDD and transformations are lazy meaning they don’t execute until you call an action on RDD. Some transformations on RDDs are flatMap(), map(), reduceByKey(), filter(), sortByKey() and return a new RDD instead of updating the current.

How the map() transformation works

The map() transformation in PySpark is used to apply a function to each element in a dataset. This function takes a single element as input and returns a transformed element as output. The map() transformation returns a new dataset that consists of the transformed elements.

rdd.map(map\_function)

data = [1, 2, 3, 4]

rdd = spark.sparkContext.parallelize(data)

rdd\_transformed = rdd.map(lambda x: x \* 2)

rdd\_transformed.collect()

**Task4: Execute the following code**

from pyspark.sql import SparkSession

# Create a SparkSession

spark = SparkSession.builder.appName("map\_example").getOrCreate()

# Create a DataFrame with sample data

data = [("Alice", 1), ("Bob", 2), ("Charlie", 3)]

df = spark.createDataFrame(data, ["name", "age"])

# Define a function to be applied to each row

def add\_one(age):

return age + 1

# Use the map() transformation to apply

# the function to the "age" column

df = df.rdd.map(lambda x: (x[0], add\_one(x[1]))).toDF(["name", "age"])

# Show the resulting DataFrame

df.show()

How the flatmap() transformation works

PySpark flatMap() is a transformation operation that flattens the RDD/DataFrame (array/map DataFrame columns) after applying the function on every element and returns a new PySpark RDD/DataFrame.

**Task5: Execute the following code**

data = ["Project Gutenberg’s",

"Alice’s Adventures in Wonderland",

"Project Gutenberg’s",

"Adventures in Wonderland",

"Project Gutenberg’s"]

rdd=spark.sparkContext.parallelize(data)

for element in rdd.collect():

print(element)

Now add the following statements to the above snippet and check the output.

rdd2=rdd.flatMap(lambda x: x.split(" "))

for element in rdd2.collect():

print(element)

#First, it splits each record by space in an RDD and flattens it. The resulting RDD consists of a single word on each record.

Unfortunately, PySpark DataFame doesn’t have flatMap() transformation however, DataFrame has explode() SQL function that is used to flatten the column. Execute the below program.

import pyspark

from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('pyspark-by-examples').getOrCreate()

arrayData = [

('James',['Java','Scala'],{'hair':'black','eye':'brown'}),

('Michael',['Spark','Java',None],{'hair':'brown','eye':None}),

('Robert',['CSharp',''],{'hair':'red','eye':''}),

('Washington',None,None),

('Jefferson',['1','2'],{})]

df = spark.createDataFrame(data=arrayData, schema = ['name','knownLanguages','properties'])

from pyspark.sql.functions import explode

df2 = df.select(df.name,explode(df.knownLanguages))

df2.printSchema()

df2.show()

How the filter() transformation works

df.filter(condition) where df is the dataframe from which the data is subset or filtered.

Sometimes while dealing with a big dataframe that consists of multiple rows and columns we have to filter the dataframe, or we want the subset of the dataframe for applying operations according to our needs. For getting a subset or filter the data sometimes it is not sufficient with only a single condition many times we have to pass multiple conditions to filter or get the subset of that dataframe.

**Task 6: Create the following dataframe and add the filters.**

# importing necessary libraries

from pyspark.sql import SparkSession

# function to create SparkSession

def create\_session():

spk = SparkSession.builder .master("local").appName("Student\_report.com").getOrCreate()

return spk

def create\_df(spark, data, schema):

df1 = spark.createDataFrame(data, schema)

return df1

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

# calling function to create SparkSession

spark = create\_session()

input\_data = [(1, "Shivansh", "Male", 20, 80),

(2, "Arpy", "Female", 18, 66), (3, "Raj", "Male", 21, 90),

(4, "Shima", "Female", 19, 91), (5, "Apurba", "Male", 20, 50),

(6, "Swapnil", "Male", 23, 65), (7, "Rajesh", "Male", 19, 70)]

schema = ["Id", "Name", "Gender", "Age", "Percentage"]

# calling function to create dataframe

df = create\_df(spark, input\_data, schema)

df.show()

We can pass the multiple conditions into the function in two ways:

* Using double quotes (“conditions”)

# subset or filter the dataframe by

# passing Multiple condition

df = df.filter("Gender == 'Male' and Percentage>70")

df.show()

# subset or filter the data with

# multiple condition

df = df.filter("Age>20 or Percentage>80")

df.show()

* Using dot notation in condition

# subset or filter the data with

# multiple condition

df = df.filter((df.Gender=='Male') | (df.Percentage>90))

df.show()

# subset or filter the dataframe by

# passing Multiple condition

df = df.filter((df.Gender=='Female') & (df.Age>=18))

df.show()

RDD Actions

RDD Action operation returns the values from an RDD to a driver node. In other words, any RDD function that returns non RDD[T] is considered as an action. Some actions on RDDs are count(), collect(), first(), max(), reduce() and more.

# importing module

import pyspark

from pyspark.sql import SparkSession

# creating sparksession and giving an app name

spark = SparkSession.builder.appName('sparkdf').getOrCreate()

data = [["1", "Sourav", "IT", 45000], ["2", "Aditi", "IT", 30000], ["3", "Bobby", "business", 45000],

["4", "Rohit", "IT", 45000], ["5", "Shamim", "business", 120000],

["6", "Moni", "sales", 23000], ["7", "Borno", "sales", 34000],

["8", "Shreya", "business", 456798], ["9", "Kishor", "IT", 230000],

["10", "Ron", "business", 100000] ]

# specify column names

columns = ['ID', 'NAME', 'sector', 'salary']

dataframe = spark.createDataFrame(data, columns)

# display dataframe

dataframe.show()

Using select(), where(), count()

**Task7: Execute the following code**

First Execute the following code and create a dataframe.

import pyspark

from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('sparkdf').getOrCreate()

# list of employee data with 10 row values

data = [["1", "Sourav", "IT", 45000], ["2", "Aditi", "IT", 30000], ["3", "Bobby", "business", 45000],

["4", "Rohit", "IT", 45000], ["5", "Shamim", "business", 120000],

["6", "Moni", "sales", 23000], ["7", "Borno", "sales", 34000],

["8", "Shreya", "business", 456798], ["9", "Kishor", "IT", 230000],

["10", "Ron", "business", 100000] ]

# specify column names

columns = ['ID', 'NAME', 'sector', 'salary']

# creating a dataframe from the lists of data

dataframe = spark.createDataFrame(data, columns)

# display dataframe

dataframe.show()

* Add the following code to count values in the NAME column where ID is greater than 5

# count values in the NAME column

# where ID greater than 5

dataframe.select('NAME').where(dataframe.ID>5).count()

**Now execute the following tasks and write the results in the following box:**

* count ID column where ID =4
* count ID column where ID > 4 and sector is sales or IT

1. dataframe.select('ID').where(dataframe.ID == 4).count()

Output: 1

1. dataframe.select('ID').where((dataframe.ID > 4) & ((dataframe.sector == 'sales') | (dataframe.sector == 'IT'))).count()

Output: 3

**Assignment 1:**

* Read a file and create the word frequency vector, that is, each word and the number of times it is used in the file. rdd.map() and rdd.reduce() can help you in this case.
* Filter out the given common words (below) from the word frequency vector using a single filter operation.
* Then use sortBy on the count to see the different but most frequent words from the word frequency vector.

common\_words = ["us", "has", "all", "they", "from",

"who","what","on","by","more","as","not","their","can","new","it","but","be","are","--", "i","have","this","will","for","with","is","that","in","our","we","a","of", "to","and","the","that's","or","make","do","you","at","it's","than","if",

"know","last","about","no","just","now","an","because","<p>we","why","we'll", "how","two","also","every","come","we've","year","over","get","take","one", "them","we're","need","want","when","like","most","-", "been","first","where","so","these","they're","good","would","there","should","-->",

"<!--","up","i'm","his","their","which","may","were","such","some","those","was", "here","she","he","its","her","his","don't","i've","what's","didn't","shouldn't", "(applause.)","let's","doesn't"]