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.

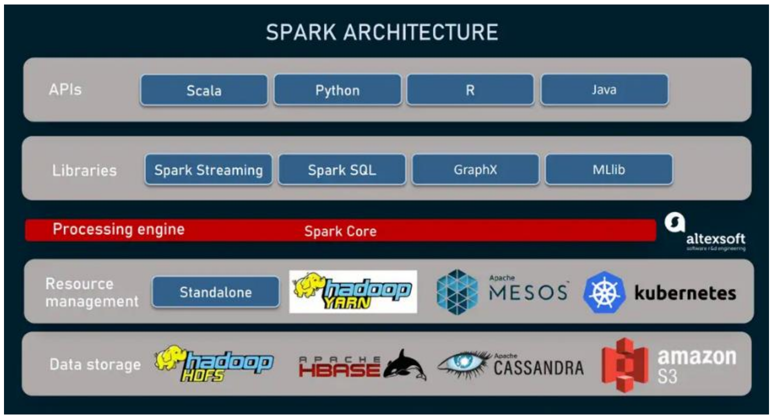

Figure 1: Spark Architecture

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.

1. SparkSession is the unified entry point introduced in Spark 2.0, encompassing all functionality including SQL, DataFrames, and Datasets, whereas SparkContext was the original entry point for RDDs and basic Spark functionalities.

2. SparkSession provides a more user-friendly and higher-level abstraction suitable for using Spark's newer APIs, whereas SparkContext is focused on low-level APIs primarily dealing with RDDs.

3. SparkSession can be used to access and manipulate data using SQL-like commands and data frames directly, which is not directly possible with SparkContext.

4. SparkSession automatically incorporates both SparkContext and SQLContext, allowing streamlined management of different contexts, whereas SparkContext does not include capabilities for working with data frames or SQL without additional context.

5. SparkSession has built-in methods to create DataFrames and perform SQL queries, simplifying tasks like reading data from various sources; SparkContext requires more setup for tasks beyond basic RDD operations.

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).

<u>Add the following codes into the cell and execute:</u>

```
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?

The "Datacamp_Ecommerce.csv" file contains the following column names:

a. InvoiceNo b. StockCode c Description d. Quantity e. InvoiceDate f. UnitPrice g. CustomerID

h. Country

There are a total of 541,909 rows in the "Datacamp_Ecommerce.csv" CSV file.

Task2: Create Dataframes from different data sources

<u>Create Dataframes</u>

1.
   ```
   my_list = [['a', 1, 2], ['b', 2, 3],['c', 3, 4]]
   col_name = ['A', 'B', 'C']
   spark.createDataFrame(my_list, col_name).show()
   ```

2.
   ```
   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

3.
   ```
   df=pd.DataFrame(my_list,col_name)
   ```

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

1. Execution Environment and Distribution:

Spark DataFrame: Operates in a distributed system environment, meaning the data is partitioned across multiple machines in a cluster. This setup enables handling of very large datasets that surpass the memory capabilities of a single machine.

pandas DataFrame: Designed to run on a single machine, holding data in local memory. This limits its capacity to smaller datasets that must fit within the available RAM of the machine but offers high-speed processing for these datasets.

2. Data Handling and Performance:

Spark DataFrame: Uses lazy evaluation, where operations are not computed immediately but optimized into an execution plan that is executed across the cluster. This approach is efficient for large-scale data transformations and aggregations.

pandas DataFrame: Executes operations immediately (eager evaluation), making it very responsive for interactive data manipulation tasks with immediate feedback, which is beneficial for data analysis and exploration on smaller data sets.

3. Output and Visualization:

Spark DataFrame: When displaying data, Spark DataFrames typically show a preview with truncated information unless explicitly instructed otherwise. This behavior is suitable for a quick overview of large datasets without overwhelming the user with too much data.

pandas DataFrame: Provides rich formatting options and displays data in a highly readable tabular form, which can be fully visualized if the dataset fits well within the system's memory. This makes pandas ideal for detailed data analysis where viewing complete records is necessary.

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.

The procedure to define a schema for a Spark DataFrame when creating it from an RDD is critical for ensuring that the DataFrame correctly interprets the types and structure of the data. Here's a detailed explanation of how to define and apply a schema:

Steps to Define and Use a Schema

1. Import Required Libraries:

First, ensure you import all necessary classes from PySpark's SQL module. These include `Row`, `DataFrame`, `StructType`, `StructField`, along with data type classes like `StringType`, `IntegerType`, etc.

```
  from pyspark.sql import Row, DataFrame

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

2. Create an RDD

Start by creating an RDD, which is a collection of elements partitioned across the nodes of the cluster that can be operated in parallel. You can create an RDD from a list of data elements.

```
  l = [["2015-06-23", 5], ["2016-07-20", 7]]

  rdd1 = spark.sparkContext.parallelize(l)
```

3. Define the Schema:

A schema in Spark defines the column names and types of a DataFrame. You define a schema using a `StructType` object, which is a collection of `StructField` objects. Each `StructField` specifies the name of the column, the data type, and a boolean indicating whether the column can contain null values.

```
  schema = StructType([

     StructField("data_date", StringType(), True),

     StructField("months_to_add", IntegerType(), True)

  ])
```

4. Apply the Schema to the RDD:

With the RDD and schema defined, you can create a DataFrame by calling the `createDataFrame` method on your Spark session. Pass in the RDD and the schema as arguments. This method transforms the RDD according to the defined schema, resulting in a DataFrame that has the specified column names and data types.

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

5. Show the DataFrame:

To verify that the DataFrame has been created as expected, use the `show()` method, which prints the contents of the DataFrame to the console. This step is useful for debugging and ensuring that the data has been loaded correctly.

```
  df.show()
```

  ```
  ```

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.

This code creates a SparkSession, then creates an RDD from the input data using **`create_RDD`** function, and finally prints the type of the RDD created. The output is **`<class 'pyspark.rdd.RDD'>`**, indicating that the RDD object has been successfully created.

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
     ```
  2. 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
2.  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. Control Over Data Types:

createDataFrame()`: Allows explicit control over data types through a `StructType` schema, offering precise specification for each column.

toDF()`: Relies on Spark's automatic type inference, which is simpler but offers less control over the data types.

2. Method of Invocation:

createDataFrame(): Called on a SparkSession object, requiring both the RDD and the schema as arguments, emphasizing a more structured approach.

toDF(): Called directly on the RDD object, only needing a list of column names, highlighting ease and immediacy.

3. Usage Context:

createDataFrame(): Preferred when detailed control over column data types is necessary, such as in formal data processing flows or when integrating with strict data sources.

toDF(): More suitable for quick transformations where the data schema is simple or already well-defined, making it ideal for rapid development and testing.

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()
```

<u>How the filter() transformation works</u>

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) count_id_4 = dataframe.filter(dataframe.ID == '4').count()

Output: 1

2) count_id_gt_4_sales_it = dataframe.filter((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"]