Practical 1- Spark installation

Practical 2- How to create RDD

Practical 3- Transformation and action on RDD

Practical 4- Counting Word Occurrences using flat map()

Practical 5-Executing SQL commands and SQL-style functions on a Data Frame

Practical 6-Customer with Data Frames

Practical 7-Use Broadcast Variables to Display Movie Names Instead of ID Numbers

Practical 8- Create Similar Movies from One Million Rating

Practical 9- Statistical operation on data frame

Practical 10- Using Spark ML to Produce Movie Recommendations

Practical 1- Howto SetUpSparkonWindow10

Apache Spark is an open-source framework that processes large volumes ofstreamdatafrommultiplesources.Sparkisusedindistributedcomputingwithmachinelearningapplications,dataanalytics,andgraph-parallelprocessing.

This guide will show you **how to install Apache Spark on Windows 10** andtestthe installation**.**

Prerequisites

- AsystemrunningWindows10
- Auseraccountwithadministratorprivileges(requiredtoinstallsoftware,modifyfile permissions,andmodifysystemPATH)
- CommandPromptorPowershell
- Atoolto extract.tarfiles,suchas 7-Zip

InstallApacheSparkonWindows

Installing Apache Spark on Windows 10 may seem complicated to noviceusers,butthissimpletutorialwillhaveyouupandrunning. If youalready haveJava8and

3installed,youcanskipthe firsttwosteps.

Step 1:InstallJava8

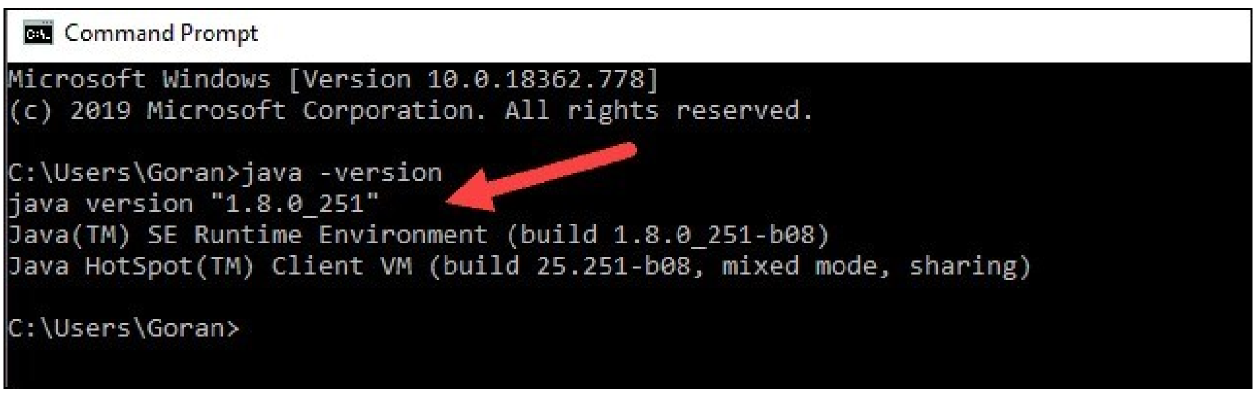
ApacheSparkrequires Java 8.Youcancheck to seeifJavais installedusingthecommandprompt.

Open the command line by clicking **Start** > type *cmd*> click **CommandPrompt**.

Typethefollowingcommandinthecommandprompt:

```
java-version
```

IfJava is installed,itwillrespondwith thefollowingoutput:



Yourversion may bedifferent.Thesecond digitistheJavaversion– inthiscase,Java 8.

Ifyoudon'thaveJava installed:

1. Openabrowserwindow,andnavigatetohttps://java.com/en/download/.

2. ClicktheJava Download buttonandsavethefile toalocation ofyourchoice.

3. Oncethedownload finishesdouble-click thefiletoinstallJava.

Note: At the time this article was written, the latest Java version is 1.8.0_251.Installing a later version will still work. This process only needs the JavaRuntime Environment (JRE) – the full Development Kit (JDK) is not required.ThedownloadlinktoJDKishttps://www.oracle.com/java/technologies/javase-downloads.html.

Step 2:Install

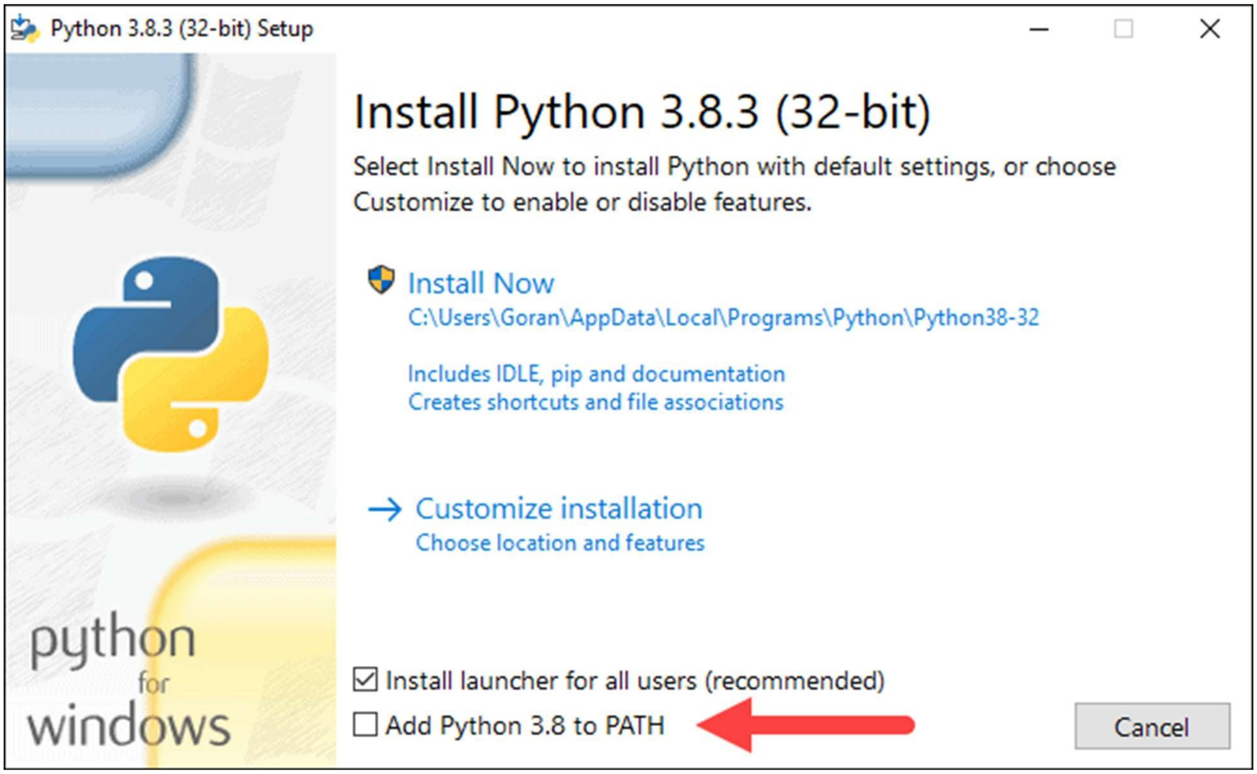
1. Toinstallthepackagemanager,navigateto https://www..org/inyourwebbrowser.

2. Mouse over the **Download** menu option and click **3.8.3**. 3.8.3 is thelatestversion at thetimeofwriting thearticle.

3. Oncethedownloadfinishes,runthefile.

4. Near the bottom of the first setup dialog box, check off *Add 3.8 toPATH*.Leave the otherbox checked.

5. Next,click **Customizeinstallation**.



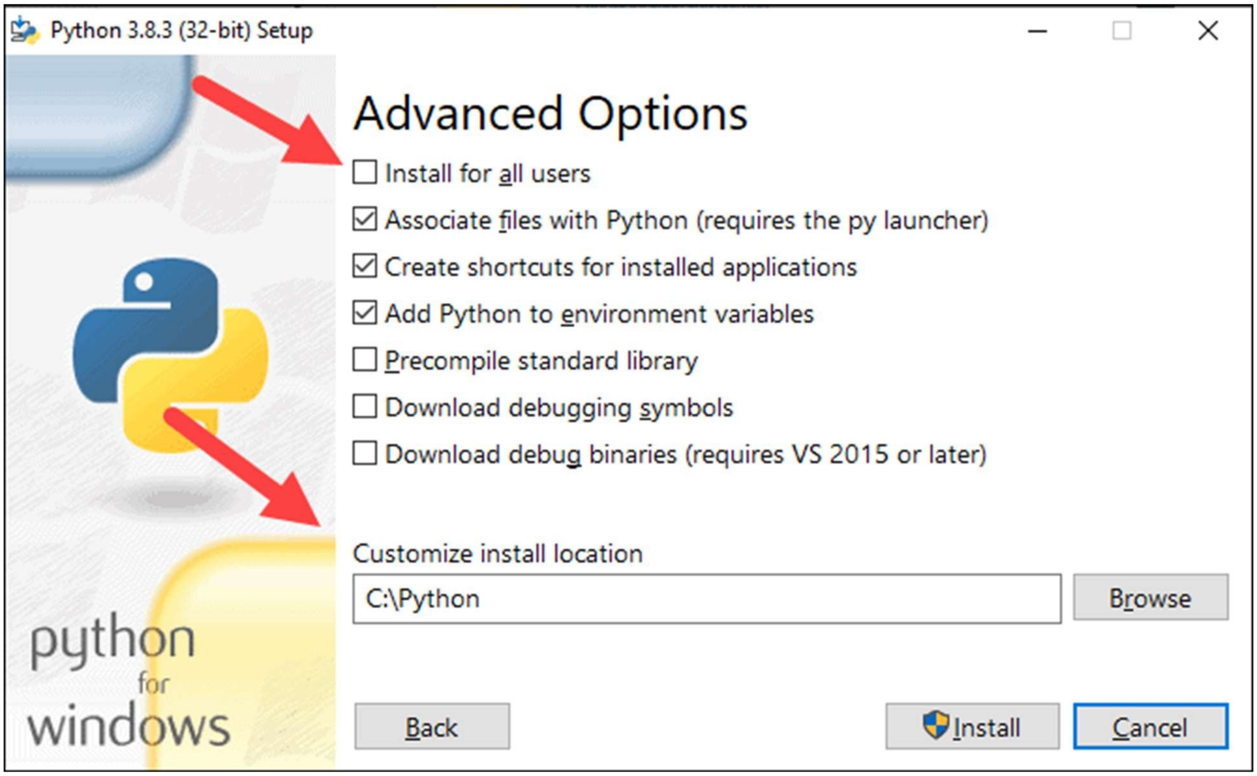
6. You can leaveallboxes checkedatthis step,oryou can unchecktheoptionsyou donotwant.

7. Click **Next**.

8. Selectthebox**Installforallusers**andleaveotherboxesastheyare.

9. Under*Customizeinstalllocation,*click**Browse**andnavigatetothe Cdrive.Adda newfolderand nameit.

10. Selectthatfolderandclick**OK**.



11. Click **Install**,and letthe installationcomplete.

12. When the installation completes, click the *Disable path length limit* optionatthe bottomand then click**Close**.

13. Ifyouhave acommandpromptopen,restartit.Verifythe installationbycheckingtheversion of:

```
python--version
```

Theoutputshouldprint **3.8.3**.

Note: Fordetailedinstructions onhowtoinstall3 onWindowsorhowtotroubleshoot potential issues,referto ourInstall 3onWindows guide.

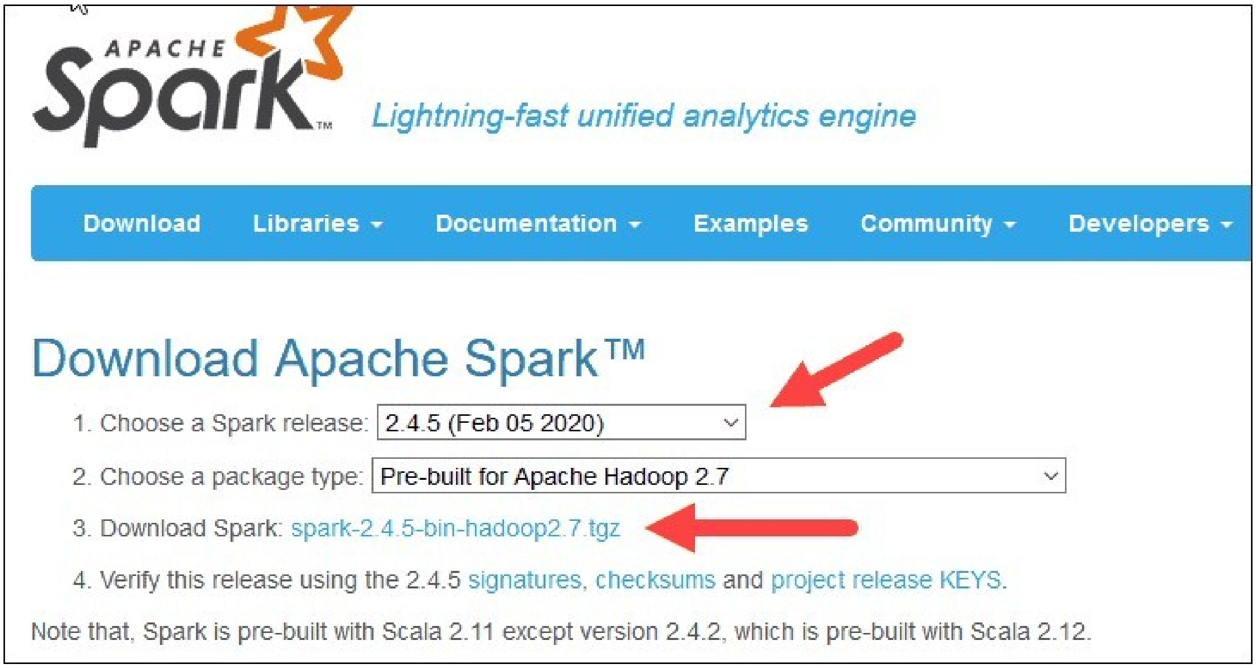
Step 3:DownloadApacheSpark

1. Openabrowserandnavigatetohttps://spark.apache.org/downloads.html.

2. Under the *Download Apache Spark* heading, there are two drop-downmenus.Usethecurrentnon-preview version.

- In our case, in ***Choose a Spark release*** drop-down menu select**2.4.5 (Feb05 2020)**.
- In the second drop-down ***Choose a package type***, leave the selection **Pre-builtforApacheHadoop2.7**.

3. Clickthe***spark-2.4.5-bin-hadoop2.7.tgz***link.



4. Apagewitha listofmirrors loadswhere youcansee differentservers todownload from. Pick any from the list and save the file to your Downloadsfolder.

Step4:VerifySparkSoftwareFile

1. Verify the integrity of your download by checking the **checksum** of the file.Thisensures youareworkingwithunaltered,uncorrupted software.

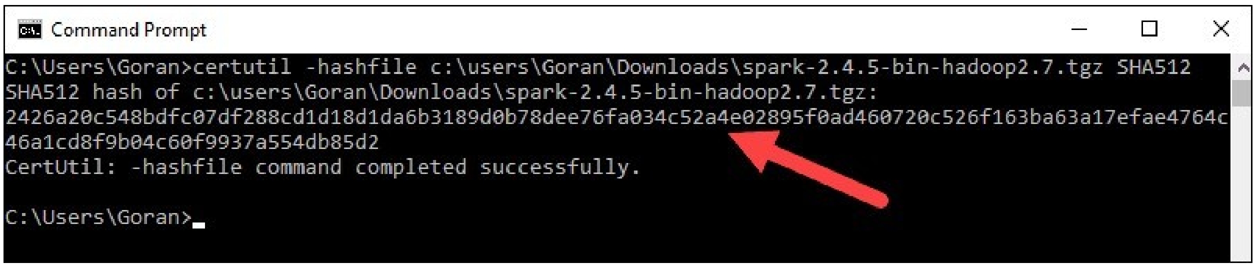
2. Navigate back to the *Spark Download* page and open the **Checksum** link,preferablyina new tab.

3. Next,open acommand lineandenterthefollowingcommand:

```
certutil-hashfilec:\users\username\Downloads\spark-
2.4.5-bin-hadoop2.7.tgzSHA512
```

4. Changetheusernametoyourusername.Thesystemdisplaysalong

alphanumericcode,alongwiththe message**Certutil:-hashfilecompletedsuccessfully**.



5. Comparethecodetothe one youopened ina newbrowsertab.Iftheymatch,yourdownload fileis uncorrupted.

Step 5:InstallApacheSpark

InstallingApacheSparkinvolves **extractingthedownloadedfile** tothedesiredlocation.

1. Create a new folder named *Spark* in the root of your C: drive. From acommandline,enterthefollowing:

```
cd \

mkdirSpark
```

2. InExplorer,locate theSparkfileyoudownloaded.

3. Right-clickthe fileand extractittoC:\Sparkusingthe toolyouhaveon yoursystem(e.g.,7-Zip).

4. Now, your *C:\Spark* folder has a new folder *spark-2.4.5-bin-hadoop2.7* withthenecessary files inside.

Step6:Add winutils.exeFile

Downloadthe **winutils.exe**filefortheunderlyingHadoopversionfortheSparkinstallationyou downloaded.

1. Navigate to this URL https://github.com/cdarlint/winutils and insidethe**bin**folder,locate**winutils.exe**,andclick it.

2. Find the**Download**buttonon therightsidetodownloadthefile.

3. Now, create new folders ***Hadoop***and **bin** on C: using Windows Explorer ortheCommand Prompt.

4. Copythewinutils.exefilefromtheDownloadsfolderto**C:\hadoop\bin**.

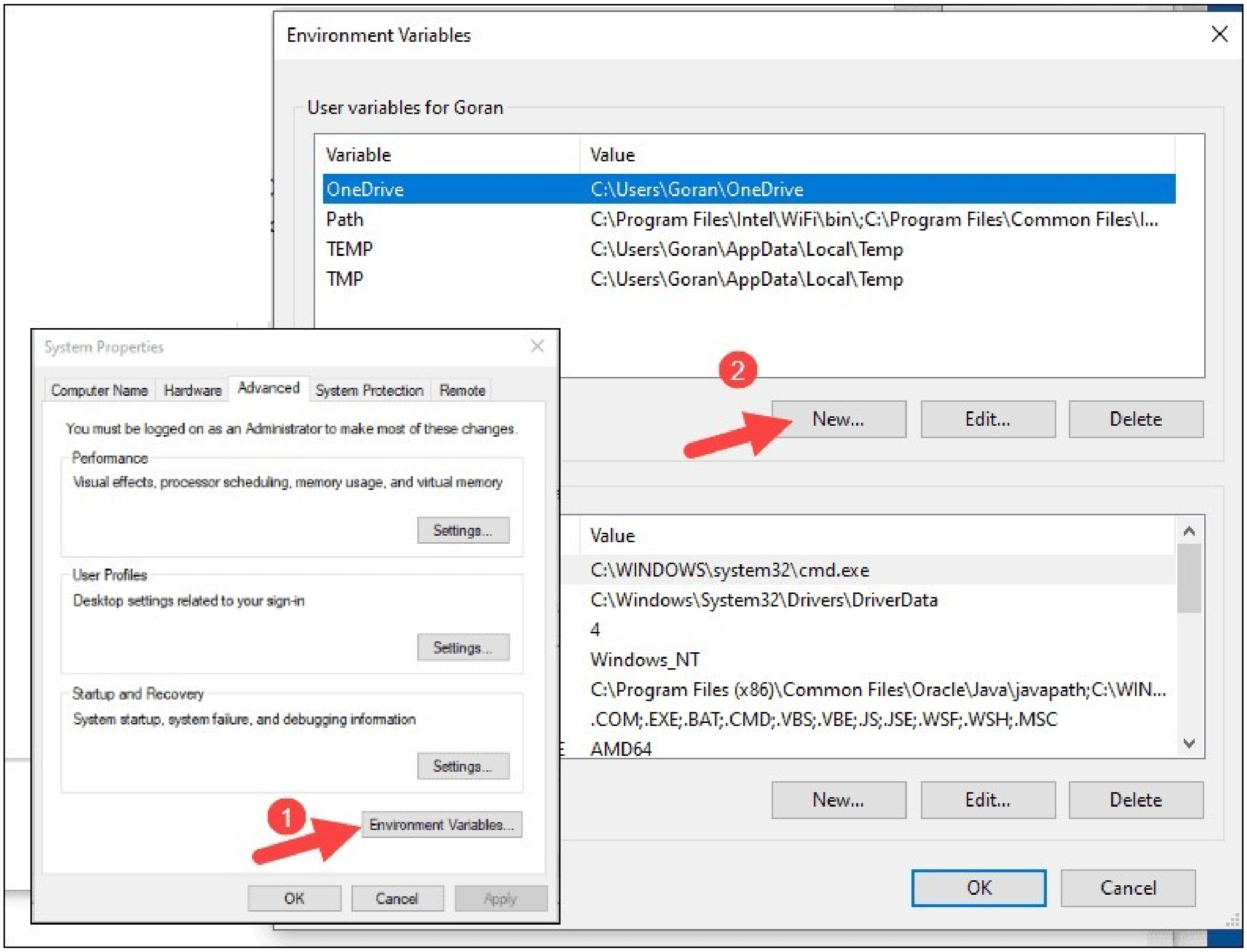
Step7:ConfigureEnvironmentVariables

This step adds the Spark and Hadoop locations to your system PATH. Itallowsyou to runthe Sparkshelldirectlyfroma commandpromptwindow.

1. Click **Start**andtype *environment*.

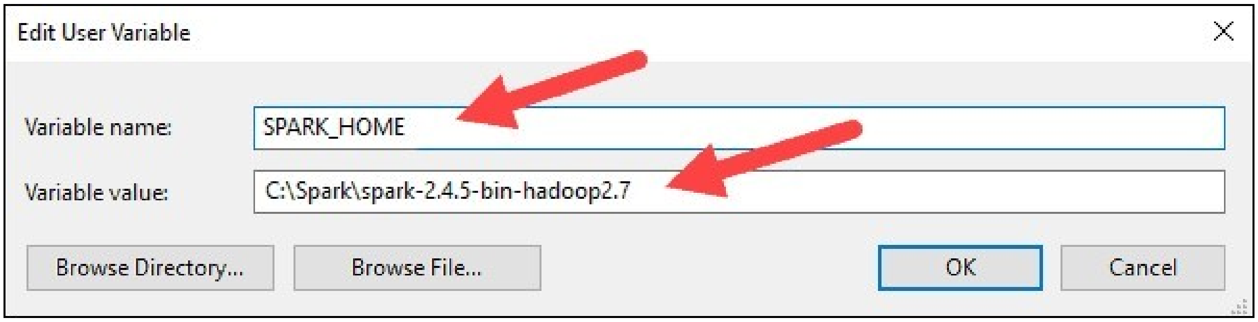
2. Selecttheresultlabeled***Editthesystem environmentvariables***.

3. A System Properties dialog box appears. In the lower-right corner,click**EnvironmentVariables**andthenclick **New**inthe next window.

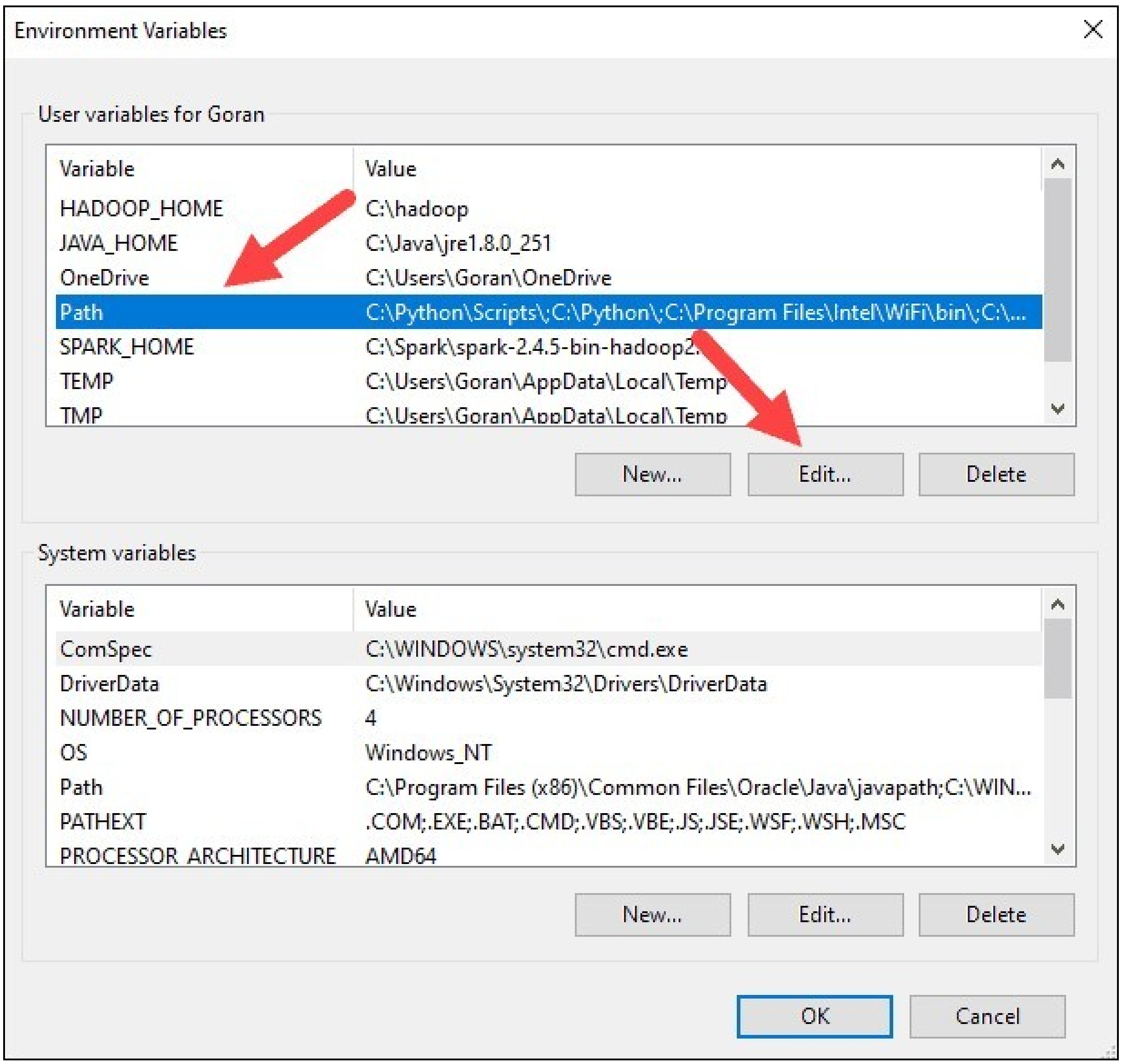


4. For*VariableName*type***SPARK_HOME***.

5. For *Variable Value* type **C:\Spark\spark-2.4.5-bin-hadoop2.7** and clickOK.If you changedthefolderpath,usethat one instead.

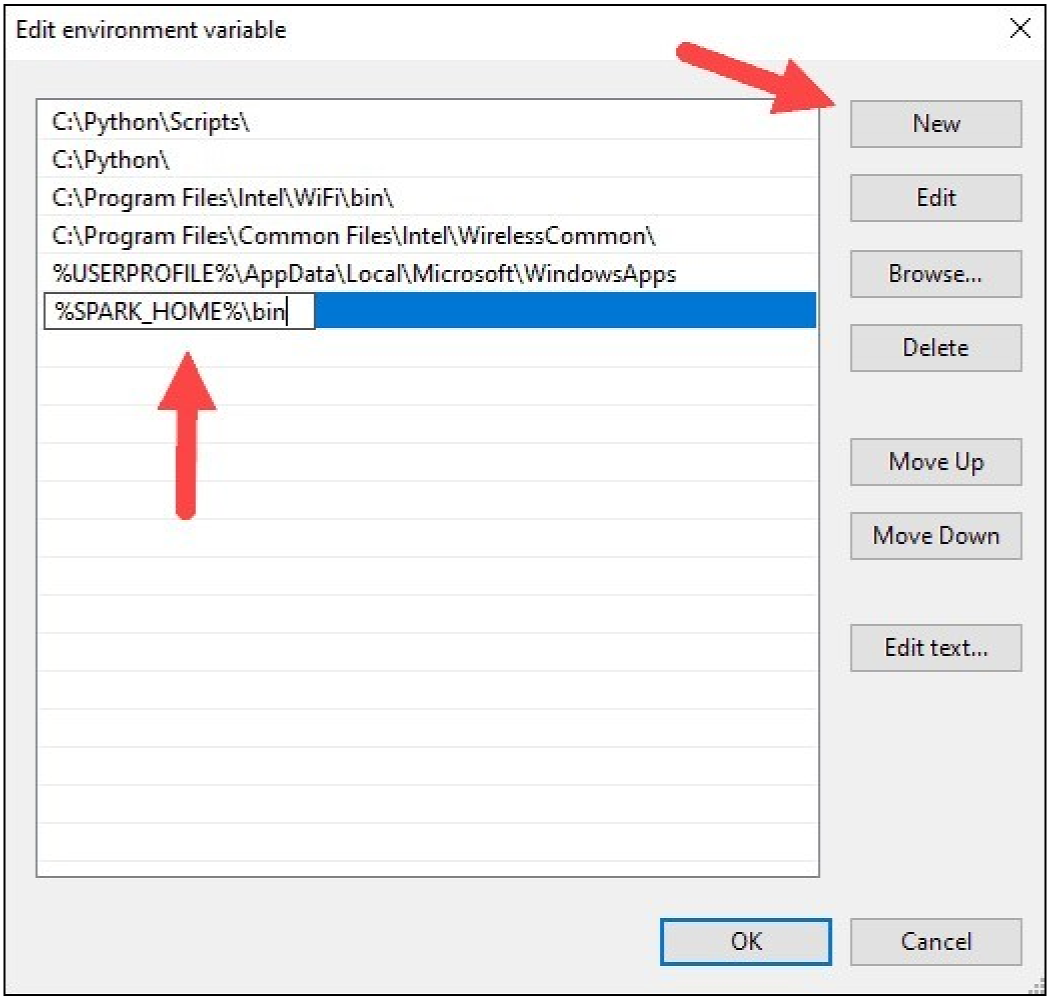


6. In the top box, click the **Path** entry, then click **Edit**. Be careful with editingthesystempath.Avoiddeletinganyentries alreadyon thelist.



7. You shouldseeaboxwithentries onthe left.Onthe right,click**New**.

8. The system highlights a new line. Enter the path to the Sparkfolder **C:\Spark\spark-2.4.5-bin-hadoop2.7\bin**. We recommendusing**%SPARK_HOME%\bin**toavoidpossibleissueswiththepath.



9. Repeatthisprocess forHadoop andJava.

- For Hadoop, the variable name is **HADOOP_HOME** and for the value usethe path of the folder you created earlier: **C:\hadoop.** Add **C:\hadoop\bin** tothe**Pathvariable**field,butwerecommendusing**%HADOOP_HOME%\bin**.
- ForJava, thevariable nameis**JAVA_HOME**andforthe value usethepathto your Java JDK directory (in our case it's **C:\ProgramFiles\Java\jdk1.8.0_251**).

10. Click **OK**to close allopenwindows.

Note: Starbyrestarting theCommand Prompttoapplychanges.Ifthatdoesn'twork,you willneedtoreboot thesystem.

Step8:Launch Spark

1. Open a new command-prompt window using the right-click and **Run asadministrator**:

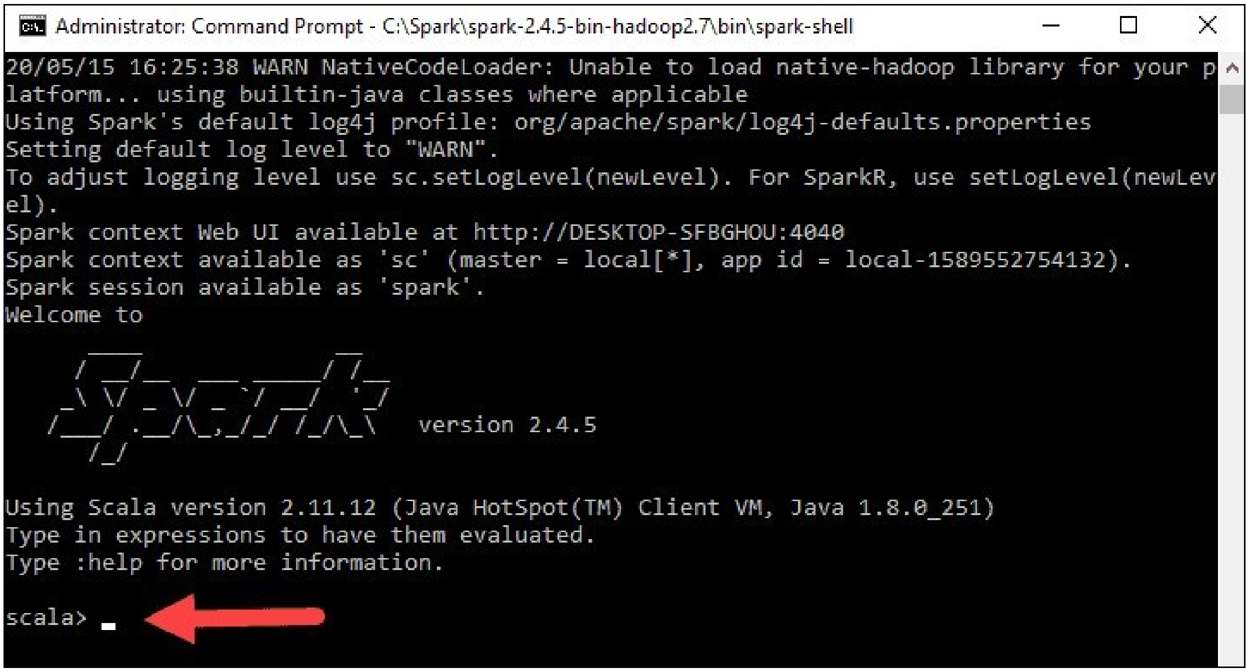
2. TostartSpark,enter:

```
C:\Spark\spark-2.4.5-bin-hadoop2.7\bin\spark-shell
```

Ifyouset the**environment path**correctly,youcantype**spark-shell**to launchSpark.

3. The system should display several lines indicating the status of theapplication.You may get aJavapop-up.Select**Allowaccess** to continue.

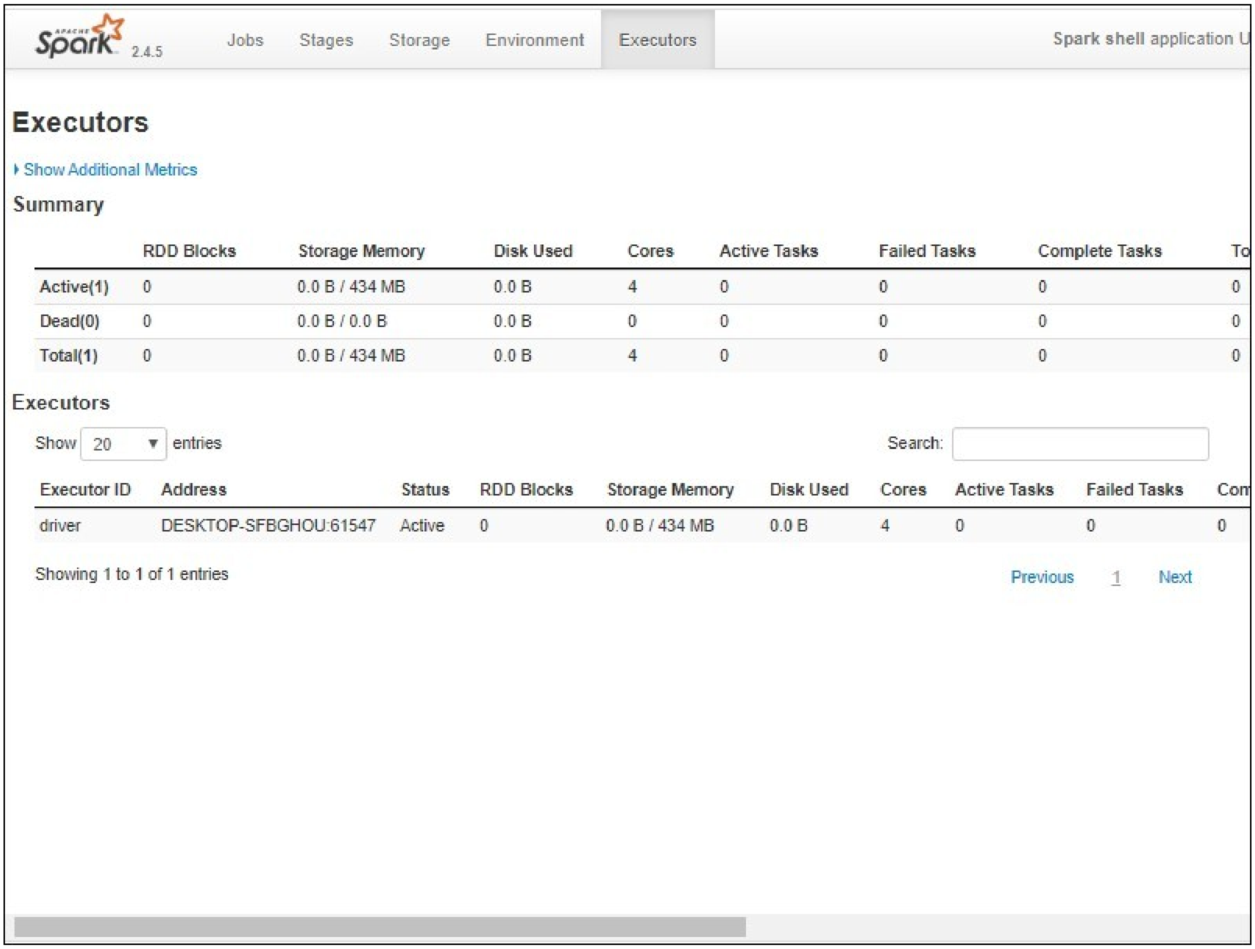
Finally,theSparklogo appears,andthepromptdisplays the**Scala shell**.



4.,Opena webbrowserandnavigateto**http://localhost:4040/**.

5. Youcan replace**localhost**withthenameofyoursystem.

6. Youshouldseean ApacheSparkshellWebUI.Theexamplebelowshowsthe*Executors* page.



7. ToexitSpark andclose the Scala shell,press**ctrl-d**in the command-promptwindow.

Note: Ifyou installed, youcanrunSparkusing withthiscommand:

```
pyspark
```

Exitusing**quit()**.

TestSpark

In this example, we will launch the Spark shell and use Scala to read thecontentsofa file.You can usean existing file,suchas the*README*file intheSpark directory, or you can create your own. We created *pnaptest*with sometext.

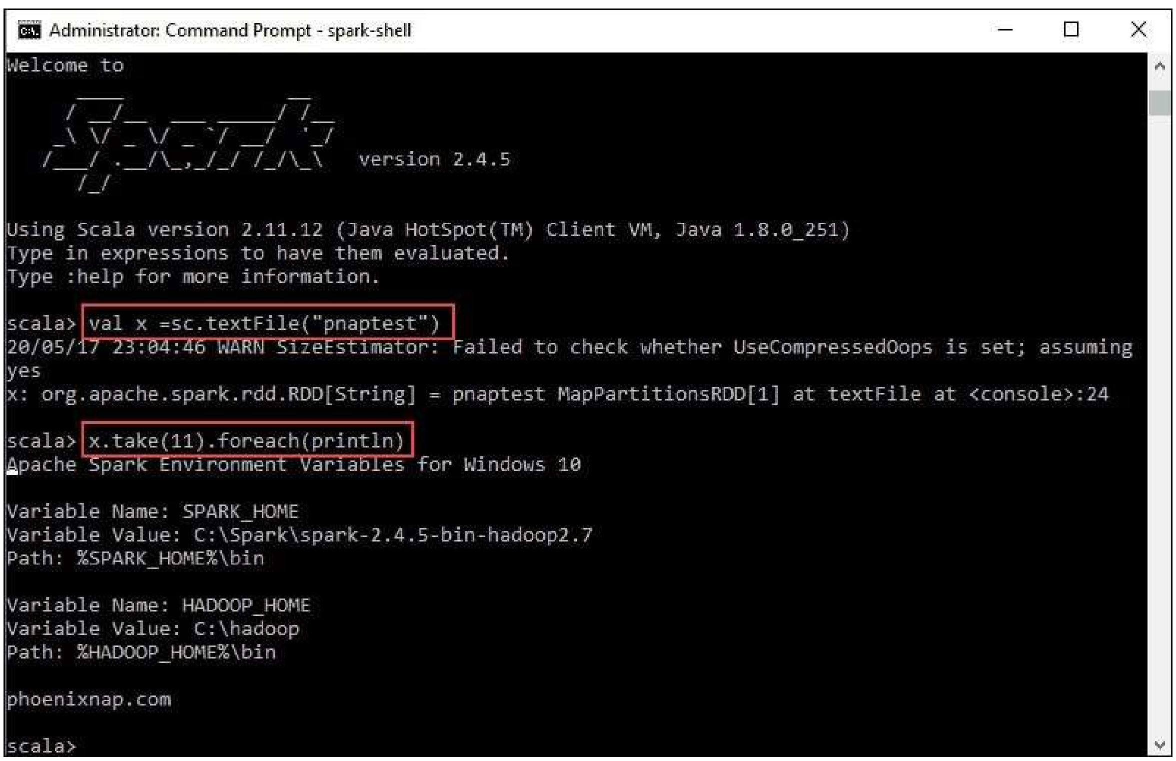
1. Openacommand-promptwindowandnavigatetothefolderwiththefileyouwant touseandlaunch theSparkshell.

2. First,state avariable to usein the Sparkcontextwiththe name ofthe file.Rememberto add the fileextensionifthere is any.

```
valx=sc.textFile("pnaptest")
```

3. TheoutputshowsanRDDiscreated.Then,wecanviewthefilecontentsbyusing thiscommandtocallan action:

```
x.take(11).foreach(println)
```



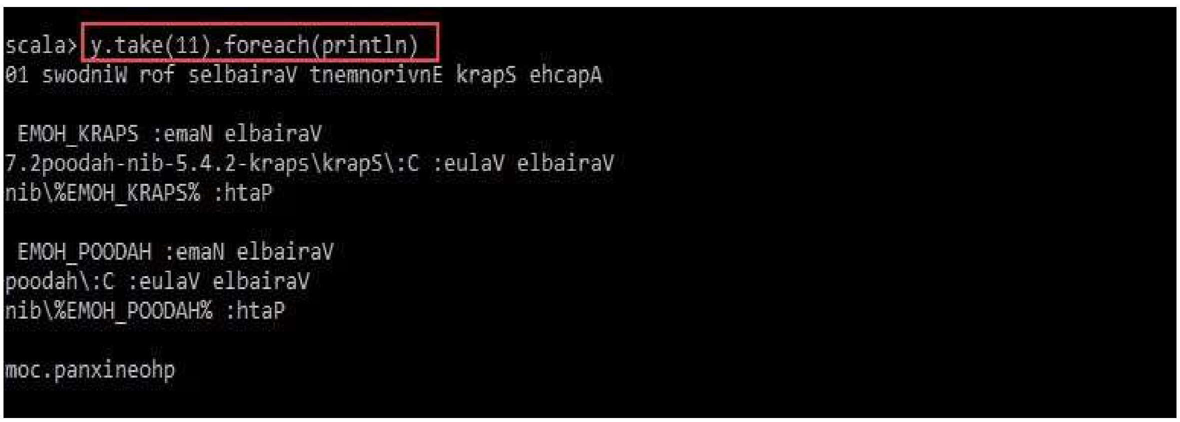
ThiscommandinstructsSpark toprint11 linesfromthefileyouspecified.Toperform an action on this file (**value x**), add another value **y**, and do a maptransformation.

4. Forexample,youcanprintthecharactersinreversewiththiscommand:

```
valy=x.map(_.reverse)
```

5. Thesystemcreates achildRDD inrelationto thefirstone.Then,specifyhowmany lines youwant to printfromthe value**y**:

```
y.take(11).foreach(println)
```



The output prints 11 lines of the *pnaptest*file in the reverse order.Whendone, exittheshellusing**ctrl-d**.

Conclusion

Youshould nowhave aworkinginstallation ofApacheSparkon Windows 10with all dependencies installed. Get started running an instance of Spark inWindowsenvironment.

Practical2- How to Create RDD using methods

In Apache Spark, an RDD (Resilient Distributed Dataset) is a fundamental data structure that represents a distributed collection of objects. RDDs can be created in several ways. Below are the most common methods to create an RDD in Spark:

1. **Parallelizing a Collection (Creating RDD from an Existing Collection)**

You can create an RDD by parallelizing an existing collection (like a list, array, etc.) in your driver program. This is done using the parallelize method of SparkContext.

Example in PySpark:

```
frompysparkimportSparkContext

sc = SparkContext("local", "RDD Example")

# Create a list of data
data = [1, 2, 3, 4, 5]

# Parallelize the data to create an RDD
rdd = sc.parallelize(data)

# Print the RDD
print(rdd.collect())
```

- **Output**: [1, 2, 3, 4, 5]

2. **Loading Data from External Storage (Creating RDD from a File)**

You can create an RDD by loading data from external storage systems (e.g., HDFS, local file system, S3, or other distributed file systems).

```
frompysparkimportSparkContext
sc = SparkContext("local", "RDD Example")

# Create an RDD by reading a text file
rdd = sc.textFile("path/to/your/file.txt")
```

# Print the content of the RDD
print(rdd.collect())

- **Output**: The lines in the file will be loaded into the RDD.

3. From Existing RDD (Transformation)

You can create an RDD from another RDD using transformations. For instance, applying map, filter, or flatMap to an existing RDD creates a new RDD.

Example in PySpark:

```
frompysparkimportSparkContext
sc = SparkContext("local", "RDD Example")

# Create an RDD
rdd = sc.parallelize([1, 2, 3, 4, 5])

# Apply a transformation (multiply each element by 2)
new_rdd = rdd.map(lambda x: x * 2)

# Print the new RDD
print(new_rdd.collect())
```

- **Output**: [2, 4, 6, 8, 10]

4. Creating RDD Using**wholeTextFiles** (For Text Files in a Directory)

wholeTextFiles is useful for reading multiple files from a directory and returning an RDD of pairs (fileName, content).

Example in PySpark:

```
frompysparkimportSparkContext
sc = SparkContext("local", "RDD Example")
```

```
# Read multiple files from a directory
rdd = sc.wholeTextFiles("path/to/directory")

# Print the content of the RDD (file names and contents)
print(rdd.collect())
```

- **Output**: Each entry in the RDD will be a tuple (fileName, content).

5. Creating RDD using parallelize **with More Partitions**

By default, Spark will parallelize an RDD using a default number of partitions based on the cluster size. You can specify the number of partitions manually.

```
frompysparkimportSparkContext
sc = SparkContext("local", "RDD Example")

# Parallelize the data with 4 partitions
rdd = sc.parallelize([1, 2, 3, 4, 5], 4)

# Print the RDD
print(rdd.collect())
```

Output: [1, 2, 3, 4, 5] (but the data is distributed across 4 partitions)

Practical 3- Transformation and action on RDD

RDD Transformations:

Transformations are operations that return a new RDD and are lazy, meaning they are not executed until an action is called.

1. map():

Description: Applies a function to each element of the RDD and returns a new RDD.

Example:

rdd = sc.parallelize([1, 2, 3, 4])

result = rdd.map(lambda x: x * 2)

print(result.collect()) # [2, 4, 6, 8]

2. filter():

Description: Filters elements of the RDD based on a function that returns True or False.

Example:

rdd = sc.parallelize([1, 2, 3, 4, 5, 6])

result = rdd.filter(lambda x: x % 2 == 0)

print(result.collect()) # [2, 4, 6]

3. flatMap():

Description: Similar to map, but each input element can produce zero or more output elements (i.e., it "flattens" the result).

Example:

rdd = sc.parallelize([1, 2, 3])

result = rdd.flatMap(lambda x: (x, x * 2))

print(result.collect()) # [1, 2, 2, 4, 3, 6]

4. reduceByKey():

Description: Combines values with the same key using a specified function. Useful for aggregating data.

Example:

rdd = sc.parallelize([('a', 1), ('b', 2), ('a', 3)])

result = rdd.reduceByKey(lambda x, y: x + y)

print(result.collect()) # [('a', 4), ('b', 2)]

5. groupByKey():

Description: Groups the values of the RDD by key.

Example:

rdd = sc.parallelize([('a', 1), ('b', 2), ('a', 3)])

result = rdd.groupByKey().mapValues(list)

print(result.collect()) # [('a', [1, 3]), ('b', [2])]

6. join():

Description: Joins two RDDs by key.

Example:

rdd1 = sc.parallelize([('a', 1), ('b', 2)])

rdd2 = sc.parallelize([('a', 3), ('b', 4)])

result = rdd1.join(rdd2)

print(result.collect()) # [('a', (1, 3)), ('b', (2, 4))]

7. distinct():

Description: Removes duplicates from an RDD.

Example:

rdd = sc.parallelize([1, 2, 3, 2, 1])

result = rdd.distinct()

print(result.collect()) # [1, 2, 3]

RDD Actions:

Actions are operations that trigger the execution of transformations and return results.

1. collect():

Description: Returns the entire RDD as a list to the driver program.

Example:

rdd = sc.parallelize([1, 2, 3, 4])

result = rdd.collect()

print(result) # [1, 2, 3, 4]

2. count():

Description: Returns the number of elements in the RDD.

Example:

rdd = sc.parallelize([1, 2, 3, 4])

result = rdd.count()

print(result) # 4

3. reduce():

Description: Aggregates the elements of the RDD using a specified binary function (e.g., sum, max, min).

Example:

rdd = sc.parallelize([1, 2, 3, 4])

result = rdd.reduce(lambda x, y: x + y)

print(result) # 10

4. first():

Description: Returns the first element of the RDD.

Example:

rdd = sc.parallelize([1, 2, 3, 4])

result = rdd.first()

print(result) # 1

5. take(n):

Description: Returns the first n elements of the RDD.

Example:

rdd = sc.parallelize([1, 2, 3, 4])

result = rdd.take(3)

print(result) # [1, 2, 3]

6. saveAsTextFile():

Description: Writes the RDD to a file on the distributed storage system.

Example:

rdd = sc.parallelize([1, 2, 3, 4])

```
rdd.saveAsTextFile("/dbfs/tmp/output.txt")
```

7. countByKey():

Description: Counts the occurrences of each key in a key-value RDD.

Example:

```
rdd = sc.parallelize([('a', 1), ('b', 2), ('a', 3), ('b', 4)])

result = rdd.countByKey()

print(result)  # {'a': 2, 'b': 2}
```

Practical 4- Counting Word Occurrences using flat map()

Create an RDD containing text data.

Use flatMap() to break the text into individual words.

Apply map() to create key-value pairs, where the key is the word, and the value is 1.

Use reduceByKey() to aggregate the word counts.

Display the results using collect().

```
# 1. Sample text data
text_data = [
    "Apache Spark is amazing",
    "Spark is a unified analytics engine",
    "It provides high performance for large-scale data processing"
]
# 2. Create an RDD from the sample data
rdd = sc.parallelize(text_data)

# 3. Use flatMap to split each line into words
words_rdd = rdd.flatMap(lambda line: line.split(" "))

# 4. Map each word to a key-value pair (word, 1)
word_pairs_rdd = words_rdd.map(lambda word: (word.lower(), 1))

# 5. Use reduceByKey to count occurrences of each word
word_counts_rdd = word_pairs_rdd.reduceByKey(lambda x, y: x + y)

# 6. Collect the result and print it
word_counts = word_counts_rdd.collect()

# Print the word counts
for word, count in word_counts:
print(f"{word}: {count}")



#OUTPUT:
apache: 1
spark: 2
```

is: 2

amazing: 1

a: 1

unified: 1

analytics: 1

engine: 1

it: 1

provides: 1

high: 1

performance: 1

for: 1

large-scale: 1

data: 1

processing: 1

Practical 5-Executing SQL commands and SQL-style functions on a Data Frame

```
# Step 1: Create a sample DataFrame
data = [
    ("Alice", 29, "Engineering"),
    ("Bob", 35, "Sales"),
    ("Charlie", 40, "Engineering"),
    ("David", 30, "HR"),
    ("Eva", 25, "Sales")
]
columns = ["name", "age", "department"]

df = spark.createDataFrame(data, columns)

# Step 2: Register the DataFrame as a temporary SQL view
df.createOrReplaceTempView("employees")

# Step 3: Execute SQL commands on the DataFrame
# Example SQL queries

# a) Query to select all rows
sql_query = "SELECT * FROM employees"
result = spark.sql(sql_query)
result.show()

# b) Query to filter employees older than 30
sql_query = "SELECT name, age, department FROM employees WHERE age > 30"
result = spark.sql(sql_query)
result.show()

# c) Group by department and calculate the average age of employees
sql_query = "SELECT department, AVG(age) as avg_age FROM employees GROUP BY
department"
result = spark.sql(sql_query)
```

```
result.show()

# Step 4: SQL-style functions directly on DataFrame (without SQL)
# Example using DataFrame API

# a) Filter employees older than 30
filtered_df = df.filter(df.age> 30)
filtered_df.show()

# b) Group by department and calculate the average age of employees
grouped_df = df.groupBy("department").avg("age")
grouped_df.show()

# c) Select employees and add a new column with a conditional expression
frompyspark.sql import functions as F
df_with_new_col = df.withColumn("age_category",
F.when(df.age< 30, "Young")
                              .when((df.age>= 30) & (df.age< 40), "Mid-aged")
                              .otherwise("Old"))
df_with_new_col.show()
```

OUTPUT-

```
# After executing the SQL query for all rows
+-------+---+-----------+
|   name|age|department|
+-------+---+-----------+
|  Alice| 29|Engineering|
|    Bob| 35|      Sales|
|Charlie| 40|Engineering|
|  David| 30|         HR|
|    Eva| 25|      Sales|
+-------+---+-----------+

# After executing the SQL query for employees older than 30
+-------+---+-----------+
```

```
|   name|age|department|
+-------+---+-----------+
|    Bob| 35|      Sales|
|Charlie| 40|Engineering|
+-------+---+-----------+

# After executing the SQL query for average age by department
+-----------+--------+
| department|avg_age |
+-----------+--------+
|Engineering|     34.5|
|      Sales|     30.0|
|         HR|     30.0|
+-----------+--------+

# After filtering employees older than 30 (using DataFrame API)
+-----+---+-----------+
| name|age|department|
+-----+---+-----------+
|  Bob| 35|      Sales|
|Charlie| 40|Engineering|
+-----+---+-----------+

# After calculating the average age by department (using DataFrame API)
+-----------+--------+
| department|avg(age)|
+-----------+--------+
|Engineering|    34.5|
|      Sales|    30.0|
|         HR|    30.0|
+-----------+--------+

# After adding a new column with age categories
+-----+---+-----------+-----------+
```

```
| name|age|department|age_category|
+-----+---+-----------+-----------+
|Alice| 29|Engineering|      Young|
|  Bob| 35|      Sales|   Mid-aged|
|Charlie| 40|Engineering|       Old|
|David| 30|         HR|   Mid-aged|
|  Eva| 25|      Sales|      Young|
+-----+---+-----------+-----------+
```

Practical 6-Create dataframe of Customer with transfomation

Step 1: Create a sample DataFrame with customer data

```
frompyspark.sqlimport functions as F
frompyspark.sql.typesimportStructType,  StructField,  IntegerType,  StringType,  FloatType,
DateType
fromdatetimeimportdatetime

# Sample data (customer_id, name, age, gender, total_spend, join_date)
data = [
    (1, "Alice", 29, "Female", 200.0, datetime(2020, 5, 1)),
    (2, "Bob", 35, "Male", 350.0, datetime(2019, 3, 15)),
    (3, "Charlie", 40, "Male", 150.0, datetime(2021, 7, 22)),
    (4, "David", 25, "Male", 500.0, datetime(2020, 10, 10)),
    (5, "Eva", 32, "Female", 120.0, datetime(2021, 1, 15)),
    (6, "Fay", 45, "Female", 400.0, datetime(2019, 12, 5)),
    (7, "George", 50, "Male", 600.0, datetime(2018, 9, 18)),
    (8, "Hannah", 28, "Female", 250.0, datetime(2022, 2, 20)),
]

# Define the schema
schema = StructType([
StructField("customer_id", IntegerType(), True),
StructField("name", StringType(), True),
StructField("age", IntegerType(), True),
StructField("gender", StringType(), True),
StructField("total_spend", FloatType(), True),
StructField("join_date", DateType(), True)
])

# Create a DataFrame from the sample data
df = spark.createDataFrame(data, schema)

# Show the DataFrame to inspect the data
df.show()
```

Step 2: Filter customers based on age and total spend

```
# Filter customers who are older than 30 and have spent more than 200
filtered_df = df.filter((df.age>30) & (df.total_spend>200))
filtered_df.show()
```

Step 3: Grouping and Aggregation

a) Group by Gender and Calculate the Average Spend

```
# Group by gender and calculate average spend
gender_avg_spend_df = df.groupBy("gender").agg(F.avg("total_spend").alias("avg_spend"))
gender_avg_spend_df.show()
```

b) Calculate Total Spend by Age Group

```
# Create an age group column
df_with_age_group = df.withColumn(
"age_group",
F.when(df.age<30, "Under 30")
    .when((df.age>= 30) & (df.age<40), "30-39")
    .when((df.age>= 40) & (df.age<50), "40-49")
    .otherwise("50+")
)

# Group by age group and calculate total spend
age_group_spend_df =
df_with_age_group.groupBy("age_group").agg(F.sum("total_spend").alias("total_spend"))
age_group_spend_df.show()
```

Step 4: SQL Queries on DataFrame

1. **Register the DataFrame as a temporary SQL view**:

```
df.createOrReplaceTempView("customers")
```

2. **SQL Query to Filter Customers with Spend > 300**:

```
# Execute an SQL query to get customers who have spent more than 300
sql_query = "SELECT * FROM customers WHERE total_spend> 300"
sql_result = spark.sql(sql_query)
sql_result.show()
```

3. **SQL Query to Get Total Spend by Gender**:

```
# Execute an SQL query to get total spend by gender
sql_query = "SELECT gender, SUM(total_spend) as total_spend FROM customers GROUP BY gender"
sql_gender_spend = spark.sql(sql_query)
sql_gender_spend.show()
```

Step 5: Customer Segmentation Example (Loyalty Program)

You can create customer segments based on their total spend and join date:

```
# Define a loyalty program based on total spend
df_with_loyalty = df.withColumn(
"loyalty_level",
F.when(df.total_spend<200, "Bronze")
    .when((df.total_spend>= 200) & (df.total_spend<400), "Silver")
    .when(df.total_spend>= 400, "Gold")
)

# Show the results
df_with_loyalty.show()
```

Sample Output:

sql

```
# Output after creating DataFrame
+-----------+-------+---+------+----------+----------+
```

```
|customer_id|name|age|gender|total_spend|join_date|
+-----------+-------+---+------+----------+----------+
|1|  Alice|29|Female|200.0|2020-05-01|
|2|    Bob|35|  Male|350.0|2019-03-15|
|3|Charlie|40|  Male|150.0|2021-07-22|
|4|  David|25|  Male|500.0|2020-10-10|
|5|    Eva|32|Female|120.0|2021-01-15|
|6|    Fay|45|Female|400.0|2019-12-05|
|7| George|50|  Male|600.0|2018-09-18|
|8|Hannah|28|Female|250.0|2022-02-20|
+-----------+-------+---+------+----------+----------+

# Output after filtering customers older than 30with spend >200
+-----------+-----+---+------+----------+----------+
|customer_id|name|age|gender|total_spend|join_date|
+-----------+-----+---+------+----------+----------+
|2|  Bob|35|  Male|350.0|2019-03-15|
|4|David|25|  Male|500.0|2020-10-10|
|6|  Fay|45|Female|400.0|2019-12-05|
|7|George|50|  Male|600.0|2018-09-18|
+-----------+-----+---+------+----------+----------+

# Output after calculating average spend by gender
+------+------------------+
|gender|avg_spend|
+------+------------------+
|Female|267.5|
|  Male|400.0|
+------+------------------+

# Output after calculating total spend by age group
+-------+----------+
|age_group|total_spend|
+-------+----------+
```

```
| Under 30|950.0|
|30-39|1000.0|
|40-49|700.0|
|50+|600.0|
+-------+----------+

# Output after creating customer loyalty levels
+-----------+-------+---+------+----------+----------+------------+
|customer_id|name|age|gender|total_spend|join_date|loyalty_level|
+-----------+-------+---+------+----------+----------+------------+
|1|  Alice|29|Female|200.0|2020-05-01|      Bronze|
|2|    Bob|35|  Male|350.0|2019-03-15|     Silver|
|3|Charlie|40|  Male|150.0|2021-07-22|      Bronze|
|4|  David|25|  Male|500.0|2020-10-10|       Gold|
|5|    Eva|32|Female|120.0|2021-01-15|      Bronze|
|6|    Fay|45|Female|400.0|2019-12-05|     Silver|
|7| George|50|  Male|600.0|2018-09-18|       Gold|
|8|Hannah|28|Female|250.0|2022-02-20|     Silver|
+-----------+-------+---+------+----------+----------+------------+
```

Practical 7-Use Broadcast Variables to Display Movie Names Instead of ID Numbers

In this practical, we will demonstrate how to use **Broadcast Variables** in Apache Spark to optimize the performance of joining a large dataset with a smaller dataset. Specifically, we will use broadcast variables to replace movie ID numbers with their corresponding movie names.

You have two datasets:

1. A **ratings dataset** containing user ratings for movies, where each rating is associated with a **movie ID**.
2. A **movies dataset** that maps each **movie ID** to a movie name.

You want to display the **movie names** instead of the **movie IDs** in the ratings dataset. Since the movies dataset is small and can fit into memory, we can use a **broadcast variable** to optimize the join operation.

Steps:

1. **Create a sample ratings dataset** (user_id, movie_id, rating).
2. **Create a sample movies dataset** (movie_id, movie_name).
3. **Broadcast the movies dataset** to optimize the join operation.
4. **Join the ratings dataset with the broadcasted movies dataset** to replace movie IDs with movie names.

Solution Code:

Step 1: Create Sample DataFrames

```
frompyspark.sqlimportSparkSession
frompyspark.sqlimport functions as F

# Initialize Spark session
spark = SparkSession.builder.appName("BroadcastExample").getOrCreate()

# Sample ratings data (user_id, movie_id, rating)
```

```
ratings_data = [
    (1, 101, 4.5),
    (2, 102, 3.0),
    (3, 103, 5.0),
    (4, 101, 4.0),
    (5, 104, 3.5)
]

ratings_columns = ["user_id", "movie_id", "rating"]

ratings_df = spark.createDataFrame(ratings_data, ratings_columns)

# Sample movies data (movie_id, movie_name)
movies_data = [
    (101, "The Matrix"),
    (102, "Inception"),
    (103, "The Dark Knight"),
    (104, "Forrest Gump")
]

movies_columns = ["movie_id", "movie_name"]

movies_df = spark.createDataFrame(movies_data, movies_columns)
```

Step 2: Broadcast the Movies DataFrame

Broadcast the smaller dataset (movies DataFrame) to all nodes in the cluster so that it doesn't have to be shuffled during the join operation.

```
# Broadcast the movies DataFrame to optimize the join
broadcast_movies_df = spark.sparkContext.broadcast(movies_df.collect())
```

Step 3: Join the DataFrames Using the Broadcast Variable

You can now join the **ratings dataset** with the broadcasted **movies dataset** using the movie_id column. We will use a approach to replace movie IDs with movie names.

# Step 3: Replace movie IDs with movie names using the broadcasted dataset
# Convert the broadcasted movies data into a dictionary for faster lookups
movie_dict = {row['movie_id']: row['movie_name'] for row inbroadcast_movies_df.value}

Define a UDF (User Defined Function) to map movie IDs to movie names
defget_movie_name(movie_id):
returnmovie_dict.get(movie_id, "Unknown")

Register the UDF
frompyspark.sql.functionsimportudf
frompyspark.sql.typesimportStringType
get_movie_name_udf = udf(get_movie_name, StringType())

# Add a new column to ratings_df with the movie names
ratings_with_movie_names_df = ratings_df.withColumn("movie_name", get_movie_name_udf(ratings_df.movie_id))

# Show the result
ratings_with_movie_names_df.show()

Step 4: Output

The result will show the ratings dataset with **movie names** instead of **movie IDs**.

diff

```
+-------+--------+------+-------------------+
|user_id|movie_id|rating|         movie_name|
+-------+--------+------+-------------------+
|      1|     101|   4.5|         The Matrix|
|      2|     102|   3.0|          Inception|
|      3|     103|   5.0|    The Dark Knight|
|      4|     101|   4.0|         The Matrix|
|      5|     104|   3.5|       Forrest Gump|
+-------+--------+------+-------------------+
```

Practical 8- Create Similar Movies from One Million Rating

Given a large dataset of movie ratings by users, the objective is to find similar movies. We'll use a **user-item matrix** (user ratings for movies) to train an ALS model and recommend movies based on the similarity of their ratings.

Steps:

1. **Load the dataset** containing movie ratings.
2. **Preprocess the data** (filter, clean, etc.).
3. **Use ALS (Alternating Least Squares)** for collaborative filtering.
4. **Make recommendations** for similar movies based on user-item interactions.
5. **Evaluate the model** using appropriate metrics like RMSE.

Step 1: Create or Load the Dataset

In this example, we will use the MovieLens 1M dataset (which contains one million ratings). However, since this is a typical problem, you can use any large dataset of movie ratings.

Let's assume the following columns for the ratings dataset:

- user_id: Unique identifier for users.
- movie_id: Unique identifier for movies.
- rating: Rating given by the user to a movie.
- timestamp: Timestamp of when the rating was made.

Step 2: Load the Data in Spark

```
frompyspark.sqlimportSparkSession
frompyspark.sql.functionsimport col
frompyspark.ml.recommendationimport ALS
frompyspark.ml.evaluationimportRegressionEvaluator
importos

# Initialize the Spark session
spark = SparkSession.builder.appName("MovieSimilarity").getOrCreate()
```

```
# Load the MovieLens dataset (replace with your actual file path)
# The dataset is assumed to be in CSV format.
ratings_file = "/path/to/ratings.csv"# Example file path (adjust accordingly)
movies_file = "/path/to/movies.csv"

ratings_df = spark.read.option("header", "true").csv(ratings_file)
movies_df = spark.read.option("header", "true").csv(movies_file)

# Show the data to inspect it
ratings_df.show(5)
movies_df.show(5)
```

Assuming that the ratings data has the following columns: userId, movieId, rating, and timestamp.

Step 3: Data Preprocessing and Cleaning

Ensure the data is in the correct format (casting columns to integers and floats as needed).

```
# Cast columns to appropriate data types
ratings_df = ratings_df.select(
col("userId").cast("int"),
col("movieId").cast("int"),
col("rating").cast("float")
)

movies_df = movies_df.select(
col("movieId").cast("int"),
col("title")
)
# Show cleaned data
ratings_df.show(5)
movies_df.show(5)
```

Step 4: Train the ALS Model for Collaborative Filtering

ALS (Alternating Least Squares) is the algorithm used for collaborative filtering in Spark MLlib. It works by factoring the user-item matrix into two matrices, one for users and one for items (movies in this case), and optimizing them to predict missing values.

```
# Split data into training and test sets
(training_data, test_data) = ratings_df.randomSplit([0.8, 0.2])

# Train the ALS model
als = ALS(userCol="userId", itemCol="movieId", ratingCol="rating", coldStartStrategy="drop")
model = als.fit(training_data)

# Make predictions
predictions = model.transform(test_data)

# Evaluate the model using RMSE
evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating", predictionCol="prediction")
rmse = evaluator.evaluate(predictions)

print(f"Root-Mean-Square Error (RMSE): {rmse}")
```

Step 5: Finding Similar Movies Based on User Ratings

To find similar movies, we can either:

- Use **item-based collaborative filtering**, which finds movies similar to each other based on user ratings.
- Or, use the **ALS model** to predict how similar different movies are to a target movie by checking the latent factors learned by the model.

Let's proceed with finding **movie-to-movie similarity** using the trained ALS model.

```
# Get the movie features from the ALS model
movie_factors = model.itemFactors

# Show movie factors (latent features learned by the model)
movie_factors.show(5)

# Now let's compute the similarity between movies by their latent features (dot product of the
features)
frompyspark.ml.linalgimportDenseVector
frompyspark.sqlimport functions as F

# UDF to calculate similarity
defcosine_similarity(v1, v2):
fromnumpyimport dot
fromnumpy.linalgimport norm
returnfloat(dot(v1, v2) / (norm(v1) * norm(v2)))

# Register the UDF
cosine_similarity_udf = F.udf(cosine_similarity)

# Let's compare movie 1 (The Matrix) with other movies
movie_id = 1# The Matrix
movie_row = movie_factors.filter(movie_factors.movieId == movie_id).collect()[0]
movie_vector = movie_row.features.toArray()

# Compute similarity between the selected movie and all other movies
movie_similarities = movie_factors.withColumn(
"similarity", cosine_similarity_udf(movie_factors.features, F.lit(movie_vector))
)

# Show top 5 similar movies to movie 1
movie_similarities.orderBy("similarity", ascending=False).show(5)
```

Step 6: Making Movie Recommendations

You can also use the ALS model to recommend movies to a user based on their previous ratings:

```
# Get top 5 movie recommendations for a specific user (e.g., user 1)
user_id = 1
user_recommendations = model.recommendForUserSubset(
ratings_df.filter(ratings_df.userId == user_id), numItems=5
)

# Show the recommendations
user_recommendations.show()
```

Step 7: Finding Similar Movies to a Movie Based on ALS Model

```
# Get top 5 similar movies for a given movie_id
defget_similar_movies(movie_id):
# Get the movie features for the given movie
movie_features = movie_factors.filter(movie_factors.movieId == movie_id).select("features").collect()[0][0]

# Compute cosine similarity between the movie's features and all other movie features
similarity_df = movie_factors.withColumn(
"similarity", cosine_similarity_udf(F.col("features"), F.lit(movie_features))
    )

# Sort by similarity and return the top 5 similar movies
returnsimilarity_df.orderBy("similarity", ascending=False).limit(6)  # Including the movie itself

# Get similar movies to movie 1 (The Matrix)
similar_movies = get_similar_movies(1)
similar_movies.show()
```

Example Output:

diff

```
+-------+-------------------+--------------------+
|movieId|            title|        similarity|
+-------+-------------------+--------------------+
|     1 |        The Matrix |                1.0|
|    101 |       Inception   |                0.9|
|    108 |       The Dark Kn  |                0.85|
|    110 |       Interstellar|               0.8|
|    109 |       The Prestige|               0.78|
+-------+-------------------+--------------------+
```

Practical 9- Statistical operation on data frame

1. **Descriptive Statistics**: Mean, median, standard deviation, count, min, and max.
2. **Correlation**: Pearson correlation between two columns.
3. **Covariance**: Covariance between two columns.
4. **Skewness and Kurtosis**: Measure the shape of the distribution.
5. **Statistical Summaries**: Summary statistics for multiple columns.

Use built-in **functions** from the **pyspark.sql.functions** module to simplify these operations.

Steps:

1. **Create a Sample DataFrame**: We'll start by creating a sample DataFrame.
2. **Descriptive Statistics**: Calculate basic statistics such as mean, standard deviation, and count.
3. **Correlation**: Calculate the correlation between numeric columns.
4. **Covariance**: Calculate the covariance between columns.
5. **Skewness& Kurtosis**: Calculate the skewness and kurtosis of a column.

Solution Code:

Step 1: Create a Sample DataFrame

We'll start by creating a simple DataFrame with numeric data that we can use to perform statistical operations.

```
frompyspark.sqlimportSparkSession
frompyspark.sqlimport functions as F

# Initialize the Spark session
spark = SparkSession.builder.appName("StatisticalOperations").getOrCreate()

# Sample data with numeric values
data = [
    ("Alice", 25, 10.5),
    ("Bob", 30, 12.5),
    ("Catherine", 35, 14.0),
```

```
    ("David", 40, 16.0),
    ("Eva", 45, 18.5)
]

# Define the schema
columns = ["name", "age", "score"]

# Create the DataFrame
df = spark.createDataFrame(data, columns)

# Show the DataFrame
df.show()
```

Output:

diff

```
+--------+---+-----+
|    name|age|score|
+--------+---+-----+
|   Alice| 25| 10.5|
|     Bob| 30| 12.5|
|Catherine| 35| 14.0|
|   David| 40| 16.0|
|     Eva| 45| 18.5|
+--------+---+-----+
```

Step 2: Descriptive Statistics

We can compute basic descriptive statistics for a numerical column or all numerical columns using the describe method.

```
# Calculate descriptive statistics for the entire DataFrame
df.describe().show()

# Calculate descriptive statistics for a specific column, e.g., "age"
```

df.select("age").describe().show()

Output:

lua

```
+-------+---+-----+------------------+
|summary|age|score|             name|
+-------+---+-----+------------------+
|  count|  5|    5|                 5|
|   mean| 35.0| 14.5|             null|
| stddev| 7.07106781187| 3.07155445072|             null|
|    min|  25| 10.5|            Alice|
|    max|  45| 18.5|              Eva|
+-------+---+-----+------------------+
```

Step 3: Correlation

We can calculate the **Pearson correlation** between two numeric columns to measure how strongly the columns are related.

```
# Calculate the correlation between "age" and "score"
correlation = df.stat.corr("age", "score")
print(f"Pearson correlation between 'age' and 'score': {correlation}")
```

Output:

sql

```
Pearson correlation between'age'and'score': 0.999290033418601
```

Step 4: Covariance

We can compute the **covariance** between two columns using the stat.cov method. Covariance measures how much two random variables change together.

```
# Calculate the covariance between "age" and "score"
```

covariance = df.stat.cov("age", "score")
print(f"Covariance between 'age' and 'score': {covariance}")

Output:

sql

Covariance between'age'and'score': 10.714285714285714.

Step 5: Skewness and Kurtosis

Skewness measures the asymmetry of the distribution of a column, while **Kurtosis** measures the "tailedness" of the distribution.

# Calculateskewness and kurtosis for the "score" column
skewness = df.select(F.skewness("score")).collect()[0][0]
kurtosis = df.select(F.kurtosis("score")).collect()[0][0]

print(f"Skewness of 'score': {skewness}")
print(f"Kurtosis of 'score': {kurtosis}")

Output:

arduino

Skewness of 'score': 0.19064396845767423
Kurtosis of 'score': -1.4704937501718193

Step 6: Statistical Summaries

You can generate various statistical summaries for multiple columns at once using the summary method. This method provides additional statistics such as **min, max, mean, count, stddev**, **median**, **25th percentile**, **75th percentile**, etc.

# Get a full statistical summary of the DataFrame
df.summary().show()

Get a statistical summary for specific columns

df.select("age", "score").summary().show()

Output:

lua

```
+-------+---+-----+
|summary|age|score|
+-------+---+-----+
|  count|  5|    5|
|   mean| 35.0| 14.5|
| stddev| 7.07106781187| 3.07155445072|
|    min|  25| 10.5|
|    25%|  30| 11.5|
|    50%|  35| 14.0|
|    75%|  40| 16.0|
|    max|  45| 18.5|
+-------+---+-----+
```

Practical 10- Using Spark ML to Produce Movie Recommendations

1. Load the MovieLens dataset.
2. Use **ALS (Alternating Least Squares)** for collaborative filtering.
3. Train the model on user-item ratings (movies).
4. Make movie recommendations for users.
5. Evaluate the model using **Root Mean Squared Error (RMSE)**.

Step-by-Step Guide:

Step 1: Load the Dataset

We'll first load the MovieLens dataset, which contains user ratings for various movies. The dataset typically contains columns such as userId, movieId, rating, and timestamp.

```
frompyspark.sqlimportSparkSession
# Create a Spark session
spark = SparkSession.builder.appName("MovieRecommendation").getOrCreate()

# Load the MovieLens ratings dataset
ratings_file_path = "/path/to/ratings.csv"# Change this path based on your file location
movies_file_path = "/path/to/movies.csv"# Change this path based on your file location

# Load the ratings and movies datasets
ratings_df = spark.read.option("header", "true").csv(ratings_file_path)
movies_df = spark.read.option("header", "true").csv(movies_file_path)

# Show the first few rows to inspect
ratings_df.show(5)
movies_df.show(5)
```

Sample data:

```
sql
ratings_df:
+------+-------+------+
```

|userId|movieId|rating|
+------+-------+------+
1	1	4.0
1	2	3.0
2	1	5.0
2	3	4.0
3	2	2.0
+------+-------+------+

movies_df:
+-------+--------------------+
|movieId| title|
+-------+--------------------+
|1| Toy Story (1995)|
2	Jumanji (1995)
3	Grumpier Old Men (1995)
4	Waiting to Exhale (1995)
+-------+--------------------+

Step 2: Data Preprocessing

In the MovieLens dataset, the columns userId and movieId are often represented as strings, so we need to cast them to integers for processing. Additionally, we'll convert the rating column to a float.

```
frompyspark.sqlimport functions as F

# Cast columns to appropriate data types
ratings_df = ratings_df.withColumn("userId", ratings_df["userId"].cast("int")) \
                       .withColumn("movieId", ratings_df["movieId"].cast("int")) \
                       .withColumn("rating", ratings_df["rating"].cast("float"))

ratings_df.printSchema()
```

Step 3: Split Data into Training and Test Sets

We will split the data into a **training set** (to train the model) and a **test set** (to evaluate the model).

```
# Split the ratings data into training and test sets
(training_data, test_data) = ratings_df.randomSplit([0.8, 0.2])

# Show a sample of the training data
training_data.show(5)
```

Step 4: Train the ALS (Alternating Least Squares) Model

Now, we will use Spark's **ALS** algorithm to create a collaborative filtering model. ALS tries to predict missing ratings based on the observed ratings and factorizes the user-item matrix.

```
frompyspark.ml.recommendationimport ALS
frompyspark.ml.evaluationimportRegressionEvaluator

# Define and train the ALS model
als = ALS(userCol="userId", itemCol="movieId", ratingCol="rating", coldStartStrategy="drop")
model = als.fit(training_data)

# Make predictions on the test data
predictions = model.transform(test_data)

# Show the predicted ratings
predictions.show(5)
```

Step 5: Evaluate the Model using RMSE

We will use **Root Mean Squared Error (RMSE)** to evaluate the quality of the model. RMSE is commonly used for evaluating recommendation systems because it measures the average magnitude of the prediction errors.

# Evaluate the model using RMSE
evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating", predictionCol="prediction")
rmse = evaluator.evaluate(predictions)

print(f"Root Mean Squared Error (RMSE) = {rmse}")

Step 6: Making Recommendations

Once we have a trained ALS model, we can use it to generate **movie recommendations**. For example, we can recommend top movies for a specific user or find similar movies for a particular movie.

6.1: Top N Movie Recommendations for a User

We can recommend the top N movies for a specific user by calling the recommendForUserSubset() method.

# Recommend top 5 movies for a user (e.g., user 1)
user_id = 1
user_recommendations = model.recommendForUserSubset(ratings_df.filter(ratings_df.userId == user_id), numItems=5)

# Show the recommendations
user_recommendations.show()

This will show the top 5 movies that the model recommends for the specified user.

6.2: Top N Movie Recommendations for All Users

We can also recommend top N movies for all users in the dataset.

Recommend top 5 movies for all users

```
all_users_recommendations = model.recommendForAllUsers(5)

# Show the recommendations
all_users_recommendations.show()
```

This will provide a DataFrame with userId and the top 5 recommended movies for each user.

6.3: Finding Similar Movies

We can find similar movies using the **item factors** (the learned feature vectors of the movies) and calculating the cosine similarity between them.

```
# Get the movie features learned by ALS
movie_factors = model.itemFactors

# Show the first 5 movie features
movie_factors.show(5)
```