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

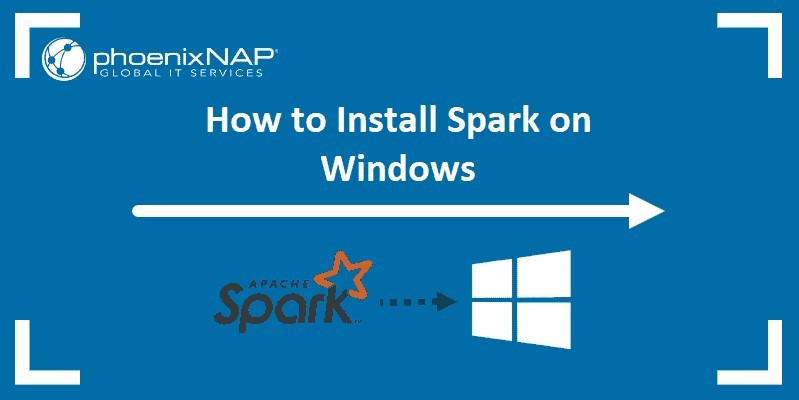
Practical 9- Statistical operation on data frame

Practical 10- Using Spark ML to Produce Movie Recommendations

**Practical 1- How to Set Up Spark on Window 10**

Apache Spark is an open-source framework that processes large volumes of stream data from multiple sources. Spark is used in distributed computing with machine learning applications, data analytics, and graph-parallel processing.

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



## Prerequisites

* A system running Windows 10
* A user account with administrator privileges (required to install software, modify file permissions, and modify system PATH)
* Command Prompt or Powershell
* A tool to extract .tar files, such as 7-Zip

# Install Apache Spark on Windows

Installing Apache Spark on Windows 10 may seem complicated to novice users, but this simple tutorial will have you up and running. If you already have Java 8 and 3 installed, you can skip the first two steps.

## Step 1: Install Java 8

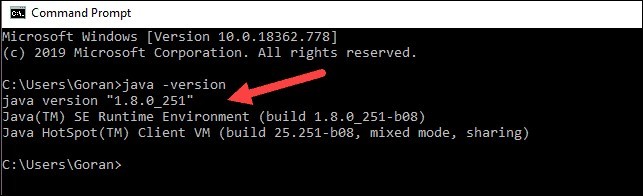
Apache Spark requires Java 8. You can check to see if Java is installed using the command prompt.

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

Type the following command in the command prompt:

java -version

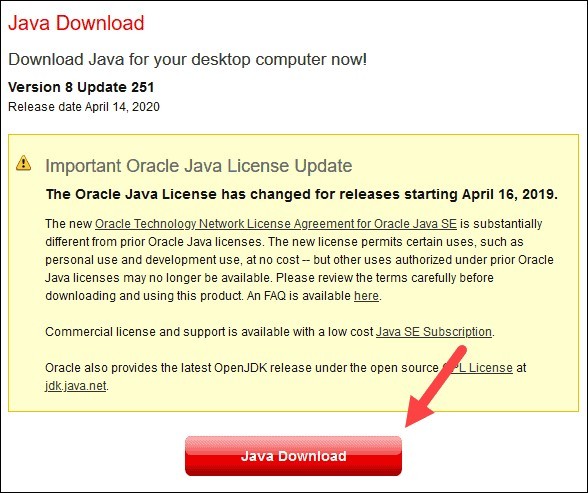
If Java is installed, it will respond with the following output:



Your version may be different. The second digit is the Java version – in this case, Java 8.

If you don’t have Java installed:

1. Open a browser window, and navigate to https://java.com/en/download/.



1. Click the **Java Download** button and save the file to a location of your choice.
2. Once the download finishes double-click the file to install Java.

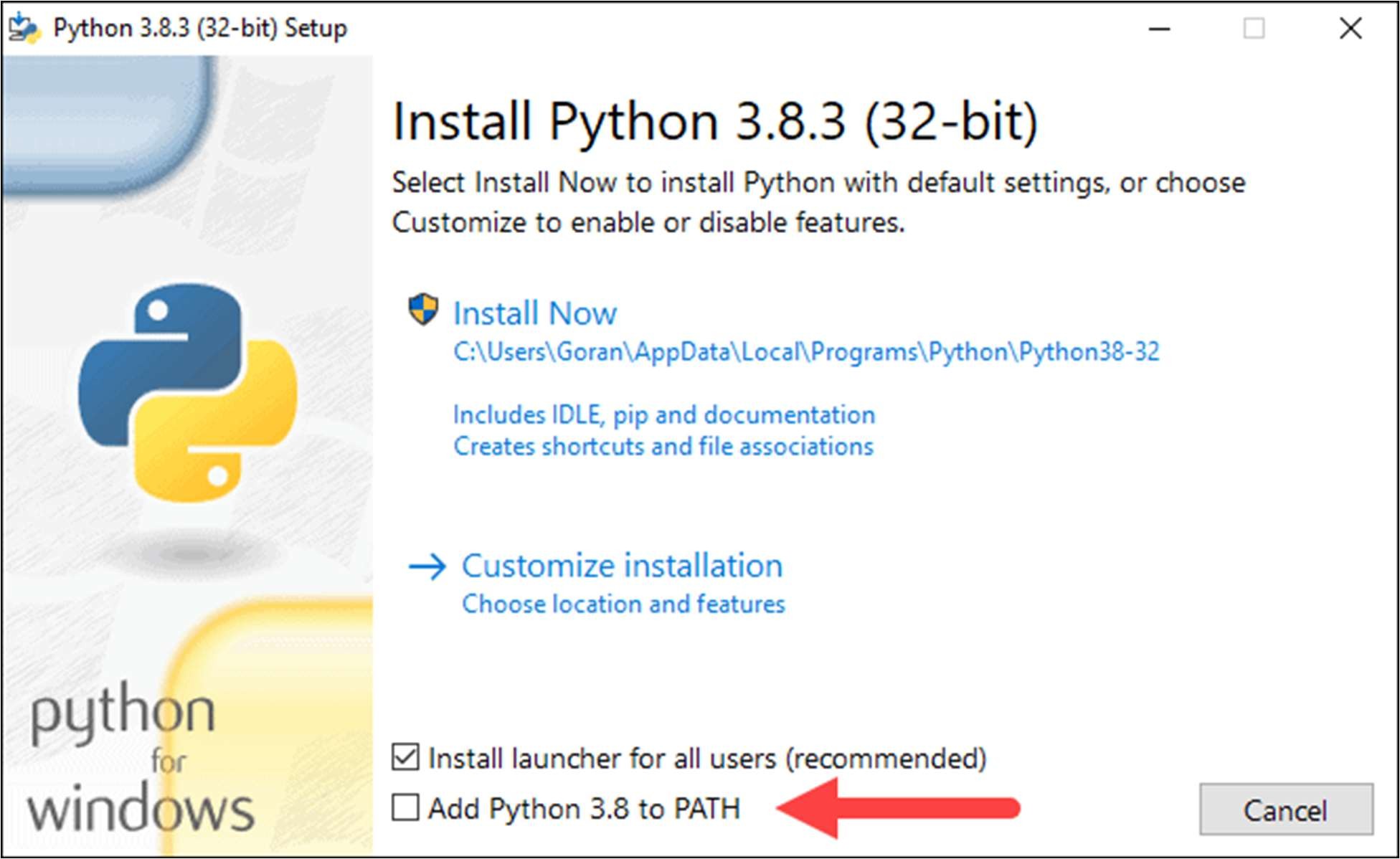
**Note:** At the time this article was written, the latest Java version is 1.8.0\_251. Installing a later version will still work. This process only needs the Java Runtime Environment (JRE) – the full Development Kit (JDK) is not required. The download link to JDK is https://[www.oracle.com/java/technologies/javase-](http://www.oracle.com/java/technologies/javase-) downloads.html.

## Step 2: Install

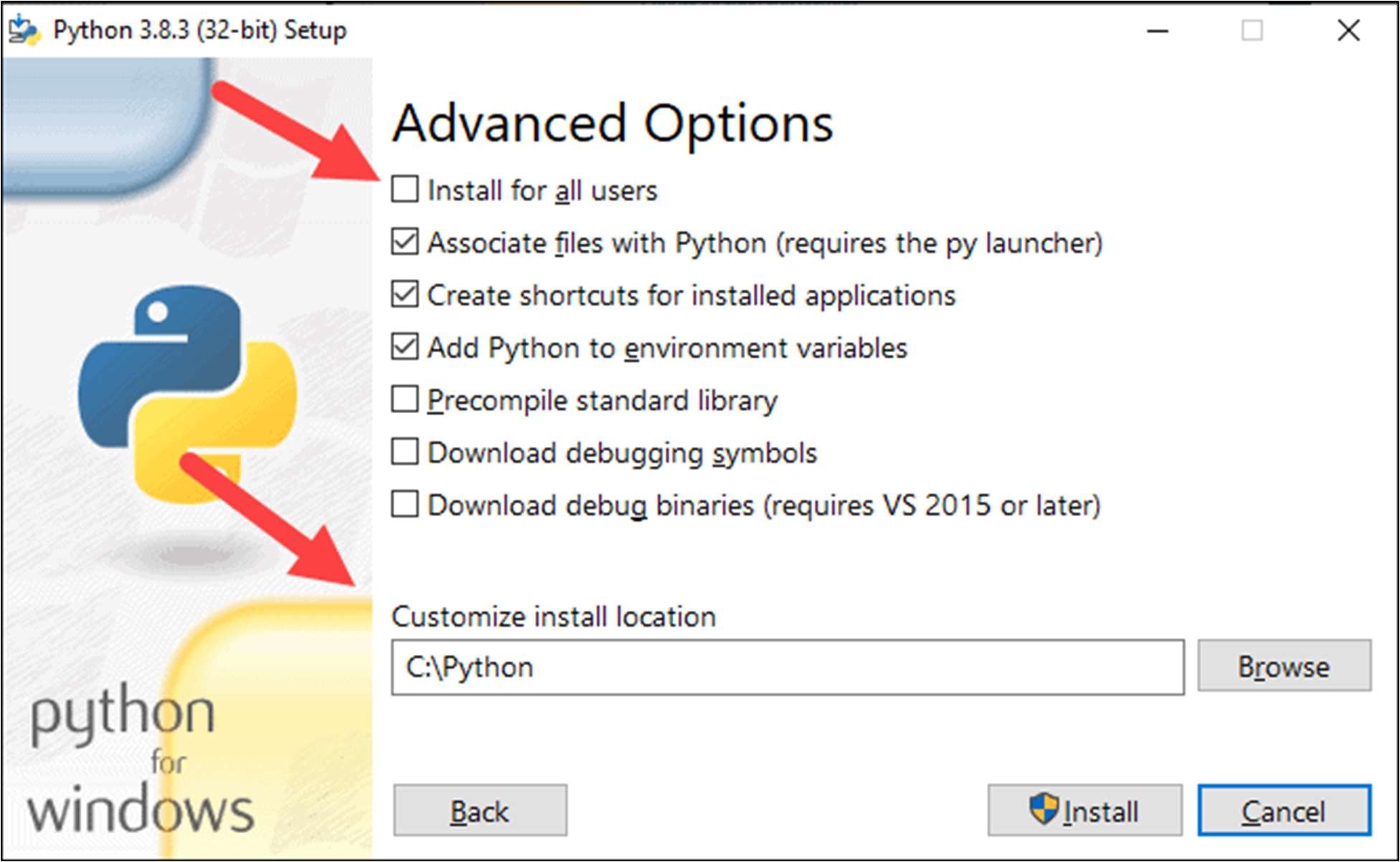
1. To install the package manager, navigate to https://[www..org/](http://www.python.org/) in your web browser.
2. Mouse over the **Download** menu option and click  **3.8.3**. 3.8.3 is the latest version at the time of writing the article.
3. Once the download finishes, run the file.



1. Near the bottom of the first setup dialog box, check off *Add 3.8 to PATH*. Leave the other box checked.
2. Next, click **Customize installation**.



1. You can leave all boxes checked at this step, or you can uncheck the options you do not want.
2. Click **Next**.
3. Select the box **Install for all users** and leave other boxes as they are.
4. Under *Customize install location,* click **Browse** and navigate to the C drive. Add a new folder and name it .
5. Select that folder and click **OK**.



1. Click **Install**, and let the installation complete.
2. When the installation completes, click the *Disable path length limit* option at the bottom and then click **Close**.
3. If you have a command prompt open, restart it. Verify the installation by checking the version of :

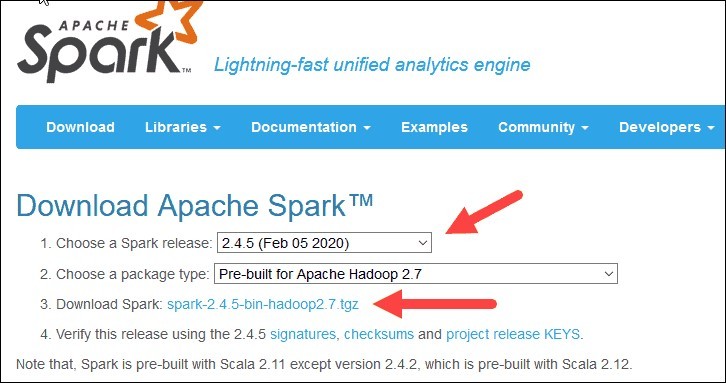
python –-version

The output should print  **3.8.3**.

**Note:** For detailed instructions on how to install 3 on Windows or how to troubleshoot potential issues, refer to our Install 3 on Windows guide.

Step 3: Download Apache Spark

1. Open a browser and navigate to https://spark.apache.org/downloads.html.
2. Under the *Download Apache Spark* heading, there are two drop-down menus. Use the current non-preview version.
   * In our case, in ***Choose a Spark release*** drop-down menu select **2.4.5 (Feb 05 2020)**.
   * In the second drop-down ***Choose a package type*,** leave the selection **Pre- built for Apache Hadoop 2.7**.
3. Click the ***spark-2.4.5-bin-hadoop2.7.tgz*** link.



1. A page with a list of mirrors loads where you can see different servers to download from. Pick any from the list and save the file to your Downloads folder.

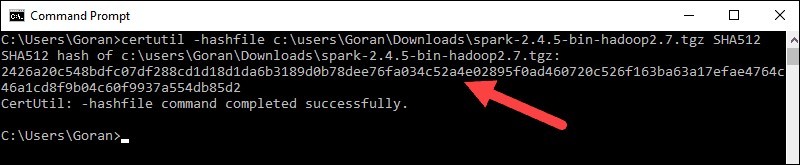
## Step 4: Verify Spark Software File

1. Verify the integrity of your download by checking the **checksum** of the file. This ensures you are working with unaltered, uncorrupted software.
2. Navigate back to the *Spark Download* page and open the **Checksum** link, preferably in a new tab.
3. Next, open a command line and enter the following command:

certutil -hashfile c:\users\username\Downloads\spark-2.4. 5-bin-hadoop2.7.tgz SHA512

1. Change the username to your username. The system displays a long

alphanumeric code, along with the message **Certutil: -hashfile completed successfully**.



1. Compare the code to the one you opened in a new browser tab. If they match, your download file is uncorrupted.

## Step 5: Install Apache Spark

Installing Apache Spark involves **extracting the downloaded file** to the desired location.

1. Create a new folder named *Spark* in the root of your C: drive. From a command line, enter the following:

cd \

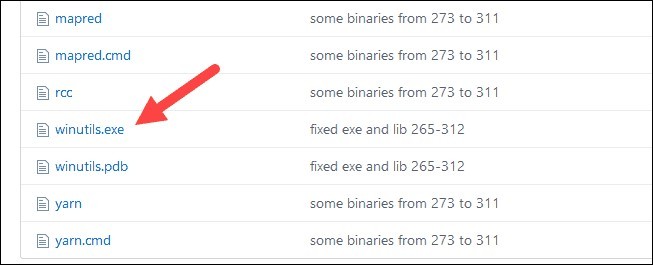
mkdir Spark

1. In Explorer, locate the Spark file you downloaded.
2. Right-click the file and extract it to *C:\Spark* using the tool you have on your system (e.g., 7-Zip).
3. Now, your *C:\Spark* folder has a new folder *spark-2.4.5-bin-hadoop2.7* with the necessary files inside.

## Step 6: Add winutils.exe File

Download the **winutils.exe** file for the underlying Hadoop version for the Spark installation you downloaded.

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

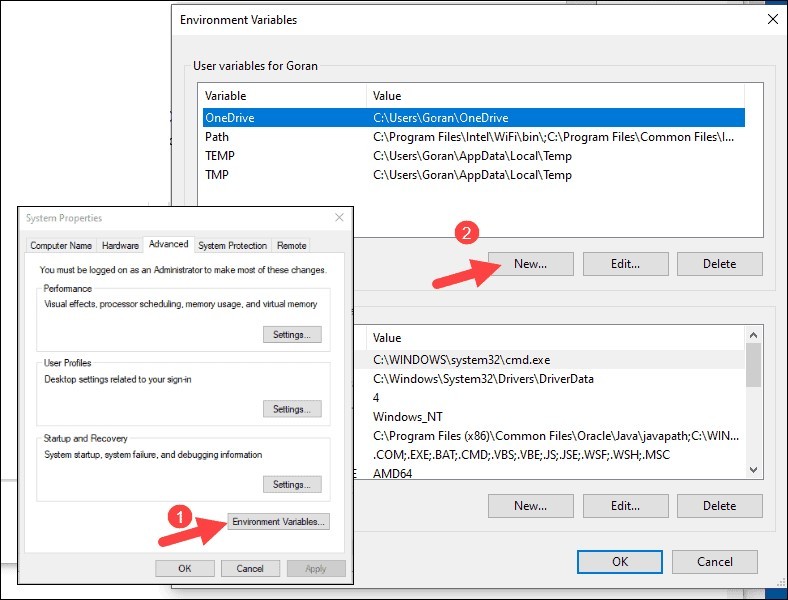


1. Find the **Download** button on the right side to download the file.
2. Now, create new folders ***Hadoop*** and **bin** on C: using Windows Explorer or the Command Prompt.
3. Copy the winutils.exe file from the Downloads folder to **C:\hadoop\bin**.

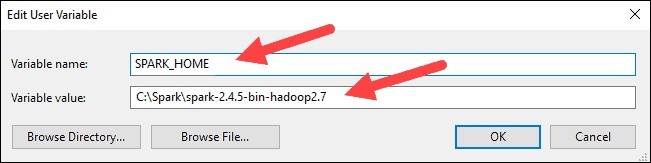
## Step 7: Configure Environment Variables

This step adds the Spark and Hadoop locations to your system PATH. It allows you to run the Spark shell directly from a command prompt window.

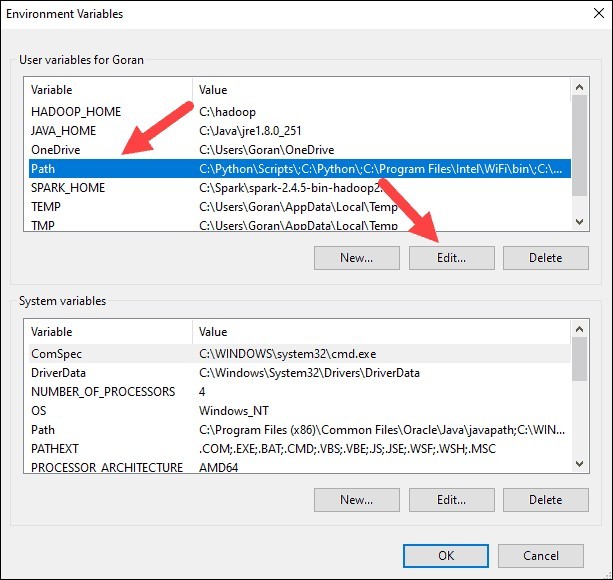
1. Click **Start** and type *environment*.
2. Select the result labeled ***Edit the system environment variables***.
3. A System Properties dialog box appears. In the lower-right corner, click **Environment Variables** and then click **New** in the next window.



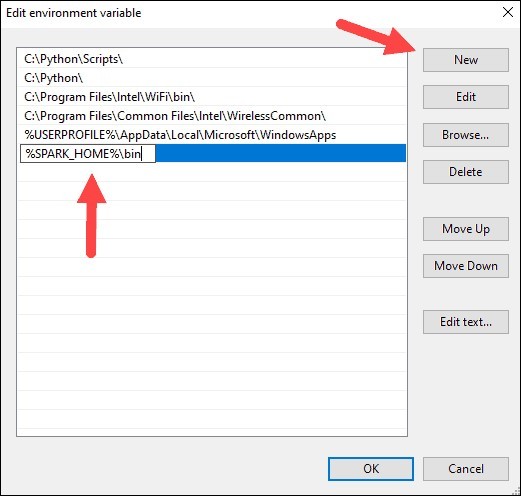
1. For *Variable Name* type ***SPARK\_HOME***.
2. For *Variable Value* type **C:\Spark\spark-2.4.5-bin-hadoop2.7** and click OK. If you changed the folder path, use that one instead.



1. In the top box, click the **Path** entry, then click **Edit**. Be careful with editing the system path. Avoid deleting any entries already on the list.



1. You should see a box with entries on the left. On the right, click **New**.
2. The system highlights a new line. Enter the path to the Spark folder **C:\Spark\spark-2.4.5-bin-hadoop2.7\bin**. We recommend using **%SPARK\_HOME%\bin** to avoid possible issues with the path.



1. Repeat this process for Hadoop and Java.
   * For Hadoop, the variable name is **HADOOP\_HOME** and for the value use the path of the folder you created earlier: **C:\hadoop.** Add **C:\hadoop\bin** to the **Path variable** field, but we recommend using **%HADOOP\_HOME%\bin**.
   * For Java, the variable name is **JAVA\_HOME** and for the value use the path to your Java JDK directory (in our case it’s **C:\Program Files\Java\jdk1.8.0\_251**).
2. Click **OK** to close all open windows.

**Note:** Star by restarting the Command Prompt to apply changes. If that doesn't work, you will need to reboot the system.

## Step 8: Launch Spark

1. Open a new command-prompt window using the right-click and **Run as administrator**:
2. To start Spark, enter:

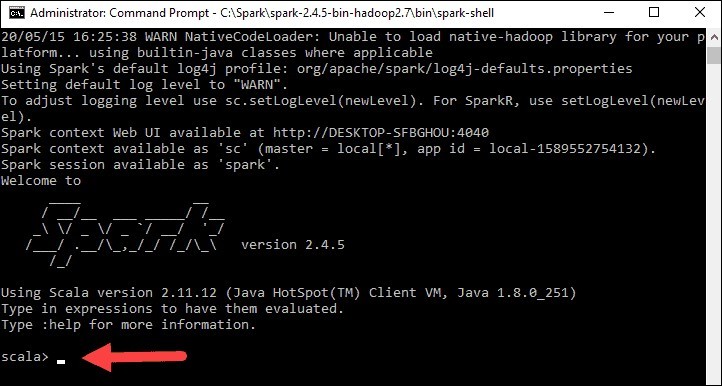
C:\Spark\spark-2.4.5-bin-hadoop2.7\bin\spark-shell

If you set the **environment path** correctly, you can type **spark-shell** to

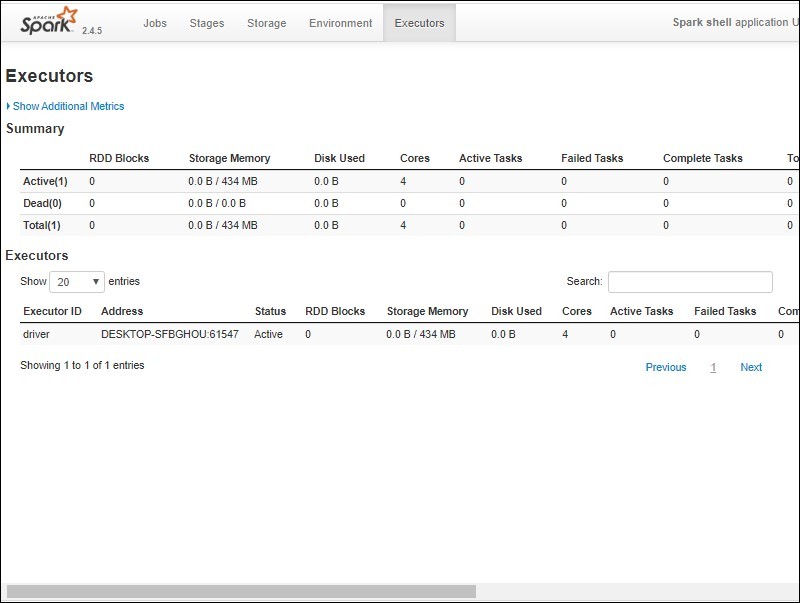
launch Spark.

1. The system should display several lines indicating the status of the application. You may get a Java pop-up. Select **Allow access** to continue.

Finally, the Spark logo appears, and the prompt displays the **Scala shell**.



1. , Open a web browser and navigate to **http://localhost:4040/**.
2. You can replace **localhost** with the name of your system.
3. You should see an Apache Spark shell Web UI. The example below shows the *Executors* page.



1. To exit Spark and close the Scala shell, press **ctrl-d** in the command- prompt window.

**Note:** If you installed , you can run Spark using with this command:

pyspark

Exit using **quit()**.

**Test Spark**

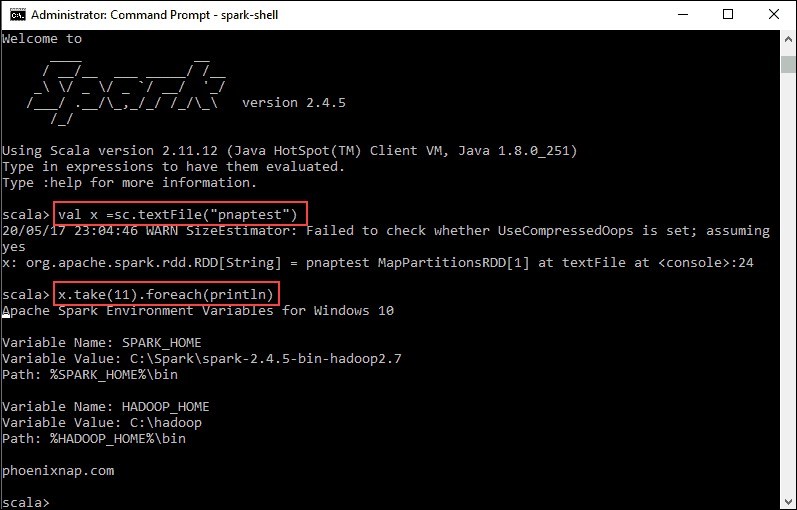
In this example, we will launch the Spark shell and use Scala to read the contents of a file. You can use an existing file, such as the *README* file in the Spark directory, or you can create your own. We created *pnaptest* with some text.

1. Open a command-prompt window and navigate to the folder with the file you want to use and launch the Spark shell.
2. First, state a variable to use in the Spark context with the name of the file. Remember to add the file extension if there is any.

val x =sc.textFile("pnaptest")

1. The output shows an RDD is created. Then, we can view the file contents by using this command to call an action:

x.take(11).foreach(println)



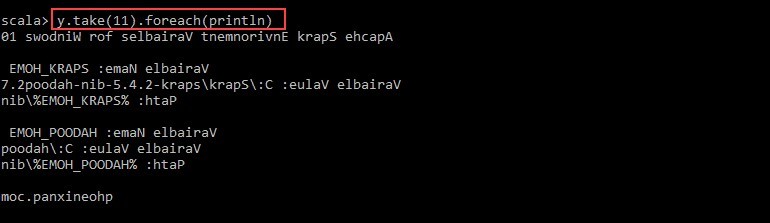
This command instructs Spark to print 11 lines from the file you specified. To perform an action on this file (**value x**), add another value **y**, and do a map transformation.

1. For example, you can print the characters in reverse with this command:

val y = x.map(\_.reverse)

1. The system creates a child RDD in relation to the first one. Then, specify how many lines you want to print from the value **y**:

y.take(11).foreach(println)



The output prints 11 lines of the *pnaptest* file in the reverse order. When done, exit the shell using **ctrl-d**.

## Conclusion

You should now have a working installation of Apache Spark on Windows 10 with all dependencies installed. Get started running an instance of Spark in Windows environment.

**Practical 2- 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:**

from pyspark import SparkContext

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).

from pyspark import SparkContext

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:**

from pyspark import SparkContext

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:**

from pyspark import SparkContext

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.

from pyspark import SparkContext

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]

1. 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]

1. 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]

1. 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)]

1. 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])]

1. 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))]

1. 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]

1. count():

Description: Returns the number of elements in the RDD.

Example:

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

result = rdd.count()

print(result) # 4

1. 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

1. first():

Description: Returns the first element of the RDD.

Example:

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

result = rdd.first()

print(result) # 1

1. 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]

1. 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")

1. 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

from pyspark.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

from pyspark.sql import functions as F

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

from datetime import datetime

# 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")

1. **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()

1. **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 30 with 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

from pyspark.sql import SparkSession

from pyspark.sql import 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 in broadcast\_movies\_df.value}

# Define a UDF (User Defined Function) to map movie IDs to movie names

def get\_movie\_name(movie\_id):

return movie\_dict.get(movie\_id, "Unknown")

# Register the UDF

from pyspark.sql.functions import udf

from pyspark.sql.types import StringType

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

from pyspark.sql import SparkSession

from pyspark.sql.functions import col

from pyspark.ml.recommendation import ALS

from pyspark.ml.evaluation import RegressionEvaluator

import os

# 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)

from pyspark.ml.linalg import DenseVector

from pyspark.sql import functions as F

# UDF to calculate similarity

def cosine\_similarity(v1, v2):

from numpy import dot

from numpy.linalg import norm

return float(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

def get\_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

return similarity\_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.

from pyspark.sql import SparkSession

from pyspark.sql import 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.

# Calculate skewness 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.

from pyspark.sql import SparkSession

# 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.

from pyspark.sql import 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.

from pyspark.ml.recommendation import ALS

from pyspark.ml.evaluation import RegressionEvaluator

# 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)