Practical 1
Installation of Spark

Aim: To install spark

Theory: 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.

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/.

2. Click the **Java Download** button and save the file to a location of your choice.

3. 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-downloads.html.

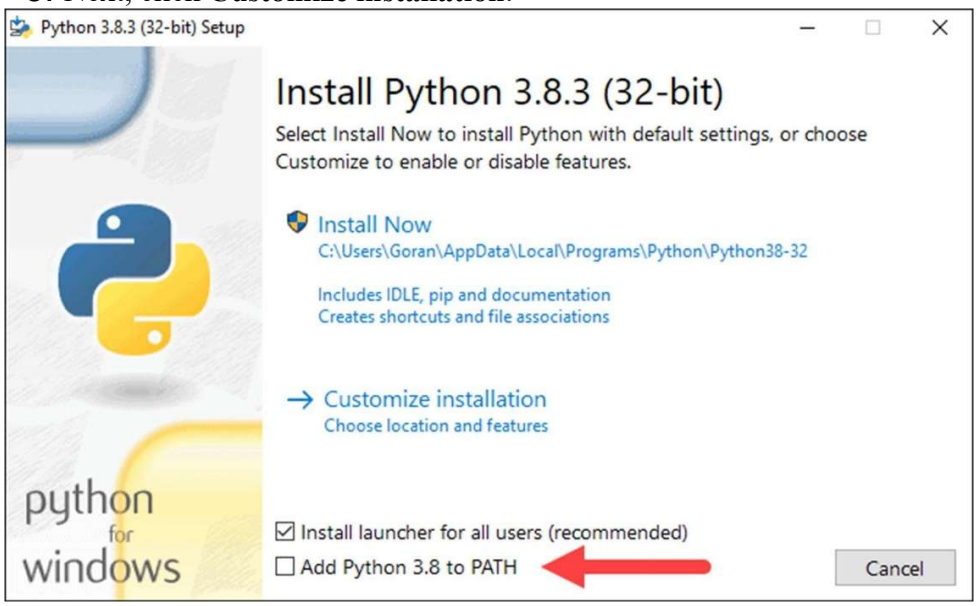
Step 2: Install

1. To install the package manager, navigate to https://www..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.

4. Near the bottom of the first setup dialog box, check off *Add 3.8 to PATH*. Leave the other box checked.

5. 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**
6. Click **Install**, and let the installation complete.
7. When the installation completes, click the *Disable path length limit* option at the bottom and then click **Close**.
8. If you have a command prompt open, restart it. Verify the installation by checking the version of :

```
python --version
```

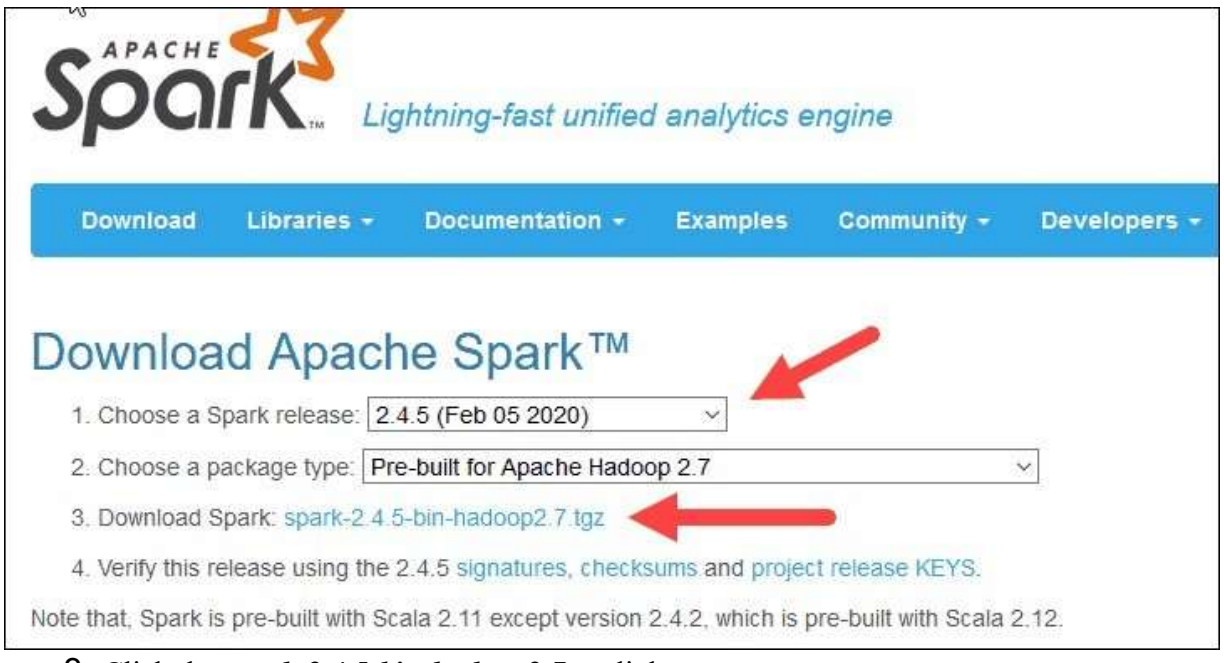
The output should print **3.8.3**.

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

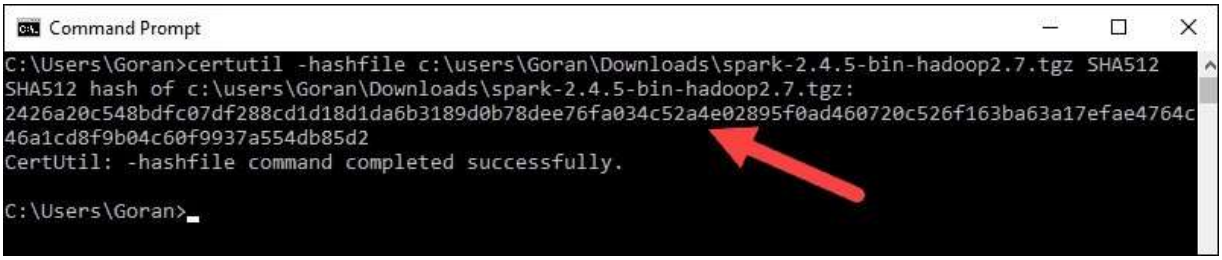
4. A page with a list of mirrors loads where you can see different servers to download from. Pick any from the list and save the file to your Downloads folder.

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
```

4. Change the username to your username. The system displays a long alphanumeric code, along with the message **Certutil: -hashfile completed successfully**.



5. 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.

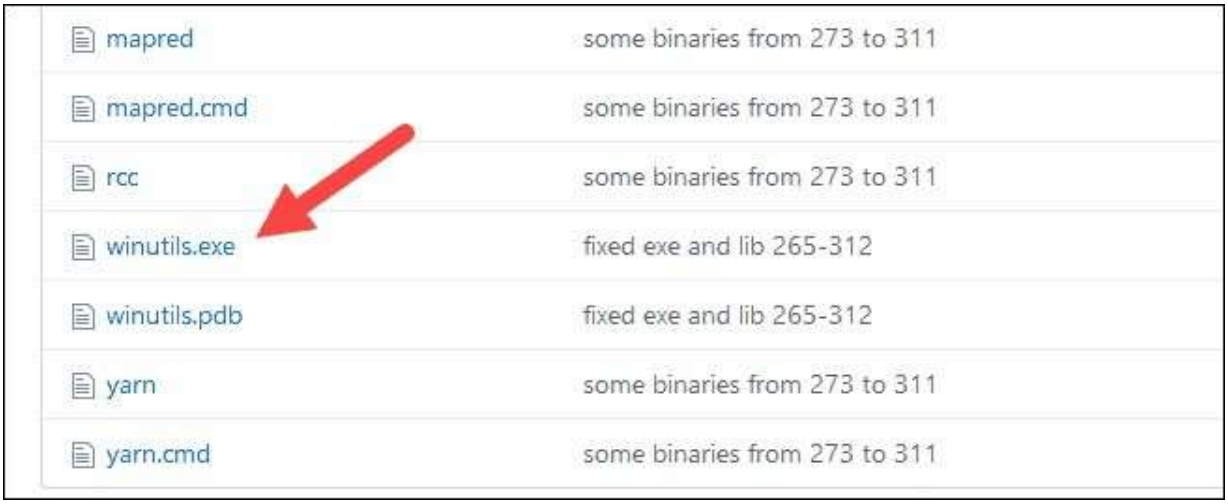
1. Create a new folder named *Spark* in the root of your C: drive. From a command line, enter the following:

```
cd \

mkdir Spark
```

2. In Explorer, locate the Spark file you downloaded.
3. Right-click the file and extract it to *C:\Spark* using the tool you have on your system (e.g., 7-Zip).
4. 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.



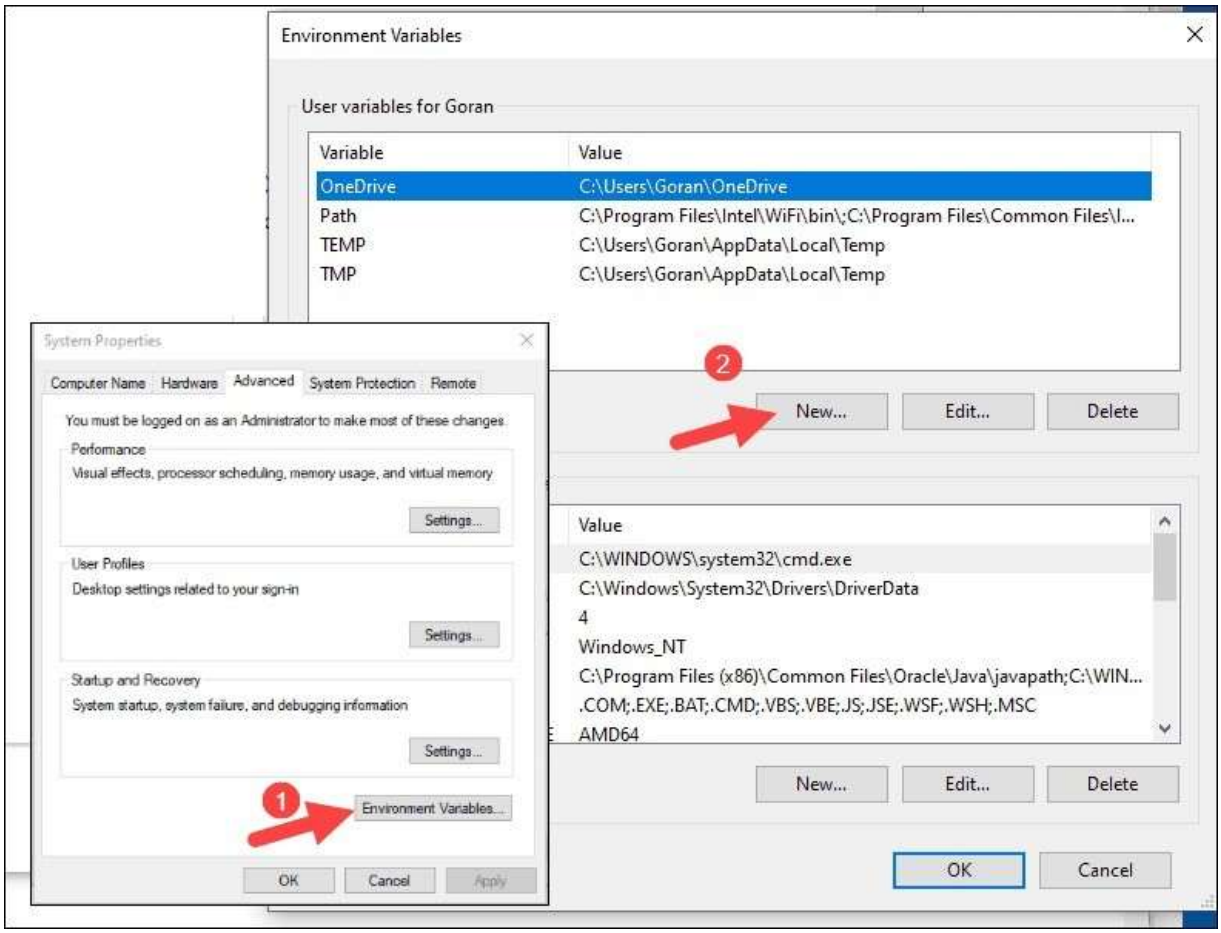
2. Find the **Download** button on the right side to download the file.

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

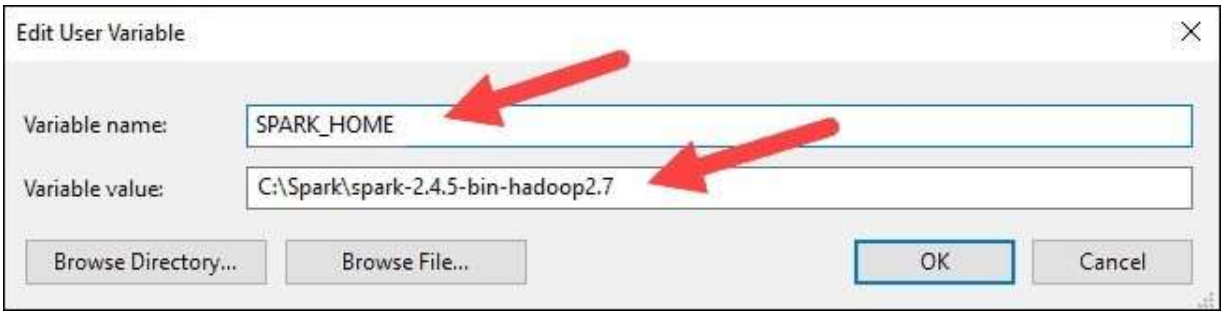
4. Copy the winutils.exe file from the Downloads folder to **C:\hadoop\bin**.

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.

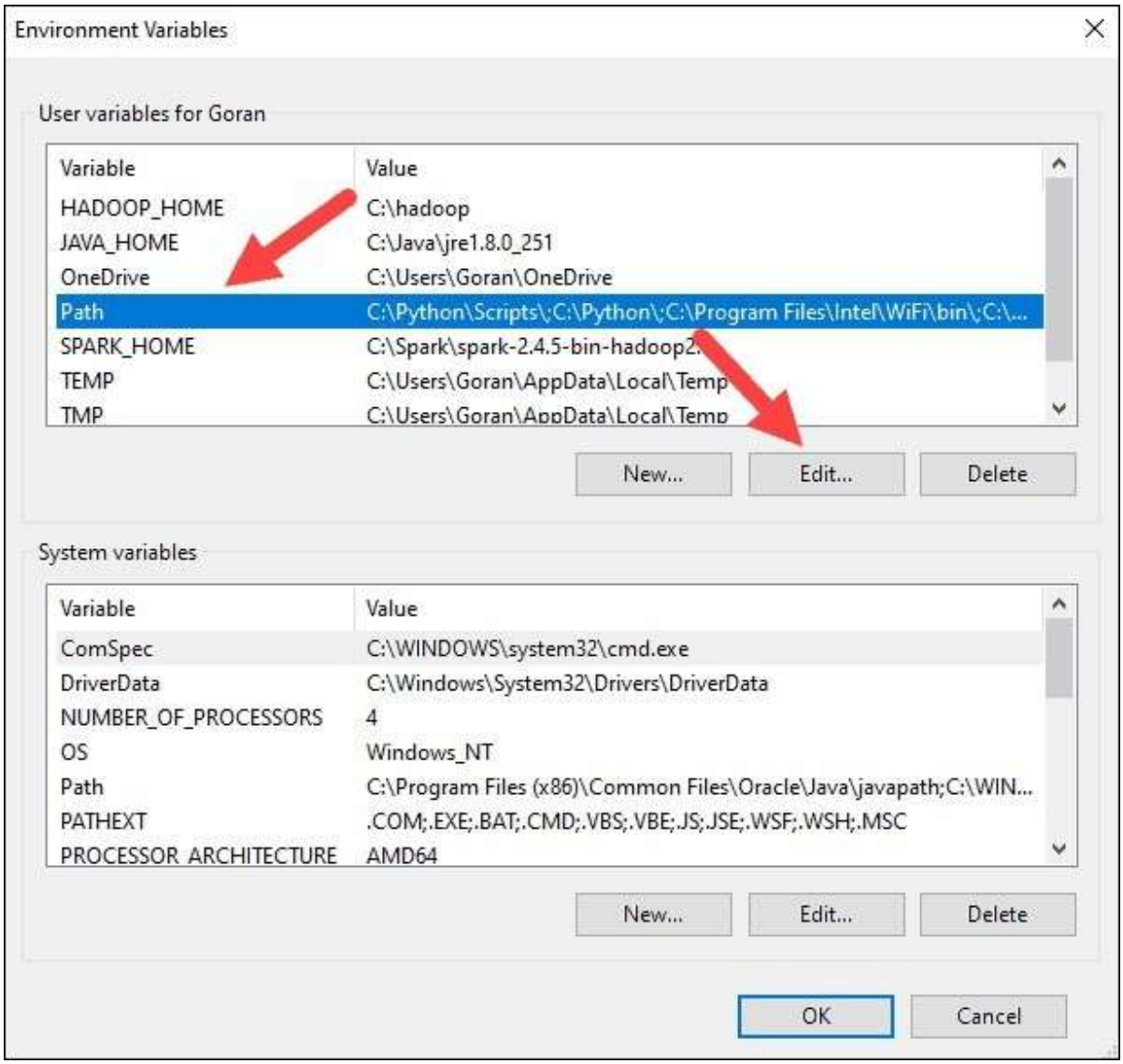
1. Click **Start** and type *environment*.
2. Select the result labeled ***Edit the system environment variables***.
3. A System Properties dialog box appears. In the lower-right corner, click **Environment Variables** and then click **New** in the next window.



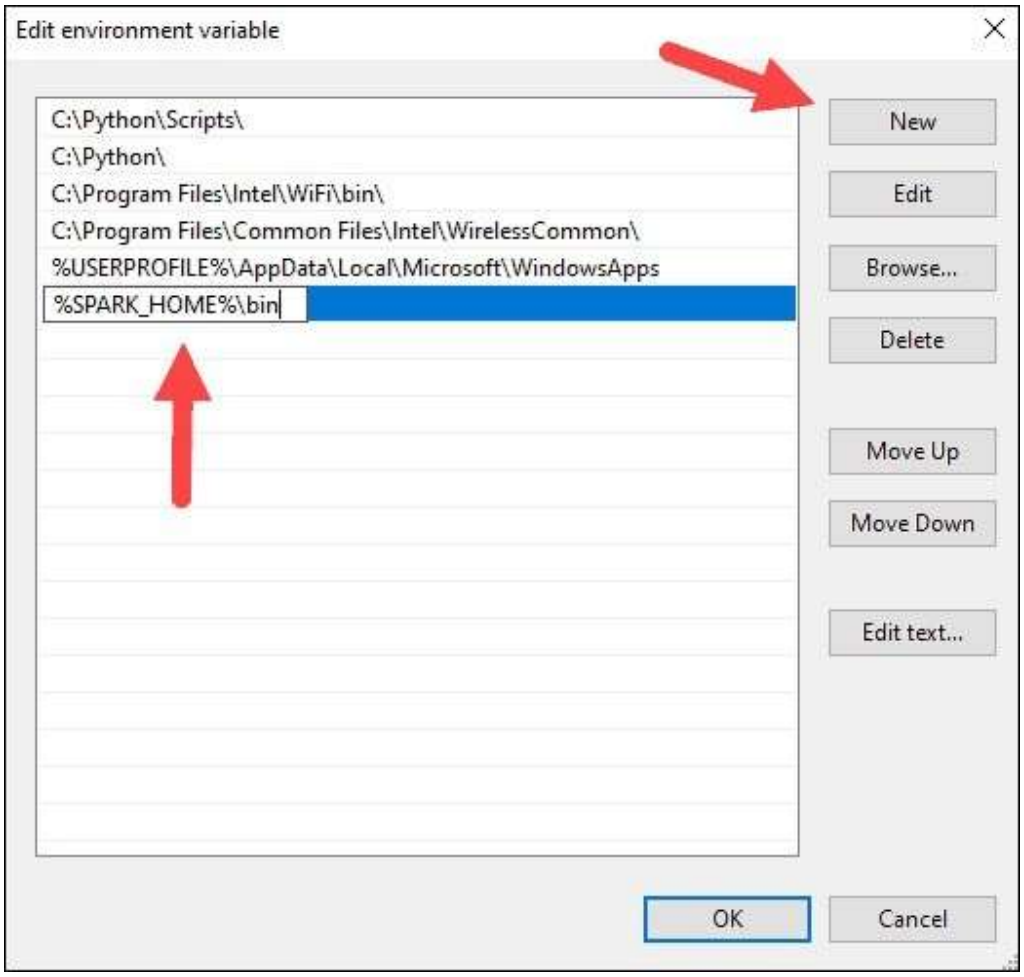
4. For *Variable Name* type ***SPARK_HOME***.
5. For *Variable Value* type **C:\Spark\spark-2.4.5-bin-hadoop2.7** and click OK. If you changed the folder path, use that one instead.



6. 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.



7. You should see a box with entries on the left. On the right, click **New**.
8. 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.



9. 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**).

10. 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**:

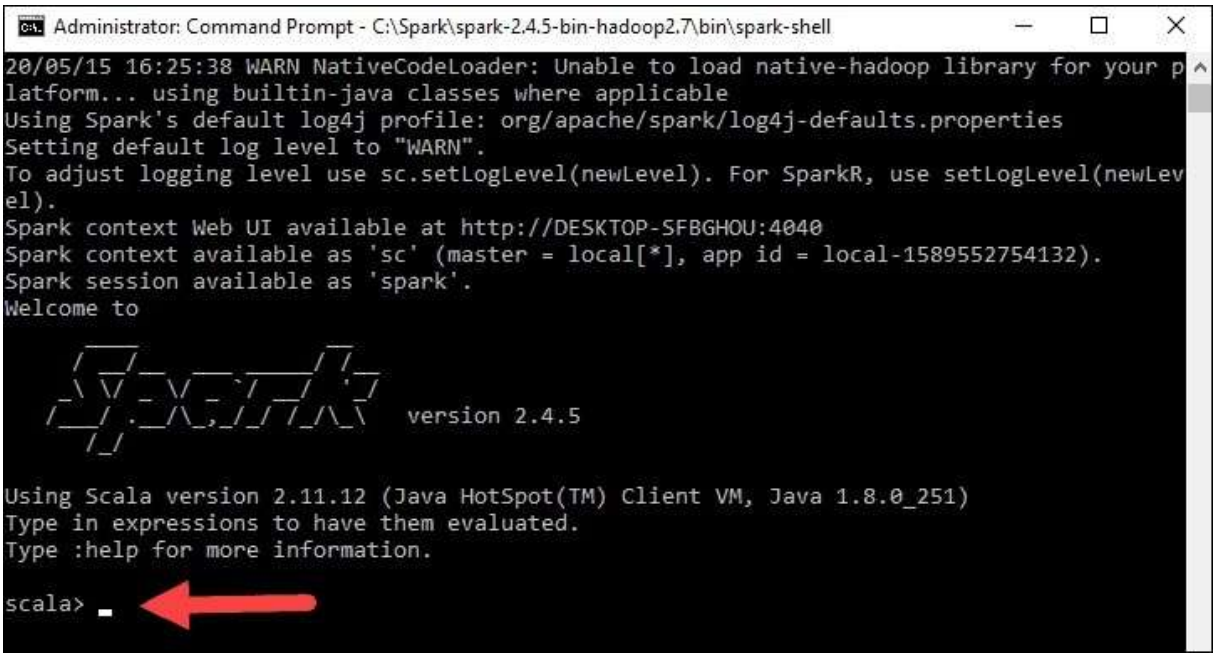
2. To start Spark, enter:

```
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

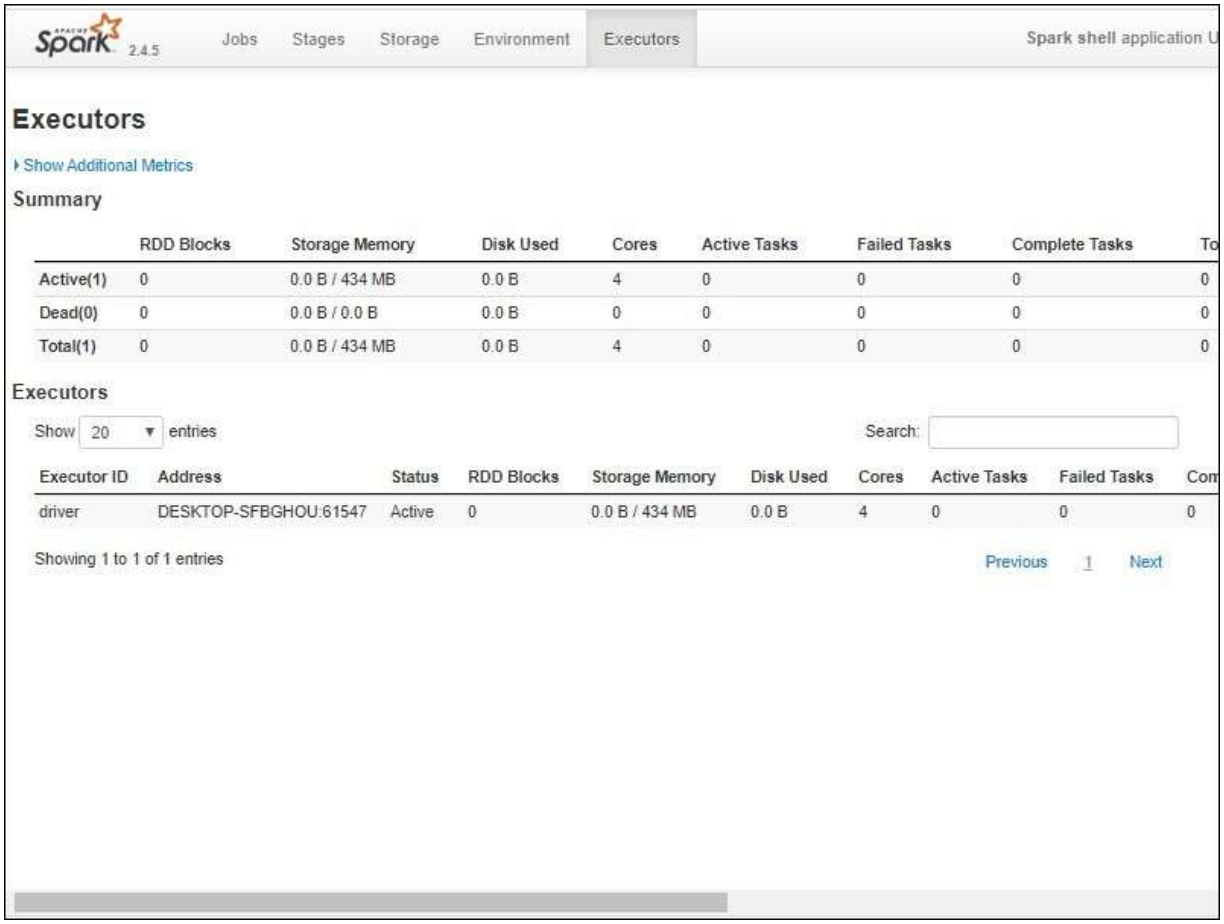
launch Spark.

3. 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**.

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



7. 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

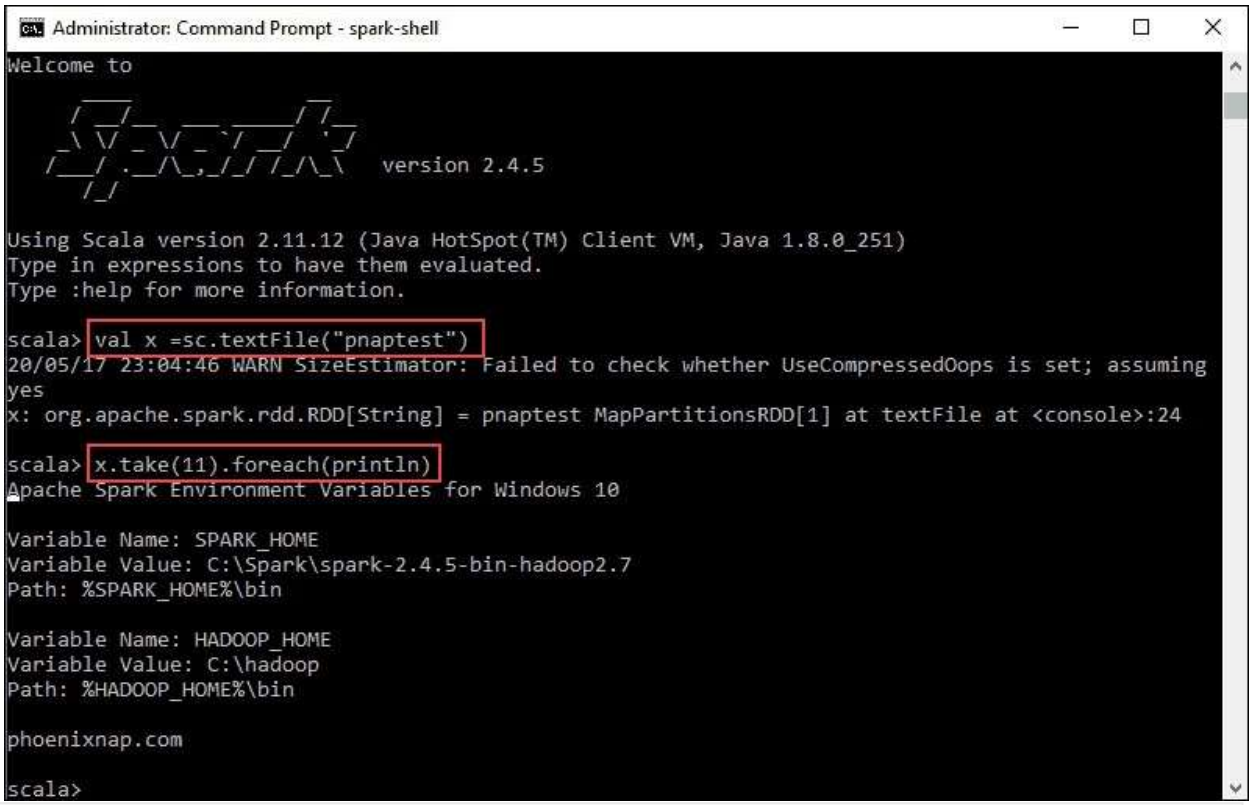
In this example, we will launch the Spark shell and use Scala to read the contents of a file. You can use an existing file, such as the *README* file in the Spark directory, or you can create your own. We created *pnaptest* with some text.

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")
```

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



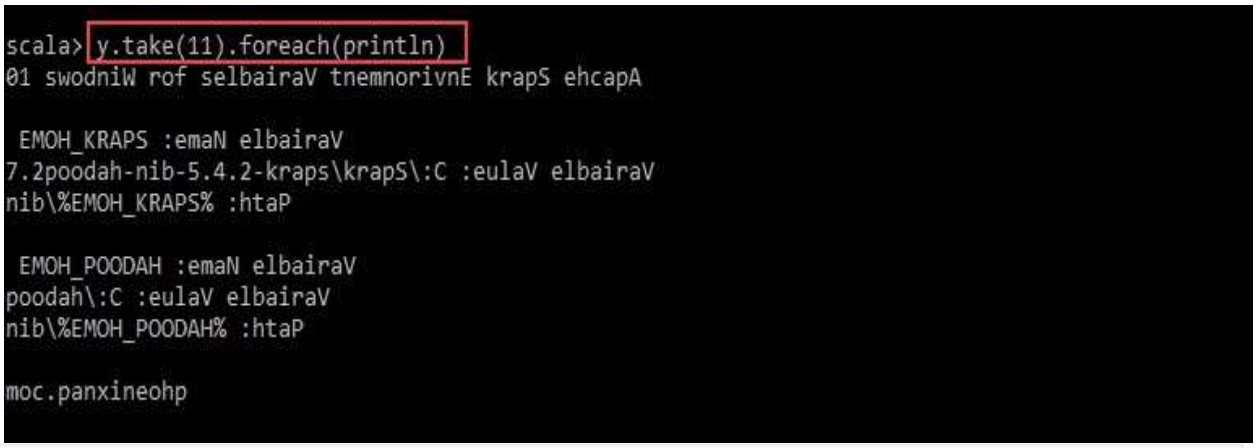

```
x.take(11).foreach(println)
```

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.

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

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

5. 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**.

Practical 2
How to create RDD

Aim: To create RDD and perform parallelize & other functions

Thoery:

1. Parallelizing a Collection
Creates an RDD from a local list to enable parallel data processing.

2. Loading Data from External Storage
Reads a file from distributed storage and converts it into an RDD.

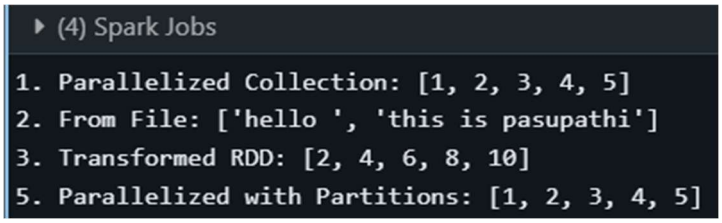
3. From Existing RDD (Transformation)
Applies a function to each element of an existing RDD to create a new one.

4. Parallelize with More Partitions
Creates an RDD with a custom number of partitions to optimize task distribution.

Code:

```
from pyspark import SparkContext
sc = SparkContext.getOrCreate()

# 1. Parallelizing a Collection
data = [1, 2, 3, 4, 5]
rdd1 = sc.parallelize(data)
print("1. Parallelized Collection:", rdd1.collect())

# 2. Loading Data from External Storage (Ensure the path exists in your Databricks environment)
# Replace with an actual file path in Databricks, like "/FileStore/sample.txt"
try:
    rdd2 = sc.textFile("/FileStore/tables/sample.txt")
    print("2. From File:", rdd2.collect())
except Exception as e:
    print("2. From File: File not found or path error -", e)

# 3. From Existing RDD (Transformation)
rdd3 = sc.parallelize([1, 2, 3, 4, 5])
new_rdd3 = rdd3.map(lambda x: x * 2)
print("3. Transformed RDD:", new_rdd3.collect())

# 4. Parallelize with More Partitions
rdd5 = sc.parallelize([1, 2, 3, 4, 5], 4)
print("5. Parallelized with Partitions:", rdd5.collect())
```

Output:



Practical 3
Transformation and action on RDD

Aim: To perform transformation and action on RDD

Theory:

RDD Transformations (Lazy)

They return a new RDD, executed only when an action is called.

- **map()** – Applies a function to each element.
- **filter()** – Keeps elements that match a condition.
- **flatMap()** – Like map but flattens the result.
- **reduceByKey()** – Merges values with the same key using a function.
- **groupByKey()** – Groups values by key.
- **join()** – Joins two RDDs by matching keys.
- **distinct()** – Removes duplicate elements.

RDD Actions (Trigger Execution)

They execute the transformations and return results.

- **collect()** – Returns all RDD elements as a list.
- **count()** – Counts total elements.
- **reduce()** – Reduces elements using a function.
- **first()** – Returns the first element.
- **take(n)** – Returns the first n elements.
- **saveAsTextFile()** – Saves RDD to a file.
- **countByKey()** – Counts occurrences of each key.

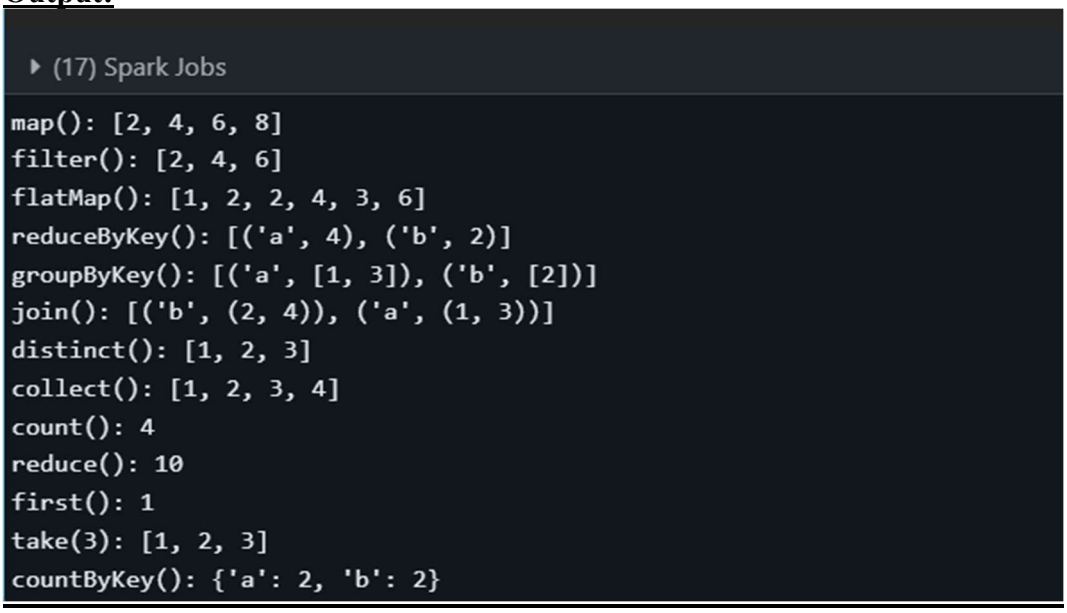
Code:

```
# RDD Transformations

# 1. map()
rdd = sc.parallelize([1, 2, 3, 4])
result = rdd.map(lambda x: x * 2)
print("map():", result.collect())  # [2, 4, 6, 8]

# 2. filter()
rdd = sc.parallelize([1, 2, 3, 4, 5, 6])
result = rdd.filter(lambda x: x % 2 == 0)
print("filter():", result.collect())  # [2, 4, 6]

# 3. flatMap()
rdd = sc.parallelize([1, 2, 3])
result = rdd.flatMap(lambda x: (x, x * 2))
print("flatMap():", result.collect())  # [1, 2, 2, 4, 3, 6]

# 4. reduceByKey()
rdd = sc.parallelize([('a', 1), ('b', 2), ('a', 3)])
result = rdd.reduceByKey(lambda x, y: x + y)
print("reduceByKey():", result.collect())  # [('a', 4), ('b', 2)]

# 5. groupByKey()
rdd = sc.parallelize([('a', 1), ('b', 2), ('a', 3)])
result = rdd.groupByKey().mapValues(list)
print("groupByKey():", result.collect())  # [('a', [1, 3]), ('b', [2])]
```

```
# 6. join()
rdd1 = sc.parallelize([('a', 1), ('b', 2)])
rdd2 = sc.parallelize([('a', 3), ('b', 4)])
result = rdd1.join(rdd2)
print("join():", result.collect())  # [('a', (1, 3)), ('b', (2, 4))]

# 7. distinct()
rdd = sc.parallelize([1, 2, 3, 2, 1])
result = rdd.distinct()
print("distinct():", result.collect())  # [1, 2, 3]
```

```
# RDD Actions

# 1. collect()
rdd = sc.parallelize([1, 2, 3, 4])
result = rdd.collect()
print("collect():", result)  # [1, 2, 3, 4]

# 2. count()
rdd = sc.parallelize([1, 2, 3, 4])
result = rdd.count()
print("count():", result)  # 4

# 3. reduce()
rdd = sc.parallelize([1, 2, 3, 4])
result = rdd.reduce(lambda x, y: x + y)
print("reduce():", result)  # 10

# 4. first()
rdd = sc.parallelize([1, 2, 3, 4])
result = rdd.first()
print("first():", result)  # 1

# 5. take(n)
rdd = sc.parallelize([1, 2, 3, 4])
result = rdd.take(3)
print("take(3):", result)  # [1, 2, 3]

# 6. saveAsTextFile() - Will write to DBFS (Databricks File System)
rdd = sc.parallelize([1, 2, 3, 4])
rdd.saveAsTextFile("dbfs:/tmp/output.txt")  # Use dbfs: URI scheme in Databricks

# 7. countByKey()
rdd = sc.parallelize([('a', 1), ('b', 2), ('a', 3), ('b', 4)])
result = rdd.countByKey()
print("countByKey():", dict(result))  # {'a': 2, 'b': 2}
```

Output:


(17) Spark Jobs
map(): [2, 4, 6, 8]
filter(): [2, 4, 6]
flatMap(): [1, 2, 2, 4, 3, 6]
reduceByKey(): [('a', 4), ('b', 2)]
groupByKey(): [('a', [1, 3]), ('b', [2])]
join(): [('b', (2, 4)), ('a', (1, 3))]
distinct(): [1, 2, 3]
collect(): [1, 2, 3, 4]
count(): 4
reduce(): 10
first(): 1
take(3): [1, 2, 3]
countByKey(): {'a': 2, 'b': 2}

Practical 4
Counting Word Occurrences using flat map()

Aim: To count the frequency of each word in the given text using RDD operations in PySpark.

Theory:

Apache Spark uses **RDDs (Resilient Distributed Datasets)** for distributed data processing.
The parallelize() method converts local data into an RDD.
flatMap() is used to split each line into individual words.
map() transforms each word into a key-value pair like (word, 1).
reduceByKey() aggregates the counts of each word.
collect() fetches the result back to the driver for display.

Code:

```
from pyspark import SparkContext
sc = SparkContext.getOrCreate()

# Sample text data
text_data = [
    "hi guys",
    "good morning",
    "this is pasupathi",
    "hope u guys are doing well"
]

# Create an RDD from the sample data
rdd = sc.parallelize(text_data)

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

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

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

# Collect the result and print it
word_counts = word_counts_rdd.collect()
for word, count in word_counts:
    print(f"{word}: {count}"
```

Output:


(1) Spark Jobs
good: 1
are: 1
this: 1
guys: 2
morning: 1
doing: 1
well: 1
hi: 1
pasupathi: 1
is: 1
hope: 1
u: 1

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

Aim: To perform SQL queries and DataFrame operations on structured data using PySpark in Databricks

Theory:
A **PySpark DataFrame** is created using structured data representing employees with name, age, and department.
The DataFrame is registered as a **temporary SQL view** using createOrReplaceTempView(), enabling SQL-style queries.
SQL Queries are executed using spark.sql() to:

- Select all employee records.
- Filter employees with age greater than 30.
- Group employees by department and compute average age.

DataFrame API is used for the same tasks without SQL:

- filter() to select employees older than 30.
- groupBy().avg() to compute department-wise average age.
- withColumn() adds a new column age_category using when() and otherwise() conditions to classify age.

Code:

```
 # 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

# a) Query to select all rows
sql_query = "SELECT * FROM employees"
result = spark.sql(sql_query)
print("All Employees:")
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)
print("Employees older than 30:")
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)
print("Average age by department:")
result.show()

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

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

# b) Group by department and calculate the average age
grouped_df = df.groupBy("department").avg("age")
print("Average age by department (DataFrame API):")
grouped_df.show()

# c) Add a new column with age category
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"))
print("Employees with age categories:")
df_with_new_col.show()
```

Output:

```
All Employees:
+-------+---+-----------+
|   name|age| department|
+-------+---+-----------+
|  Alice| 29|Engineering|
|    Bob| 35|      Sales|
|Charlie| 40|Engineering|
|  David| 30|         HR|
|    Eva| 25|      Sales|
+-------+---+-----------+

Employees older than 30:
+-------+---+-----------+
|   name|age| department|
+-------+---+-----------+
|    Bob| 35|      Sales|
|Charlie| 40|Engineering|
+-------+---+-----------+
```

```
Average age by department:
+-----------+-------+
| department|avg_age|
+-----------+-------+
|Engineering|   34.5|
|      Sales|   30.0|
|         HR|   30.0|
+-----------+-------+

Filtered employees (age > 30):
+-------+---+-----------+
|   name|age| department|
+-------+---+-----------+
|    Bob| 35|      Sales|
|Charlie| 40|Engineering|
+-------+---+-----------+
```

```
Average age by department (DataFrame API):
+-----------+--------+
| department|avg(age)|
+-----------+--------+
|Engineering|    34.5|
|      Sales|    30.0|
|         HR|    30.0|
+-----------+--------+

Employees 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 transformation

Aim: To perform data analysis and customer segmentation using PySpark DataFrame operations and SQL queries in Databricks.

Theory:
This practical demonstrates how to use **PySpark** for analyzing customer data. It covers:

- Creating a DataFrame with schema
- Filtering data based on conditions
- Performing grouping and aggregation (e.g., average, sum)
- Creating new columns using conditional logic
- Running SQL queries on DataFrames by creating temporary views
- Segmenting customers into loyalty tiers based on their spending

Code:

```
from pyspark.sql import functions as F
from pyspark.sql.types import StructType, StructField, IntegerType, StringType, FloatType, DateType
from datetime import datetime

# Step 1: Create a sample DataFrame with customer data
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)),
]

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)
])

df = spark.createDataFrame(data, schema)

print(" 1.Initial Customer DataFrame:")
df.show()

# Step 2: Filter customers older than 30 and spend > 200
filtered_df = df.filter((df.age > 30) & (df.total_spend > 200))
print(" 2.Customers older than 30 and total spend > 200:")
filtered_df.show()

# Step 3a: Group by gender and calculate average spend
gender_avg_spend_df = df.groupBy("gender").agg(F.avg("total_spend").alias("avg_spend"))
```

```
print(" 3.Average total spend by gender:")
gender_avg_spend_df.show()

# Step 3b: Add age group and aggregate total spend
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+")
)
age_group_spend_df =
df_with_age_group.groupBy("age_group").agg(F.sum("total_spend").alias("total_spend"))
print(" 4.Total spend by age group:")
age_group_spend_df.show()

# Step 4: Register as SQL view
df.createOrReplaceTempView("customers")

# SQL Query: Spend > 300
print(" 5.SQL Query: Customers with total spend > 300:")
sql_result = spark.sql("SELECT * FROM customers WHERE total_spend > 300")
sql_result.show()

# SQL Query: Total spend by gender
print(" 6.SQL Query: Total spend grouped by gender:")
sql_gender_spend = spark.sql("SELECT gender, SUM(total_spend) as total_spend FROM customers GROUP
BY gender")
sql_gender_spend.show()

# Step 5: Customer Segmentation - Loyalty Program
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")
)

print(" 7.Customer Loyalty Segments based on total spend:")
df_with_loyalty.show()

# BONUS: Filter customers with spend < 300
print(" 8.Customers with spend less than 300:")
low_spenders = df.filter(df.total_spend < 300)
low_spenders.show()
```

Output:

```
 1.Initial Customer 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|
+-----------+-------+---+------+-----------+----------+
```

```
 2.Customers older than 30 and total spend > 200:
+-----------+------+---+------+-----------+----------+
|customer_id|  name|age|gender|total_spend| join_date|
+-----------+------+---+------+-----------+----------+
|          2|   Bob| 35|  Male|      350.0|2019-03-15|
|          6|   Fay| 45|Female|      400.0|2019-12-05|
|          7|George| 50|  Male|      600.0|2018-09-18|
+-----------+------+---+------+-----------+----------+

 3.Average total spend by gender:
+------+---------+
|gender|avg_spend|
+------+---------+
|Female|    242.5|
|  Male|    400.0|
+------+---------+
```

```
 4.Total spend by age group:
+---------+-----------+
|age_group|total_spend|
+---------+-----------+
| Under 30|      950.0|
|    30-39|      470.0|
|    40-49|      550.0|
|      50+|      600.0|
+---------+-----------+

 5.SQL Query: Customers with total spend > 300:
+-----------+------+---+------+-----------+----------+
|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|
+-----------+------+---+------+-----------+----------+
```

```
 6.SQL Query: Total spend grouped by gender:
+------+-----------+
|gender|total_spend|
+------+-----------+
|Female|      970.0|
|  Male|     1600.0|
+------+-----------+

 7.Customer Loyalty Segments based on total spend:
+-----------+-------+---+------+-----------+----------+-------------+
|customer_id|   name|age|gender|total_spend| join_date|loyalty_level|
+-----------+-------+---+------+-----------+----------+-------------+
|          1|  Alice| 29|Female|      200.0|2020-05-01|       Silver|
|          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|         Gold|
|          7| George| 50|  Male|      600.0|2018-09-18|         Gold|
|          8| Hannah| 28|Female|      250.0|2022-02-20|       Silver|
+-----------+-------+---+------+-----------+----------+-------------+
```

```
 8.Customers with spend less than 300:
+-----------+-------+---+------+-----------+----------+
|customer_id|   name|age|gender|total_spend| join_date|
+-----------+-------+---+------+-----------+----------+
|          1|  Alice| 29|Female|      200.0|2020-05-01|
|          3|Charlie| 40|  Male|      150.0|2021-07-22|
|          5|    Eva| 32|Female|      120.0|2021-01-15|
|          8| Hannah| 28|Female|      250.0|2022-02-20|
+-----------+-------+---+------+-----------+----------+
```

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

Aim: To demonstrate how to use **Broadcast Variables** in Apache Spark to optimize the join operation between a large dataset (ratings) and a small dataset (movies), by replacing movie IDs with movie names.

Theory:
Broadcast variables in Spark are used to efficiently share a small, read-only dataset with all nodes in a cluster. This avoids the need to repeatedly send a copy of the data during tasks like joins.
In this practical, we:

- Created two DataFrames: one for movie ratings and one for movie names.
- Used Spark's broadcast feature to distribute the smaller movie dataset to all executors.
- Replaced movie IDs in the ratings DataFrame with their corresponding movie names using a **User Defined Function (UDF)** and a **broadcasted dictionary**.

Code:

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

# ✅ Step 1: Initialize Spark Session and Create Sample DataFrames
spark = SparkSession.builder.appName("BroadcastExample").getOrCreate()

# 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)

# 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)

print("🎥 Ratings Dataset:")
ratings_df.show()

print("🎬 Movies Dataset:")
movies_df.show()

# ✅ Step 2: Broadcast the Movies DataFrame
```

```
broadcast_movies_df = spark.sparkContext.broadcast(movies_df.collect())

# ✅ Step 3: Replace movie IDs with movie names using the broadcasted dataset
# Convert to dictionary for fast lookup
movie_dict = {row['movie_id']: row['movie_name'] for row in broadcast_movies_df.value}

# Define UDF to fetch movie names
def get_movie_name(movie_id):
    return movie_dict.get(movie_id, "Unknown")

get_movie_name_udf = udf(get_movie_name, StringType())

# Add movie_name column using UDF
ratings_with_movie_names_df = ratings_df.withColumn("movie_name", get_movie_name_udf(ratings_df.movie_id))

# ✅ Step 4: Show final output
print("✅ Ratings Dataset with Movie Names (Using Broadcast Variable):")
ratings_with_movie_names_df.show()
```

Output:

```
🎥 Ratings Dataset:
+-------+--------+------+
|user_id|movie_id|rating|
+-------+--------+------+
|      1|     101|   4.5|
|      2|     102|   3.0|
|      3|     103|   5.0|
|      4|     101|   4.0|
|      5|     104|   3.5|
+-------+--------+------+
```

```
🎬 Movies Dataset:
+--------+---------------+
|movie_id|     movie_name|
+--------+---------------+
|     101|     The Matrix|
|     102|      Inception|
|     103|The Dark Knight|
|     104|   Forrest Gump|
+--------+---------------+
```

```
✅ Ratings Dataset with Movie Names (Using Broadcast Variable):
+-------+--------+------+---------------+
|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

Aim: To implement a movie recommendation system using **Apache Spark's ALS (Alternating Least Squares)** algorithm and compute the **similarity between movies** using cosine similarity of latent features.

Theory:

A **Recommendation System** suggests items (like movies or products) to users based on their preferences or behavior. One popular collaborative filtering method is **ALS (Alternating Least Squares)**, especially suited for large-scale datasets and sparse user-item matrices.

In this practical:

- We create sample user ratings and movie metadata.
- The data is loaded into a **Spark DataFrame**.
- An **ALS model** is trained to learn latent features for both users and movies.
- We use these features to:
    - Predict ratings on unseen data and evaluate using **Root-Mean-Square Error (RMSE)**.
    - Generate **top-N movie recommendations** for a given user.
    - Compute **cosine similarity** between movies to find similar movies based on their latent feature vectors.

Code:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, udf
from pyspark.sql.types import FloatType
from pyspark.ml.recommendation import ALS
from pyspark.ml.evaluation import RegressionEvaluator
import pandas as pd
import numpy as np
import os

# Step 1: Create Spark session
spark = SparkSession.builder.appName("MovieRecommendationALS").getOrCreate()

# Step 2: Create and save ratings.csv and movies.csv
ratings_data = pd.DataFrame({
    "userId": [1, 1, 1, 2, 2, 3, 3, 3],
    "movieId": [1, 2, 3, 1, 4, 2, 3, 4],
    "rating": [4.0, 5.0, 3.0, 4.5, 2.0, 3.5, 4.0, 1.0]
})
movies_data = pd.DataFrame({
    "movieId": [1, 2, 3, 4],
    "title": ["The Matrix", "Inception", "Interstellar", "Avengers"]
})

# Save files to correct location
base_path = "/databricks/driver/"
ratings_path = os.path.join(base_path, "ratings.csv")
movies_path = os.path.join(base_path, "movies.csv")
ratings_data.to_csv(ratings_path, index=False)
movies_data.to_csv(movies_path, index=False)

# Step 3: Load CSVs into Spark
ratings_df = spark.read.option("header", "true").csv(f"file:{ratings_path}", inferSchema=True)
movies_df = spark.read.option("header", "true").csv(f"file:{movies_path}", inferSchema=True)
```

```
# Step 4: Prepare the data
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")
)

# Step 5: Train ALS model
(training_data, test_data) = ratings_df.randomSplit([0.8, 0.2], seed=42)

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

# Step 6: Evaluate model only if test_data is not empty
if test_data.count() > 0:
    predictions = model.transform(test_data)
    if predictions.count() > 0:
        evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating", predictionCol="prediction")
        rmse = evaluator.evaluate(predictions)
        print(f"Root-Mean-Square Error (RMSE): {rmse:.4f}")
    else:
        print(" ⚠️ Predictions are empty after filtering. Cannot compute RMSE.")
else:
    print(" ⚠️ Test data is empty. Skipping RMSE evaluation.")

# Step 7: Recommend top 5 movies for user 1
user_id = 1
user_subset = ratings_df.filter(ratings_df.userId == user_id).select("userId").distinct()
recommendations = model.recommendForUserSubset(user_subset, 5)
recommendations.show(truncate=False)

# Step 8: Compute movie similarity (Cosine) using ALS latent features
movie_factors = model.itemFactors.withColumnRenamed("id", "movieId")
target_movie_id = 1
target_features = movie_factors.filter(col("movieId") == target_movie_id).select("features").collect()[0][0]
broadcast_vector = spark.sparkContext.broadcast(np.array(target_features))

def cosine_similarity(v):
    v = np.array(v)
    target = broadcast_vector.value
    return float(np.dot(v, target) / (np.linalg.norm(v) * np.linalg.norm(target)))

cosine_similarity_udf = udf(cosine_similarity, FloatType())
similar_movies = movie_factors.withColumn("similarity", cosine_similarity_udf(col("features")))
similar_movies = similar_movies.orderBy("similarity", ascending=False).limit(6)

# Step 9: Join with titles and show results
similar_movies = similar_movies.join(movies_df, on="movieId", how="left")
```

```
similar_movies.select("movieId", "title", "similarity").show(truncate=False)
```

Output:

```
⚠️ Predictions are empty after filtering. Cannot compute RMSE.
+------+--------------------------------------------------+
|userId|recommendations                                   |
+------+--------------------------------------------------+
|1     |[{2, 4.8902454}, {1, 3.9954252}, {4, 1.5983232}]|
+------+--------------------------------------------------+


+-------+----------+----------+
|movieId|title     |similarity|
+-------+----------+----------+
|1      |The Matrix|1.0       |
|4      |Avengers  |0.98318845|
|2      |Inception |0.9482022 |
+-------+----------+----------+
```

Practical 9
Statistical operation on data frame

Aim: To perform basic statistical operations on a DataFrame using PySpark, including descriptive statistics, correlation, covariance, skewness, kurtosis, and statistical summaries.

Theory:
PySpark is a powerful tool for big data analytics, and it provides built-in functions for statistical analysis using the pyspark.sql module. Here's a brief overview of each operation:

- **Descriptive Statistics** (describe()): Computes count, mean, standard deviation, min, and max for numeric columns.
- **Correlation** (stat.corr()): Measures the linear relationship between two numeric variables using Pearson correlation.
- **Covariance** (stat.cov()): Indicates how two variables change together.
- **Skewness** (skewness()): Describes the asymmetry of the data distribution.
- **Kurtosis** (kurtosis()): Measures the "tailedness" or extremity of data points in the distribution.
- **Summary** (summary()): Provides detailed statistics like percentiles in addition to the above.

Code:

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F

# Step 1: Initialize Spark session
spark = SparkSession.builder.appName("StatisticalOperations").getOrCreate()

# Step 2: Create sample DataFrame
data = [
   ("Alice", 25, 10.5),
   ("Bob", 30, 12.5),
   ("Catherine", 35, 14.0),
   ("David", 40, 16.0),
   ("Eva", 45, 18.5)
]
columns = ["name", "age", "score"]
df = spark.createDataFrame(data, columns)
df.show()

# Step 3: Descriptive Statistics
print("Descriptive Statistics (All Columns):")
df.describe().show()

print("Descriptive Statistics (Age Column Only):")
df.select("age").describe().show()

# Step 4: Correlation
correlation = df.stat.corr("age", "score")
print(f"Pearson correlation between 'age' and 'score': {correlation}")

# Step 5: Covariance
covariance = df.stat.cov("age", "score")
print(f"Covariance between 'age' and 'score': {covariance}")

# Step 6: Skewness and Kurtosis
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}")

# Step 7: Statistical Summaries
print("Full Statistical Summary (All Columns):")
df.summary().show()

print("Statistical Summary for 'age' and 'score':")
df.select("age", "score").summary().show()
```

Output:

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

Descriptive Statistics (All Columns):
+-------+-----+-----------------+----------------+
|summary| name|              age|           score|
+-------+-----+-----------------+----------------+
|  count|    5|                5|               5|
|   mean| null|             35.0|            14.3|
| stddev| null|7.905694150420948|3.09434968935316|
|    min|Alice|               25|            10.5|
|    max|  Eva|               45|            18.5|
+-------+-----+-----------------+----------------+
```

```
Descriptive Statistics (Age Column Only):
+-------+-----------------+
|summary|              age|
+-------+-----------------+
|  count|                5|
|   mean|             35.0|
| stddev|7.905694150420948|
|    min|               25|
|    max|               45|
+-------+-----------------+
```

```
Pearson correlation between 'age' and 'score': 0.9964034540998122
Covariance between 'age' and 'score': 24.375
Skewness of 'score': 0.17235559601033562
Kurtosis of 'score': -1.164344293028107
Full Statistical Summary (All Columns):
+-------+-----+-----------------+----------------+
|summary| name|              age|           score|
+-------+-----+-----------------+----------------+
|  count|    5|                5|               5|
|   mean| null|             35.0|            14.3|
| stddev| null|7.905694150420948|3.09434968935316|
|    min|Alice|               25|            10.5|
|    25%| null|               30|            12.5|
|    50%| null|               35|            14.0|
|    75%| null|               40|            16.0|
|    max|  Eva|               45|            18.5|
+-------+-----+-----------------+----------------+
```

```
Statistical Summary for 'age' and 'score':
+-------+-----------------+----------------+
|summary|              age|           score|
+-------+-----------------+----------------+
|  count|                5|               5|
|   mean|             35.0|            14.3|
| stddev|7.905694150420948|3.09434968935316|
|    min|               25|            10.5|
|    25%|               30|            12.5|
|    50%|               35|            14.0|
|    75%|               40|            16.0|
|    max|               45|            18.5|
+-------+-----------------+----------------+
```

Practical 10
Using Spark ML to Produce Movie Recommendations

Aim: To build a movie recommendation system using the ALS (Alternating Least Squares) algorithm on the MovieLens dataset with PySpark in Databricks.

Theory:
A recommendation system is used to suggest items (like movies) to users based on their preferences. Collaborative Filtering is a popular method that relies on user-item interactions (ratings) rather than content.
ALS (Alternating Least Squares) is a matrix factorization technique used in collaborative filtering. It decomposes the large user-item rating matrix into two smaller matrices — one for users and one for items — to predict missing ratings.
In this practical:

- We simulate the MovieLens dataset using dummy data.
- Train an ALS model on the ratings.
- Evaluate it using RMSE (Root Mean Squared Error).
- Generate top-N movie recommendations for users.

Code:

```
from pyspark.sql import SparkSession
from pyspark.sql import Row
from pyspark.ml.recommendation import ALS
from pyspark.ml.evaluation import RegressionEvaluator

# Create Spark session
spark = SparkSession.builder.appName("ALSRecommendationSystem").getOrCreate()

# Step 1: Create dummy ratings and movies (20 movies, multiple users)
ratings_data = [
    Row(userId=1, movieId=1, rating=4.0), Row(userId=1, movieId=2, rating=3.0), Row(userId=1, movieId=3, rating=5.0),
    Row(userId=2, movieId=1, rating=5.0), Row(userId=2, movieId=4, rating=4.0), Row(userId=2, movieId=5, rating=3.0),
    Row(userId=3, movieId=2, rating=2.0), Row(userId=3, movieId=6, rating=4.5), Row(userId=3, movieId=7, rating=3.5),
    Row(userId=4, movieId=1, rating=4.5), Row(userId=4, movieId=8, rating=2.5), Row(userId=4, movieId=9, rating=3.0),
    Row(userId=5, movieId=10, rating=5.0), Row(userId=5, movieId=3, rating=4.0), Row(userId=5, movieId=11, rating=3.0),
    Row(userId=6, movieId=12, rating=4.0), Row(userId=6, movieId=13, rating=2.5), Row(userId=6, movieId=4, rating=4.5),
    Row(userId=7, movieId=14, rating=3.0), Row(userId=7, movieId=15, rating=3.5), Row(userId=7, movieId=16, rating=4.0),
    Row(userId=8, movieId=5, rating=2.0), Row(userId=8, movieId=6, rating=3.0), Row(userId=8, movieId=17, rating=4.5),
    Row(userId=9, movieId=18, rating=5.0), Row(userId=9, movieId=19, rating=4.0), Row(userId=9, movieId=20, rating=3.5),
    Row(userId=10, movieId=1, rating=3.5), Row(userId=10, movieId=10, rating=2.0), Row(userId=10, movieId=11, rating=4.5)
]

movies_data = [
   Row(movieId=1, title="Toy Story (1995)"), Row(movieId=2, title="Jumanji (1995)"),
   Row(movieId=3, title="Grumpier Old Men (1995)"), Row(movieId=4, title="Waiting to Exhale (1995)"),
```

```
    Row(movieId=5, title="Father of the Bride Part II (1995)"), Row(movieId=6, title="Heat (1995)"),
    Row(movieId=7, title="Sabrina (1995)"), Row(movieId=8, title="Tom and Huck (1995)"),
    Row(movieId=9, title="Sudden Death (1995)"), Row(movieId=10, title="GoldenEye (1995)"),
   Row(movieId=11, title="American President (1995)"), Row(movieId=12, title="Dracula: Dead and Loving It (1995)"),
    Row(movieId=13, title="Balto (1995)"), Row(movieId=14, title="Nixon (1995)"),
    Row(movieId=15, title="Cutthroat Island (1995)"), Row(movieId=16, title="Casino (1995)"),
    Row(movieId=17, title="Sense and Sensibility (1995)"), Row(movieId=18, title="Four Rooms (1995)"),
   Row(movieId=19, title="Ace Ventura: When Nature Calls (1995)"), Row(movieId=20, title="Money Train (1995)")
]

# Create DataFrames
ratings_df = spark.createDataFrame(ratings_data)
movies_df = spark.createDataFrame(movies_data)

# Show sample data
ratings_df.show(5)
movies_df.show(5)

# Step 2: Split data into training and test sets
(training_data, test_data) = ratings_df.randomSplit([0.8, 0.2])

# Step 3: Train ALS model
als = ALS(userCol="userId", itemCol="movieId", ratingCol="rating", coldStartStrategy="drop", nonnegative=True)
model = als.fit(training_data)

# Step 4: Make predictions
predictions = model.transform(test_data)
predictions.show()

# Step 5: Evaluate model using RMSE
evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating", predictionCol="prediction")
rmse = evaluator.evaluate(predictions)
print(f"Root Mean Squared Error (RMSE) = {rmse}")

# Step 6.1: Recommend top 5 movies for a specific user
user_id = 1
user_recommendations = model.recommendForUserSubset(ratings_df.filter(ratings_df.userId == user_id), numItems=5)
user_recommendations.show(truncate=False)

# Step 6.2: Recommend top 5 movies for all users
all_users_recommendations = model.recommendForAllUsers(5)
all_users_recommendations.show(truncate=False)

# Step 6.3: View learned item factors (movie features)
movie_factors = model.itemFactors
movie_factors.show(5)
```

Output:

```
+------+-------+------+
|userId|movieId|rating|
+------+-------+------+
|     1|      1|   4.0|
|     1|      2|   3.0|
|     1|      3|   5.0|
|     2|      1|   5.0|
|     2|      4|   4.0|
+------+-------+------+
only showing top 5 rows

+-------+--------------------+
|movieId|               title|
+-------+--------------------+
|      1|    Toy Story (1995)|
|      2|      Jumanji (1995)|
|      3|Grumpier Old Men ...|
|      4|Waiting to Exhale...|
|      5|Father of the Bri...|
+-------+--------------------+
only showing top 5 rows
```

```
Root Mean Squared Error (RMSE) = 3.0722958695577134
+------+--------------------------------------------------------------------------------------------+
|userId|recommendations                                                                             |
+------+--------------------------------------------------------------------------------------------+
|1     |[{3, 4.8626146}, {10, 4.5063205}, {1, 4.00315}, {16, 3.0748944}, {9, 2.6801414}]|
+------+--------------------------------------------------------------------------------------------+

+------+--------------------------------------------------------------------------------------------+
|userId|recommendations                                                                             |
+------+--------------------------------------------------------------------------------------------+
|10    |[{11, 4.291738}, {1, 3.496703}, {9, 2.3609731}, {18, 2.35038}, {10, 2.0577507}]             |
|1     |[{3, 4.8626146}, {10, 4.5063205}, {1, 4.00315}, {16, 3.0748944}, {9, 2.6801414}]            |
|2     |[{1, 4.9129887}, {11, 3.9102585}, {3, 3.7483354}, {17, 3.6319885}, {10, 3.5039907}]|
|3     |[{7, 3.4131098}, {16, 2.118319}, {2, 1.9503483}, {18, 1.5889444}, {14, 1.5887395}] |
|4     |[{1, 4.364947}, {11, 3.6267962}, {3, 3.3537855}, {10, 3.151288}, {9, 2.9711323}]            |
|5     |[{10, 4.7634544}, {3, 4.0536146}, {1, 3.3902302}, {11, 3.0141444}, {9, 2.2164576}] |
|6     |[{4, 4.41954}, {17, 4.0051274}, {12, 3.9284797}, {6, 2.670085}, {13, 2.4553}]               |
|7     |[{16, 3.9294887}, {14, 2.9471166}, {3, 2.5067253}, {7, 2.0247893}, {1, 1.876555}]           |
|8     |[{17, 4.4058666}, {4, 3.2685297}, {6, 2.9372444}, {12, 2.9053595}, {1, 2.292549}]           |
|9     |[{18, 4.9387875}, {20, 3.4571514}, {1, 2.2616498}, {11, 1.7058867}, {9, 1.606958}] |
+------+--------------------------------------------------------------------------------------------+
```

```
+---+--------------------+
| id|            features|
+---+--------------------+
| 10|[0.0, 4.295519E-4...|
| 20|[1.003936, 0.0, 0...|
|  1|[0.0, 0.60203534,...|
| 11|[0.0, 0.011662140...|
|  2|[0.6106738, 0.0, ...|
+---+--------------------+
only showing top 5 rows
```