Practical No: 01

Name: Hammad Ansari

Roll No: 15

Aim: Installation of Apache Spark

This practical demonstrates the **step-by-step installation and configuration of Apache Spark** on a local system, preparing the environment for running both **Scala** and **PySpark** programs.

Step 1: Install Java (JDK)

1. Visit https://www.oracle.com/java/technologies/javase-downloads.html 2. Download and install **JDK 8 or JDK 11**, as Spark requires Java to run.

3. Set environment variables:

JAVA_HOME = C:\Program Files\Java\jdk-<version> Add %JAVA_HOME%\bin to the **PATH** variable.

Step 2: Install Python (for PySpark Users)

1. Download **Python** from https://www.python.org
2. During installation, check **"Add to PATH"**.
3. Verify installation by running: python --version

Step 3: Download Apache Spark

1. Go to https://spark.apache.org/downloads.html 2. Select a version (e.g., **Spark 3.5.0 with Hadoop 3.3**).

3. Extract the downloaded ZIP file to a folder, C:\spark.

Step 4: Set Environment Variables

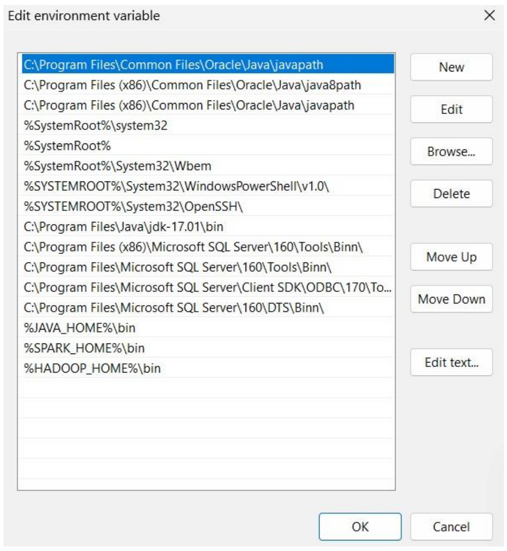
● Add the following:

○ SPARK_HOME = C:\spark

○ HADOOP_HOME = C:\hadoop *(Windows only)* ● Add to **PATH**:

○ %SPARK_HOME%\bin

○ %HADOOP_HOME%\bin


Edit environment variable

```
C:\Program Files\Common Files\Oracle\Java\javapath
C:\Program Files (x86)\Common Files\Oracle\Java\java8path
C:\Program Files (x86)\Common Files\Oracle\Java\javapath
%SystemRoot%\system32
%SystemRoot%
%SystemRoot%\System32\Wbem
%SYSTEMROOT%\System32\WindowsPowerShell\v1.0\
%SYSTEMROOT%\System32\OpenSSH\
C:\Program Files\Java\jdk-17.01\bin
C:\Program Files (x86)\Microsoft SQL Server\160\Tools\Binn\
C:\Program Files\Microsoft SQL Server\160\Tools\Binn\
C:\Program Files\Microsoft SQL Server\Client SDK\ODBC\170\To...
C:\Program Files\Microsoft SQL Server\160\DTS\Binn\
%JAVA_HOME%\bin
%SPARK_HOME%\bin
%HADOOP_HOME%\bin
```

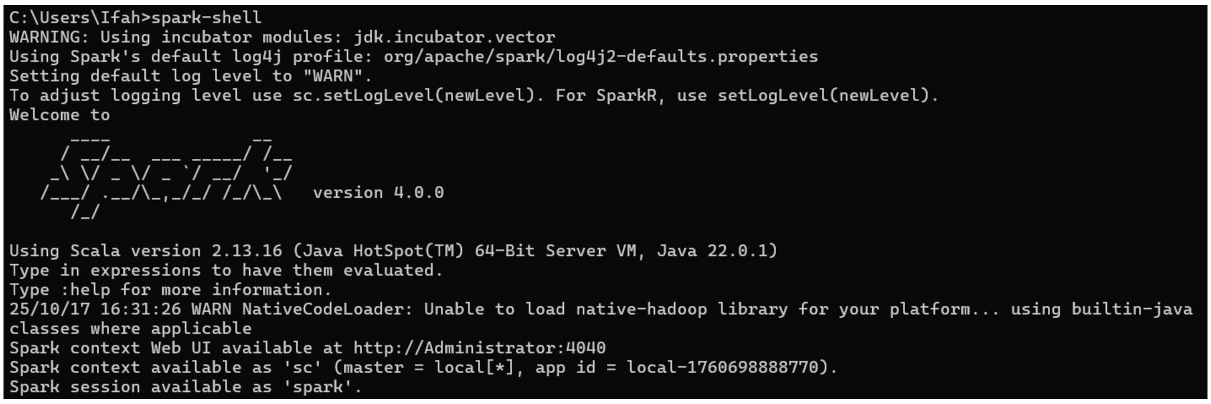
Step 5: Install winutils.exe (Windows Only)

1. Download from https://github.com/cdarlint/winutils
2. Create a folder: C:\hadoop\bin.
3. Place the downloaded **winutils.exe** file inside it.

Step 6: Test Spark Installation

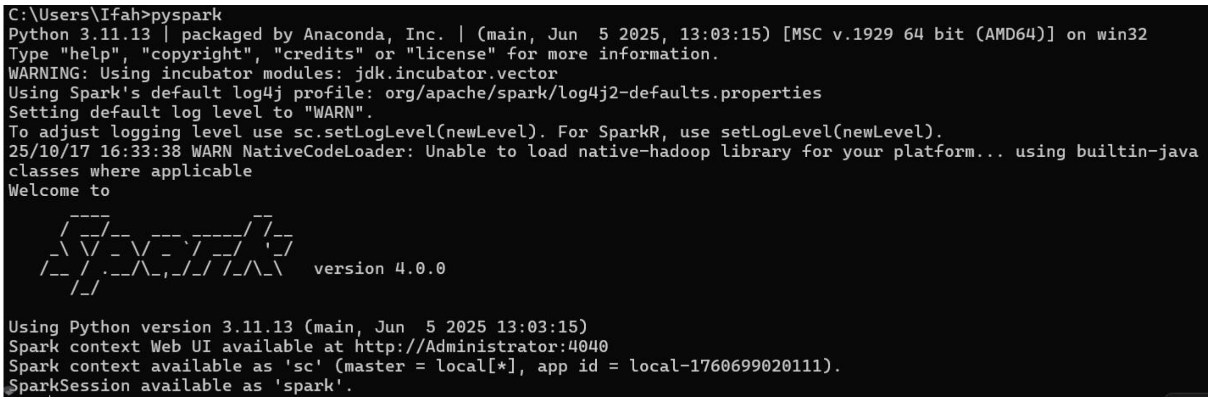
For Scala Shell:

spark-shell



```
C:\Users\Ifah>spark-shell
WARNING: Using incubator modules: jdk.incubator.vector
Using Spark's default log4j profile: org/apache/spark/log4j2-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 4.0.0
      /_/

Using Scala version 2.13.16 (Java HotSpot(TM) 64-Bit Server VM, Java 22.0.1)
Type in expressions to have them evaluated.
Type :help for more information.
25/10/17 16:31:26 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java
classes where applicable
Spark context Web UI available at http://Administrator:4040
Spark context available as 'sc' (master = local[*], app id = local-1760698888770).
Spark session available as 'spark'.
```

For PySpark: pyspark



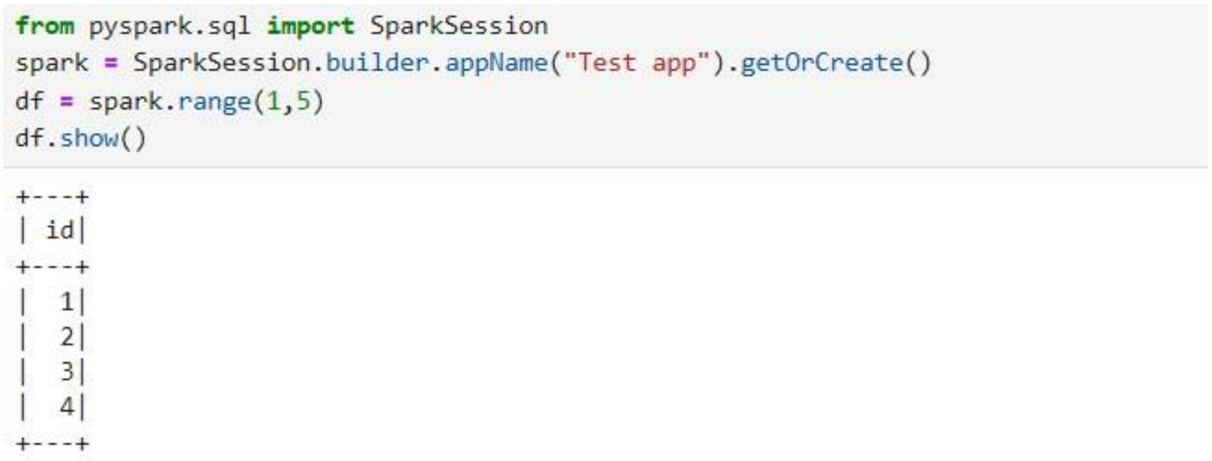
```
C:\Users\Ifah>pyspark
Python 3.11.13 | packaged by Anaconda, Inc. | (main, Jun  5 2025, 13:03:15) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
WARNING: Using incubator modules: jdk.incubator.vector
Using Spark's default log4j profile: org/apache/spark/log4j2-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
25/10/17 16:33:38 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java
classes where applicable
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 4.0.0
      /_/

Using Python version 3.11.13 (main, Jun  5 2025 13:03:15)
Spark context Web UI available at http://Administrator:4040
Spark context available as 'sc' (master = local[*], app id = local-1760699020111).
SparkSession available as 'spark'.
```

You should see:

Spark context available as 'sc'

Step 7: Test with a Simple PySpark Code from pyspark.sql import SparkSession spark = SparkSession.builder.appName("TestApp").getOrCreate() df = spark.range(1, 100) df.show()

This verifies that **SparkSession** and **PySpark environment** are working correctly.



```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Test app").getOrCreate()
df = spark.range(1,5)
df.show()
```

```
+---+
| id|
+---+
|  1|
|  2|
|  3|
|  4|
+---+
```

Practical No: 02

Name: Hammad Ansari

Roll No: 15

Aim: Demonstrating Spark Basics and RDD Interface

Date: 20-09-2025

Theory:

1. This code demonstrates basic Apache Spark operations using the RDD (Resilient Distributed Dataset) interface.
2. It begins by creating a SparkContext in local mode, which is essential to initialize any Spark application.
3. A Python list of numbers is converted into an RDD using the parallelize() method, enabling distributed data processing.
4. A transformation (map) is applied to multiply each element by 2, and an action (collect) retrieves the transformed data to the driver program.

Code:

```
from pyspark import SparkContext

# 1. Start Spark Context
sc = SparkContext("local", "Practical2_RDD")

# 2. Create an RDD from a Python collection
numbers = [10, 20, 30, 40, 50] rdd =
sc.parallelize(numbers)

# 3. Apply a Transformation: multiply each element by 2 rdd_transformed
= rdd.map(lambda x: x * 2)

# 4. Apply an Action: collect results back to driver result
= rdd_transformed.collect()

print("Original RDD:", numbers)
print("Transformed RDD (each * 2):", result)

# 5. Another Action: count the elements count
= rdd.count()
print("Number of elements in original RDD:", count)

# Stop Spark Context after use sc.stop()
```

OUTPUT:

```
Original RDD: [10, 20, 30, 40, 50]
Transformed RDD (each * 2): [20, 40, 60, 80, 100]
Number of elements in original RDD: 5
```

Practical No:3

Name: Hammad Ansari

Roll No: 15

Aim: Demonstrating Filtering RDDs,and the Minimum Tempertaure by Location Example

Date: 20-09-2025

Theory:

1. In this practical, we analyzed weather data from Indian cities using PySpark and the RDD API.
2. A CSV file with temperature records was loaded, the header was filtered out, and lines were parsed to extract city name, measurement type (TMIN or TMAX), and temperature values.
3. Temperatures were converted from tenths of degrees to Celsius for meaningful interpretation.
4. Using RDD transformations and actions, the data was filtered for minimum and maximum temperatures, grouped by city, and reduced to find the lowest and highest temperatures per city.

Code:

```
from pyspark import SparkConf, SparkContext

# Spark setup
conf = SparkConf().setMaster("local").setAppName("IndiaTemperatures") sc
= SparkContext(conf=conf)

# Load data
lines = sc.textFile(r"C:\Users\Administrator\Downloads\india_weather.csv")
```

```
header = lines.first()  # get header data =
lines.filter(lambda line: line != header)


# Parse each line def
parseLine(line):
    fields = line.split(',')     stationID = fields[0]     city
= fields[1]     measurementType = fields[3]  # TMIN or TMAX
temperature = float(fields[4]) * 0.1  # convert to Celsius
return (stationID, city, measurementType, temperature)


parsedLines = data.map(parseLine)

# Filter only TMIN and TMAX
temps = parsedLines.filter(lambda x: x[2] in ("TMIN", "TMAX"))
# Map as (city, (min_temp, max_temp)) tuples
cityTemps = temps.map(lambda x: (     x[1],
    (x[3] if x[2] == "TMIN" else float('inf'),
x[3] if x[2] == "TMAX" else float('-inf'))
))

# Reduce by city to get min and max
combinedTemps = cityTemps.reduceByKey(lambda a, b: (min(a[0], b[0]), max(a[1],
# Collect and print results results
= combinedTemps.collect()
 for city, (minTemp, maxTemp) in results:
    print(f"{city}\tMin: {minTemp:.2f}C\tMax: {maxTemp:.2f}C")
```

```
Mumbai Min: 25.00C      Max: 34.50C
Delhi  Min: 24.00C      Max: 34.00C
Chennai Min: 26.00C     Max: 33.00C
Kolkata Min: 24.50C     Max: 33.50C
```

Practical No:4

Name: Hammad Ansari

Roll No: 15

Aim: Counting word Occurence using flatmap

Date: 20-9-25

Theory:

In this practical, we performed a word count using PySpark's RDD API based on dynamic user input.

The program accepts multiple lines of text from the user until the word "STOP" is entered.

The input lines are parallelized into an RDD, and each line is split into words using the flatMap() transformation.

Words are converted to lowercase and mapped to key-value pairs (word, 1), which are then aggregated using reduceByKey() to count occurrences.

```
from pyspark.sql import SparkSession
# Step 1: Start Spark Session spark
= SparkSession.builder \
    .appName("WordCountRDD_UserInput") \
.getOrCreate()

# Step 2: Take input from the user
print("Enter multiple lines of text. Type 'STOP' to finish input:")
user_lines = [] while True:
    line = input()     if
line.strip().upper() == "STOP":
        break
    user_lines.append(line)

# Step 3: Convert input list to RDD
rdd = spark.sparkContext.parallelize(user_lines)

# Step 4: Word Count Logic
```

```
words = rdd.flatMap(lambda line: line.split(" ")) word_pairs
= words.map(lambda word: (word.lower(), 1)) word_counts =
word_pairs.reduceByKey(lambda a, b: a + b)

# Step 5: Display Results print("\n📊
Word Count Result:") for word, count in
word_counts.collect():
    print(f"{word}: {count}")
```

The final word counts are collected and displayed, demonstrating PySpark's ability to process user input and perform basic text analytics with RDDs.

```
Enter multiple lines of text. Type 'STOP' to finish input:

📊 Word Count Result:

practical: 1

data: 1 msc:

1 part: 1

science: 1

1: 1

spark: 1
```

Practical No:5

Name: Hammad Ansari

Roll No: 19

Aim:Executing SQL commands and SQL-style functions on a DataFrame

Date: 4-10-25

Theory:

In this practical, we used PySpark's DataFrame API to manipulate and analyze employee data through various transformations and queries.

A Spark session was started, and a DataFrame was created with sample data including employee ID, name, age, and salary.

We performed operations such as column selection, row filtering (e.g., employees older than 30), average salary calculation, and sorting by salary in descending order.

A new column called bonus was added by computing 10% of each employee’s salary.

```
from pyspark.sql import SparkSession from
pyspark.sql.functions import col

# Initialize Spark
spark = SparkSession.builder \
    .appName("PySparkBasicExample") \
    .master("local[*]") \
.getOrCreate()

# Sample Data
data = [
    (1, "Alice", 25, 50000) ,
    (2, "Bob", 30, 60000) ,
    (3, "Charlie", 35, 70000) ,
    (4, "David", 40, 80000)
]
columns = ["id", "name", "age", "salary"]

# Create DataFrame
df = spark.createDataFrame(data, columns)
print("=== Original DataFrame ===") df.show()

# Select columns
print("=== Select name and salary ===") df.select("name",
"salary").show()

# Filter rows (age > 30)
```

The DataFrame was registered as a temporary SQL view, allowing SQL queries to filter employees with a salary greater than 60,000, showcasing the integration of SQL with PySpark.

```
print("=== Filter: age > 30 ===") df.filter(col("age") >
30).show()

# Aggregation: average salary print("===
Average Salary ===")
df.groupBy().avg("salary").show()

# Sorting by salary descending
print("=== Sort by salary descending ===") df.orderBy(col("salary").desc()).show()

# Add new column (bonus = 10% of salary) df =
df.withColumn("bonus", col("salary") * 0.1) print("===
Add Bonus Column ===") df.show()

# Using SQL
df.createOrReplaceTempView("employees") print("=== SQL Query: salary > 60000 ===")
sql_result = spark.sql("SELECT name, salary, bonus FROM employees WHERE salary >
sql_result.show()

# Stop Spark spark.stop()
```

```
===  Original DataFrame  ===
+---+-------+---+------+
| id|   name|age|salary|
+---+-------+---+------+
|  1|  Alice| 25| 50000|
|  2|    Bob| 30| 60000|
|  3|Charlie| 35| 70000|
|  4|  David| 40| 80000|
+---+-------+---+------+

===  Select name and salary  ===
+-------+------+
|   name|salary|
+-------+------+
|  Alice| 50000|
|    Bob| 60000|
|Charlie| 70000|
|  David| 80000|
+-------+------+

===  Filter: age >  30 ===
+---+-------+---+------+
| id|   name|age|salary|
+---+-------+---+------+
|  3|Charlie| 35| 70000|
|  4|  David| 40| 80000|
+---+-------+---+------+

===  Average Salary  ===
+-----------+
|avg(salary)|
+-----------+
|    65000.0|
+-----------+

===  Sort by salary descending  ===
+---+-------+---+------+
| id|   name|age|salary|
+---+-------+---+------+
|  4|  David| 40| 80000|
|  3|Charlie| 35| 70000|
|  2|    Bob| 30| 60000|
|  1|  Alice| 25| 50000|
+---+-------+---+------+

===  Add Bonus Column  ===
+---+-------+---+-----+------+
| id|   name|age|salary| bonus|
+---+-------+---+-----+------+
|  1|  Alice| 25| 50000|5000.0|
|  2|    Bob| 30| 60000|6000.0|
|  3|Charlie| 35| 70000|7000.0|
|  4|  David| 40| 80000|8000.0|
+---+-------+---+-----+------+

===  SQL Query: salary  > 60000 ===
+-------+-----+------+
|   name|salary| bonus|
+-------+-----+------+
|  David| 80000|8000.0|
```

3

```
|Charlie| 70000|7000.0|
```

Practical No:6

Name: Hammad Ansari

Roll No: 15

Aim: Implement Total spent by customer with DataFrames

Date: 4-10-25

Theory:

In this practical, we used PySpark to calculate the total amount spent by each customer from a dataset containing purchase records.

A Spark DataFrame was created with sample data including customer_id, item_id, and amount, and the data was displayed using the .show() function.

The records were grouped by customer_id, and the sum() aggregation function was applied to the amount column to calculate total spending.

The resulting aggregated column was renamed to total_spent using the .alias() function for better readability.

This practical demonstrates fundamental PySpark operations like DataFrame creation, grouping, and aggregation, which are essential for analyzing customer spending in large datasets.

```
In [1]
```

```
from pyspark.sql import SparkSession from
pyspark.sql.functions import sum

#Start Spark
spark = SparkSession.builder.appName("TotalSpentByCustomer").getOrCreate()
```

6 1

0 6

```
data = [
    (1, 101, 500) ,
    (2, 102, 3000) ,
    (1, 103, 700) ,
    (2, 104, 10000) ,
    (3, 105, 2000)
]
columns = ["customer_id", "item_id", "amount"] df

= spark.createDataFrame(data, columns)


print("=== Original Purchases ===") df.show()

#Group by customer and calculate total spent
total_spent = df.groupBy("customer_id").agg(sum("amount").alias("total_spent"))

print("=== Total Spent by Each Customer ===") total_spent.show()
#Stop Spark
spark.stop()
```

```
===  Original Purchases  ===
+-----------+-------+------+
|customer_id|item_id|amount|
+-----------+-------+------+
|          1|    101|   500|
|          2|    102|  3000|
|          1|    103|   700|
|          2|    104| 10000|
|          3|    105|  2000|
+-----------+-------+------+

===  Total Spent by Each Customer  ===
+-----------+-----------+
|customer_id|total_spent|
+-----------+-----------+
|          1|       1200|
|          2|      13000|
|          3|       2000|
```

Practical No:7

Name: Hammad Ansari

Roll No: 15

Aim: Use Broadcast variables to display movie name instead of ID numbers in spark Date: 8-10-2025

Theory:

Two DataFrames were created — one for movie information (MovieID, MovieName) and another for user ratings (UserID, MovieID, Rating).

The movie DataFrame, being small in size, was broadcasted using PySpark's broadcast() function to optimize the join process.

A left join was performed on the MovieID column to combine user ratings with their corresponding movie names.

Broadcasting reduces data shuffling across the cluster, leading to better performance in distributed systems.

Broadcast joins are ideal when one of the datasets is small enough to fit into memory, making the join operation more efficient.\

```
from pyspark.sql import SparkSession from pyspark.sql.functions import broadcast
spark = SparkSession.builder.appName("BroadcastExample_Bollywood").getOrCreate()


# Movie dictionary as list of tuples
movie_data = [     (1, "Inception")
,
    (2, "Interstellar") ,
    (3, "The Dark Knight") ,
    (4, "Tenet") ,
    (5, "3 Idiots") ,
    (6, "Lagaan") ,
    (7, "Dangal") ,
    (8, "Gully Boy") ,
    (9, "Chhichhore") ,
    (10, "Bahubali: The Beginning") ,
    (11, "Bahubali: The Conclusion") ,
    (12, "KGF: Chapter 1") ,
    (13, "KGF: Chapter 2") ,
    (14, "RRR") ,
    (15, "Pathaan")
]

# Create movie DataFrame
movies_df = spark.createDataFrame(movie_data, ["MovieID", "MovieName"])

# Create sample ratings DataFrame
```

```
data = [
(101, 1, 5) ,
(102, 3, 4) ,
(103, 2, 5) ,
(104, 5, 4) ,
(105, 6, 5) ,
(106, 7, 5) ,
(107, 9, 4) ,
(108, 11, 5) ,
(109, 13, 5) ,
(110, 15, 4)
]
columns = ["UserID", "MovieID", "Rating"] ratings_df =
spark.createDataFrame(data, columns) ratings_df

# Broadcast movies and join
joined_df = ratings_df.join(broadcast(movies_df), on="MovieID", how="left")
joined_df.show(truncate=False) spark.stop()
```

```
+-------+------+------+------------------------+
|MovieID|UserID|Rating|MovieName               |
+-------+------+------+------------------------+
|1      |101   |5     |Inception               |
|3      |102   |4     |The Dark Knight         |
|2      |103   |5     |Interstellar            |
|5      |104   |4     |3 Idiots                |
|6      |105   |5     |Lagaan                  |
|7      |106   |5     |Dangal                  |
|9       |107   |4     |Chhichhore                |
|11      |108   |5     |Bahubali: The Conclusion|
|13      |109   |5     |KGF: Chapter 2            |
|15      |110   |4     |Pathaan                   |
```

Practical No:8

Name: Zyan Ahmed Qureshi

Roll No: 19

Aim:Create Similar Movies from One Million Rating

Date:11-10-2025

Theory:

In this practical, we aimed to find similar movies based on user ratings using the MovieLens 100k dataset and PySpark for distributed processing.

The user ratings and movie metadata were loaded and preprocessed. Ratings were grouped by users, and for each user, all unique movie pairs were generated along with intermediate statistics needed for cosine similarity.

A user-defined function (UDF) was used to compute rating-based values like the product of ratings, squared ratings, and a count of co-ratings for each movie pair.

The cosine similarity between movie pairs was calculated using the dot product and magnitude of rating vectors, and only those with at least 20 common raters were retained for meaningful results.

```
from pyspark.sql import SparkSession from
pyspark.sql import functions as F from
itertools import combinations
from pyspark.sql.types import ArrayType, StructType, StructField, IntegerType, F

# 1. Start Spark Session spark
= SparkSession.builder \
    .appName("SimilarMovies100k") \
    .config("spark.ui.showConsoleProgress", "true") \
.getOrCreate()

# 2. Define file paths (change if needed)
ratings_path = r"C:\Users\Administrator\Downloads\ml-100k\ml-100k\u.data"
movies_path = r"C:\Users\Administrator\Downloads\ml-100k\ml-100k\u.item"
```

```
# 3. Load ratings data (tab-separated) ratings =
spark.read.text(ratings_path).select(
    F.split(F.col("value"), "\t").alias("fields")
).select(
    F.col("fields")[0].cast("int").alias("userId") ,
    F.col("fields")[1].cast("int").alias("movieId") ,
F.col("fields")[2].cast("float").alias("rating") )
```

Finally, based on a movie title entered by the user, the program retrieved the top 10 most similar movies by joining with the movie names and sorting by similarity score.

```
In [1]:
```

```
# 4. Load movie data (pipe-separated)
movies = spark.read.text(movies_path).select(
    F.split(F.col("value"), "\\|").alias("fields")
).select(
    F.col("fields")[0].cast("int").alias("movieId") ,
    F.col("fields")[1].alias("title")
)

# 5. Group ratings by user
user_movies = ratings.groupBy("userId") \
    .agg(F.collect_list(F.struct("movieId", "rating")).alias("movies"))

# 6. Create UDF to generate movie pairs def
make_pairs(movie_list):
    result = []     for (m1, r1), (m2, r2) in combinations([(m.movieId, m.rating)
for m in movie         result.append((m1, m2, r1 * r2, r1 ** 2, r2 ** 2, 1))
return result

# UDF schema
pair_schema = ArrayType(StructType([
StructField("m1", IntegerType()) ,
    StructField("m2", IntegerType()) ,
    StructField("prod", FloatType()) ,
    StructField("p1", FloatType()) ,
    StructField("p2", FloatType()) ,
    StructField("cnt", IntegerType())
]))

# Register UDF
make_pairs_udf = F.udf(make_pairs, pair_schema)

# 7. Explode movie pairs
pairs  =  user_movies.select(F.explode(make_pairs_udf("movies")).alias("pair"))  \
.select(
        F.col("pair.m1").alias("movie1") ,
F.col("pair.m2").alias("movie2") ,
        F.col("pair.prod").alias("prod") ,
        F.col("pair.p1").alias("p1") ,
        F.col("pair.p2").alias("p2") ,
        F.col("pair.cnt").alias("cnt")
)

# 8. Aggregate by movie pairs
pair_stats = pairs.groupBy("movie1", "movie2").agg(
    F.sum("prod").alias("dotProduct") ,
    F.sum("p1").alias("sumSq1") ,
    F.sum("p2").alias("sumSq2") ,
    F.sum("cnt").alias("coCount")
)

# 9. Compute cosine similarity and filter similarities
= pair_stats.withColumn(
    "similarity",
    F.col("dotProduct") / (F.sqrt(F.col("sumSq1")) * F.sqrt(F.col("sumSq2")))
).filter(F.col("coCount") >= 20)

# 10. Ask user for the target movie title
target_title = input("Enter the movie name to find similar movies: ").strip()
target_movie = movies.filter(F.col("title").like(f"%{target_title}%")).first()

if target_movie:
    target_id = target_movie.movieId     top_similar =
similarities.filter(F.col("movie1") == target_id) \
        .join(movies, similarities.movie2 == movies.movieId) \
        .select("title", "similarity", "coCount") \
        .orderBy(F.col("similarity").desc()) \
        .limit(10)

    print(f"\n□ Top 10 movies similar to '{target_movie.title}':\n")
for row in top_similar.collect():
        print(f"{row['title']}  (similarity={row['similarity']:.3f},  co-raters={r
else:     print("❌Movie not found!")

# 11. Stop Spark session spark.stop()
```

```
□ Top 10 movies similar to 'Titanic (1997)':

Silence of the Lambs, The (1991) (similarity=0.980, co-raters=70)
Wrong Trousers, The (1993) (similarity=0.979, co-raters=23)
Philadelphia (1993) (similarity=0.979, co-raters=31)
Maverick (1994) (similarity=0.979, co-raters=24)
Abyss, The (1989) (similarity=0.977, co-raters=32)
Sling Blade (1996) (similarity=0.977, co-raters=35)
Cinderella (1950) (similarity=0.977, co-raters=24)
Anastasia (1997) (similarity=0.977, co-raters=25)
Schindler's List (1993) (similarity=0.976, co-raters=53)
Primal Fear (1996) (similarity=0.976, co-raters=45)
```