University of Mumbai

DEPARTMENT OF COMPUTER SCIENCE

मुंबई विद्यापीठ
University of Mumbai
Re-accredited with A++ Grade
(CGPA 3.65) by NAAC (3rd Cycle 2021)

M.Sc. Data Science – Semester I

SPARK TECHNOLOGIES

JOURNAL

2024-2025

SUBMITTED BY

Sunil Kumar Sabhajit Yadav

SEAT NO: 1300094

मुंबई विद्यापीठ
University of Mumbai
Re-accredited with A++ Grade
(CGPA 3.65) by NAAC (3rd Cycle 2021)

University of Mumbai

DEPARTMENT OF COMPUTER SCIENCE

CERTIFICATE

This is to **certify** that the work entered in this journal was done in the **University Department of Computer Science, University of Mumbai** laboratory by Mr./Ms. **Sunil Kumar Sabhajit Yadav** Seat No. **1300094** for the course of **M.Sc. (Data Science) (NEP) Semester - I** during the academic year **2024-25** in a satisfactory manner.

Subject In-charge

Head of Department

External Examiner

Date:___________

INDEX

Practical No.1

Aim: Installation of Apache Spark

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

If you already have Java 8 and Python 3 installed, you can skip the first two steps.

Step 1: Install Java 8

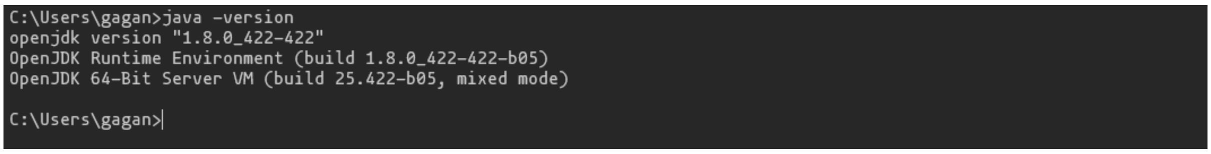
Apache Spark requires Java 8. You can check to see if Java is installed using the command prompt.

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

Type the following command

```
java -version
```

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



```
C:\Users\gagan>java -version
openjdk version "1.8.0_422-422"
OpenJDK Runtime Environment (build 1.8.0_422-422-b05)
OpenJDK 64-Bit Server VM (build 25.422-b05, mixed mode)

C:\Users\gagan>
```

Your version may be different. The second digit is the Java version – in this case, Java 8. If you don't have Java installed:

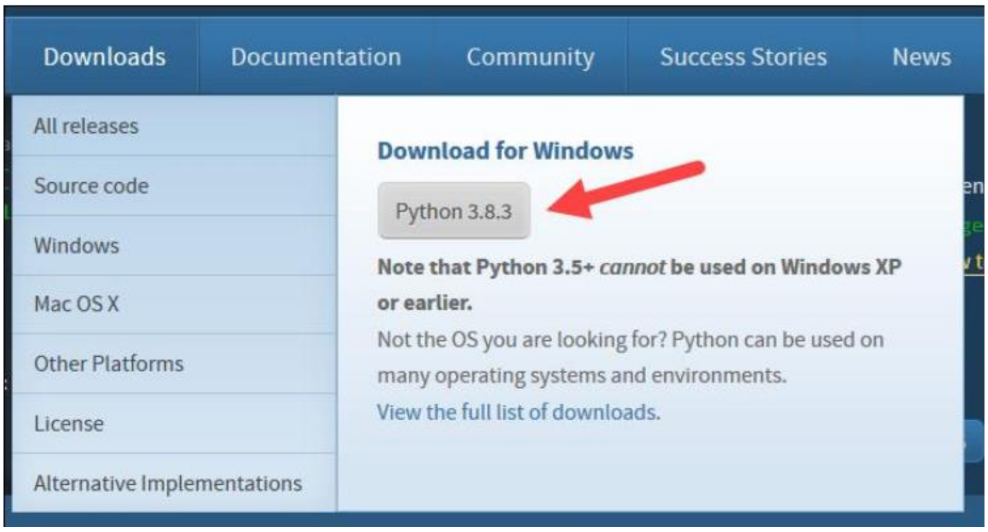
1. Open a browser window, and navigate to https://java.com/en/download/
2. Click the Java Download button and save the file to a location of your choice.
3. 3. Once the download finishes double-click the file to install Java.
4. Note: At the time this article was written, the latest Java version is 1.8.0_251. Installing a
5. later version will still work. This process only needs the Java Runtime Environment (JRE) –
6. the full Development Kit (JDK) is not required. The download link to JDK
7. is https://www.oracle.com/java/technologies/javase-downloads.html.

Step 2: Install Python

1. To install the Python package manager, navigate to https://www.python.org/ in your web browser.

2. Mouse over the Download menu option and click Python 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.



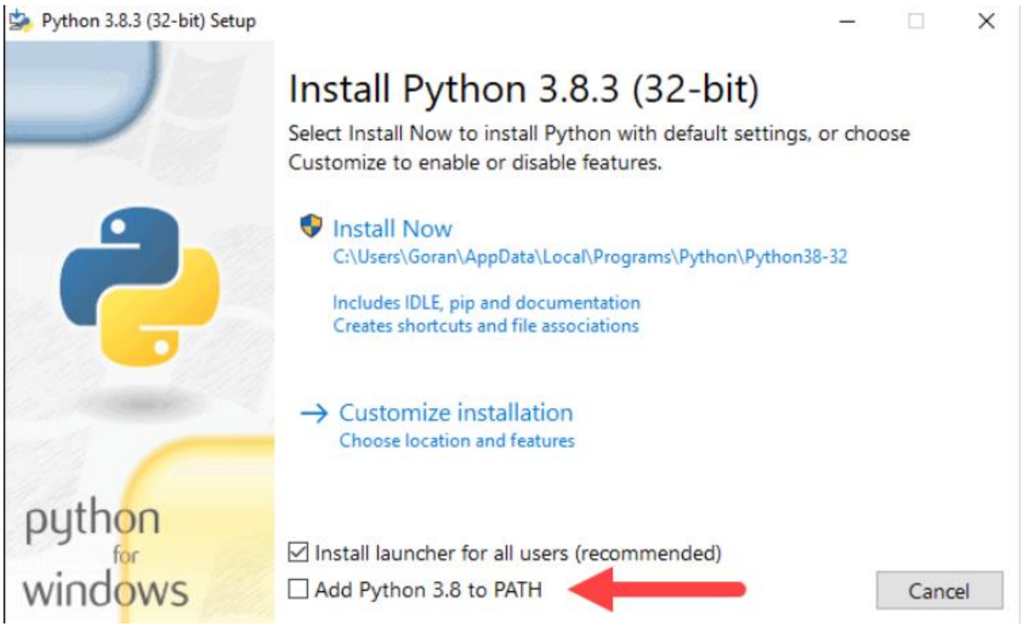
Note that Python 3.5+ *cannot* be used on Windows XP or earlier.

Not the OS you are looking for? Python can be used on many operating systems and environments.

View the full list of downloads.

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

5. Next, click Customize installation.



Install Python 3.8.3 (32-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\Goran\AppData\Local\Programs\Python\Python38-32

Includes IDLE, pip and documentation
Creates shortcuts and file associations

Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☐ Add Python 3.8 to PATH

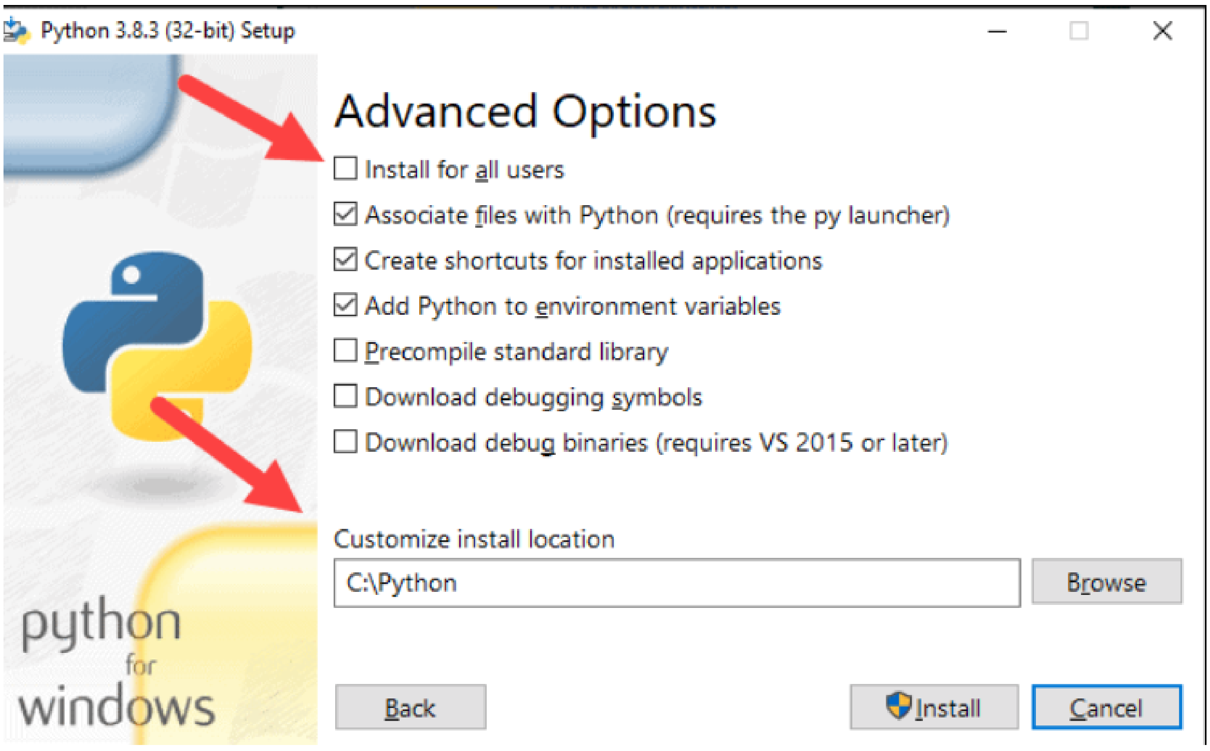
6. You can leave all boxes checked at this step, or you can uncheck the options you do not want.

7. Click Next.

8. Select the box Install for all users and leave other boxes as they are.

9. Under Customize install location, click Browse and navigate to the C drive. Add a new folder and name its Python.

10. Select that folder and click OK.



Python 3.8.3 (32-bit) Setup

Advanced Options

- [ ] Install for all users
- [✓] Associate files with Python (requires the py launcher)
- [✓] Create shortcuts for installed applications
- [✓] Add Python to environment variables
- [ ] Precompile standard library
- [ ] Download debugging symbols
- [ ] Download debug binaries (requires VS 2015 or later)

Customize install location

C:\Python

11. Click Install, and let the installation complete.

12. When the installation completes, click the Disable path length limit option at the bottom and then click Close.

13. If you have a command prompt open, restart it. Verify the installation by checking the version of Python:

```
python --version
```

The output should print Python 3.8.3.

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

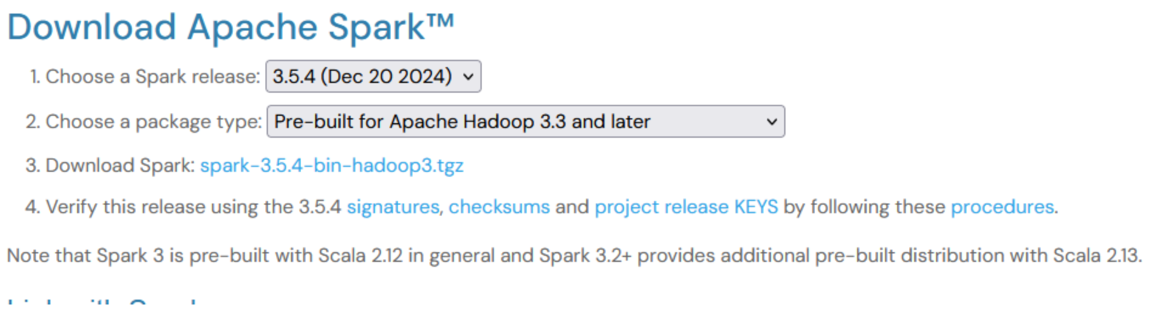
Step 3: Download Apache Spark

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 3.5.4 (Dec 20 2024).

• In the second drop-down Choose a package type, leave the selection Pre-built for Apache Hadoop 2.7.

3. Click the spark-3.5.4-bin-hadoop3.tgz link



Download Apache Spark™

1. Choose a Spark release: 3.5.4 (Dec 20 2024)
2. Choose a package type: Pre-built for Apache Hadoop 3.3 and later
3. Download Spark: spark-3.5.4-bin-hadoop3.tgz
4. Verify this release using the 3.5.4 signatures, checksums and project release KEYS by following these procedures.

Note that Spark 3 is pre-built with Scala 2.12 in general and Spark 3.2+ provides additional pre-built distribution with Scala 2.13.

```
certutil -hashfile c:\users\username\Downloads\spark-3.5.4-bin-hadoop3.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
```

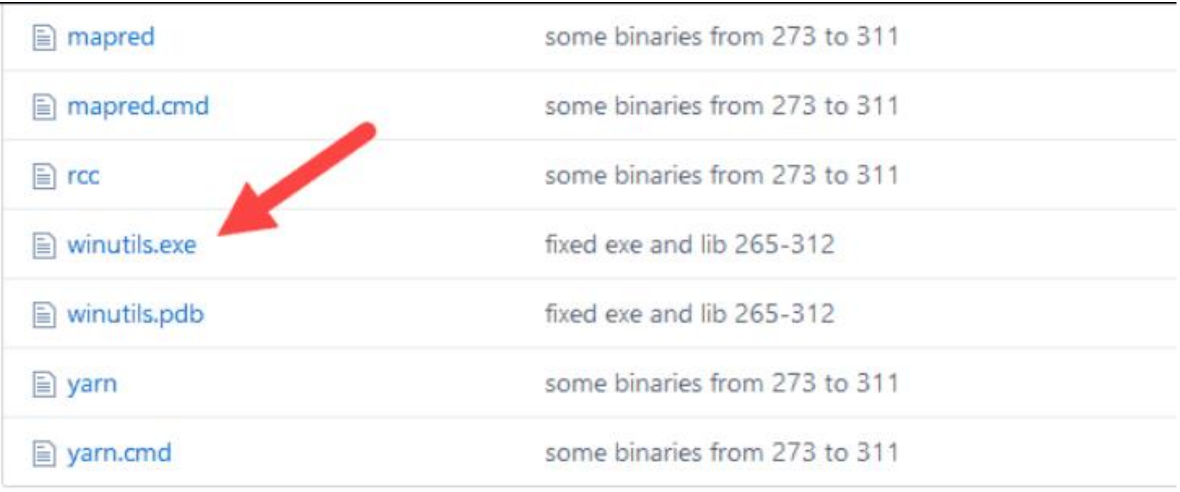
2. In Explorer, locate the Spark file you downloaded.
3. Right-click the file and extract it to C:\Spark using the tool you have on your system (e.g., 7-Zip).

4. Now, your C:\Spark folder has a new folder spark-3.5.4-bin-hadoop3.tgz 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.



mapred	some binaries from 273 to 311
mapred.cmd	some binaries from 273 to 311
rcc	some binaries from 273 to 311
winutils.exe	fixed exe and lib 265-312
winutils.pdb	fixed exe and lib 265-312
yarn	some binaries from 273 to 311
yarn.cmd	some binaries from 273 to 311

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**.

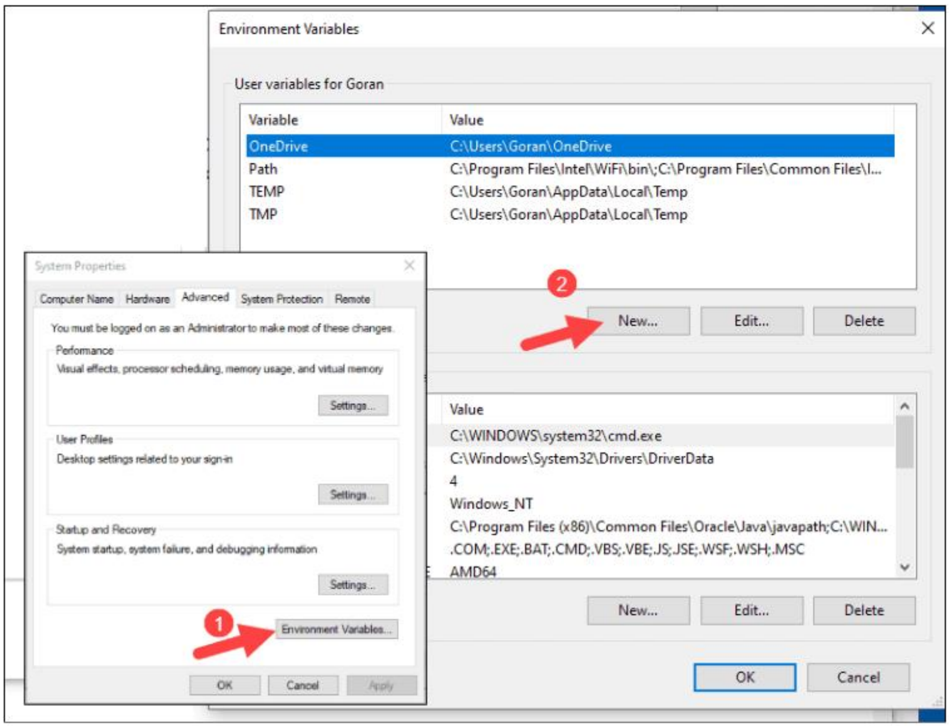
Step 7: Configure Environment Variables

Configuring environment variables in Windows 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 labelled **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.



User variables for Goran

Variable	Value
OneDrive	C:\Users\Goran\OneDrive
Path	C:\Program Files\Intel\WiFi\bin\;C:\Program Files\Common Files\I...
TEMP	C:\Users\Goran\AppData\Local\Temp
TMP	C:\Users\Goran\AppData\Local\Temp

Value
C:\WINDOWS\system32\cmd.exe
C:\Windows\System32\Drivers\DriverData
4
Windows_NT
C:\Program Files (x86)\Common Files\Oracle\Java\javapath;C:\WIN...
.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
AMD64

You must be logged on as an Administrator to make most of these changes.

Performance
Visual effects, processor scheduling, memory usage, and virtual memory

User Profiles
Desktop settings related to your sign-in

Startup and Recovery
System startup, system failure, and debugging information

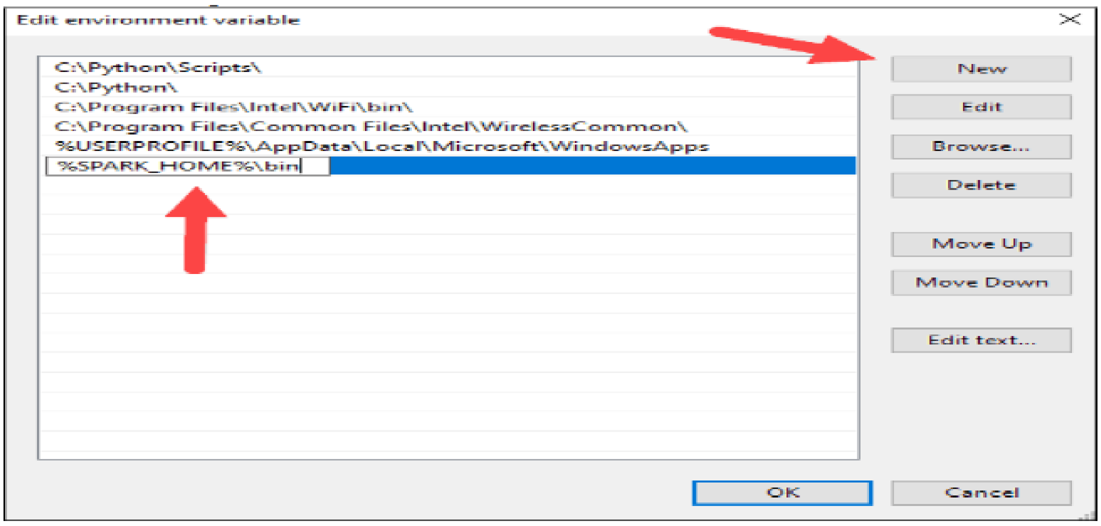
4. For Variable Name type **SPARK_HOME**.

5. For Variable Value type **C:\Spark\spark-3.5.4-bin-hadoop3.0** 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-3.5.4-bin-hadoop3.0\bin**. We recommend using **%SPARK_HOME%\bin** to avoid possible issues with the path.



```
C:\Python\Scripts\
C:\Python\
C:\Program Files\Intel\WiFi\bin\
C:\Program Files\Common Files\Intel\WirelessCommon\
%USERPROFILE%\AppData\Local\Microsoft\WindowsApps
%SPARK_HOME%\bin
```

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

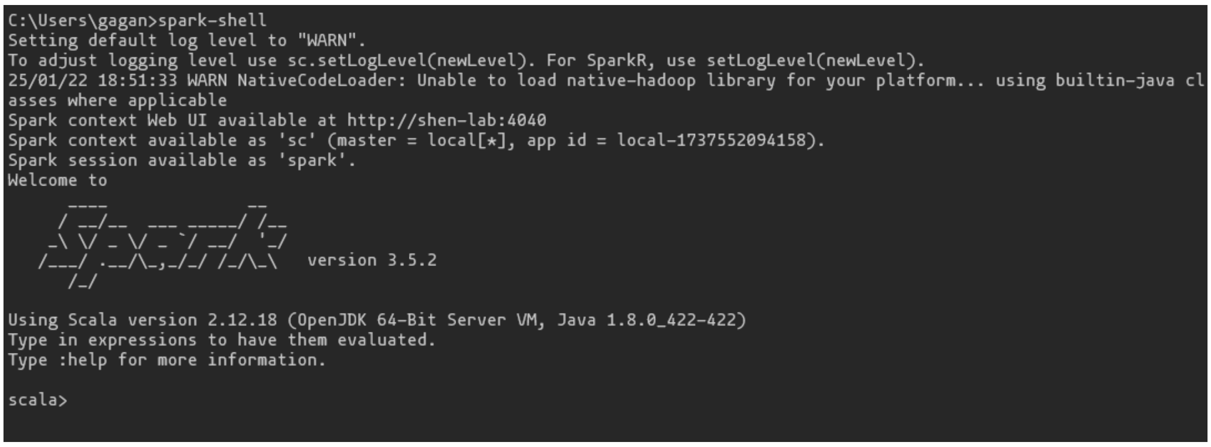
1. Open a new command-prompt window using the right-click and **Run as administrator**:
2. To start Spark, enter

```
C:\Spark\spark-3.5.4-bin-hadoop3.0\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

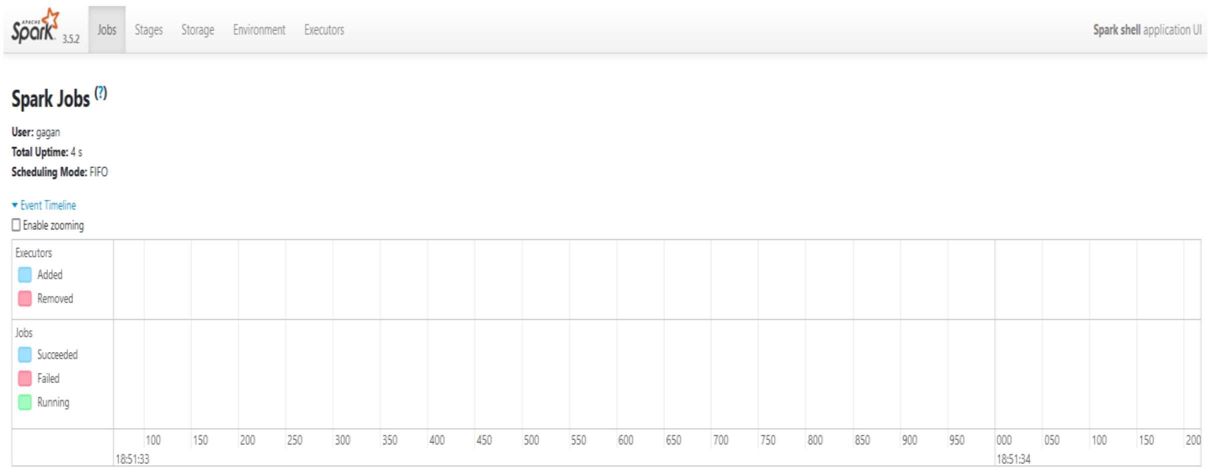


```
C:\Users\gagan>spark-shell
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
25/01/22 18:51:33 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java cl
asses where applicable
Spark context Web UI available at http://shen-lab:4040
Spark context available as 'sc' (master = local[*], app id = local-1737552094158).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.5.2
      /_/

Using Scala version 2.12.18 (OpenJDK 64-Bit Server VM, Java 1.8.0_422-422)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

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.



Spark Jobs (?)

User: gagan
Total Uptime: 4 s
Scheduling Mode: FIFO

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

Practical 2

Aim: Spark Basics and RDD interface

Theory:

Spark Basics – Apache Spark is an open-source distributed computing system

designed for big data processing and analytics. Spark provides a unified framework for batch processing, real-time streaming, machine learning, and graph processing.

At its core, Spark operates on the concept of Resilient Distributed

Datasets (RDDs), which are immutable collections of objects distributed across a cluster.

Spark offers fault tolerance through lineage information, enabling the reconstruction of lost data partitions in case of failure.

One of Spark's key features is its in-memory computing capability, allowing for faster data processing compared to traditional disk-based systems like Hadoop MapReduce.

RDD Interface – RDD (Resilient Distributed Dataset) is the fundamental data structure in Apache Spark.RDD represents an immutable, partitioned collection of records that

can be processed in parallel across a distributed cluster.

It provides fault tolerance through lineage information, enabling the reconstruction of lost data partitions. RDDs support two types of operations: transformations, which create

new RDDs from existing ones, and actions, which perform computations and return results to the driver program.

RDDs can be created from external data sources like HDFS, HBase,

or by parallelizing an existing collection in the driver program.

Practical 03

Aim: Filtering RDDs, and the Minimum Temperature by Location Example

RDD: An RDD (Resilient Distributed Dataset) is the fundamental data structure in Apache Spark, representing a distributed collection of immutable objects partitioned across a cluster.

It supports fault tolerance and parallel processing by tracking lineage for recomputation in case of failure. RDDs enable transformations (e.g., map, filter) and actions (e.g., collect, reduce) for large-scale data processing.

- **map():** Applies a transformation function to each element in an RDD, returning a new RDD with the transformed elements. Example: Doubling all numbers in a dataset.
- **filter():** Selects elements from an RDD that satisfy a given condition, returning a new RDD with only those elements. Example: Extracting even numbers from a dataset.
- **collect():** Retrieves all elements of an RDD or DataFrame and brings them to the driver node as a Python list. It’s useful for small datasets but should be avoided for large datasets to prevent memory issues. Example: Viewing all processed results locally.

Practical 03:

```
# Initialize Spark

from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("MinTemperatures")

sc = SparkContext(conf = conf)


# Function to parse line

def parseLine(line):

fields = line.split(',')

stationID = fields[0]

entryType = fields[2]

temperature = float(fields[3]) * 0.1 * (9.0 / 5.0) + 32.0

return (stationID, entryType, temperature)


# Find Minimum Temperature City

lines = sc.textFile("1800.csv")

parsedLines = lines.map(parseLine)

minTemps = parsedLines.filter(lambda x: "TMIN" in x[1])

stationTemps = minTemps.map(lambda x: (x[0], x[2]))

minTemps = stationTemps.reduceByKey(lambda x, y: min(x,y))

results = minTemps.collect();


# Show the result for TMIN Entry Type

for result in results:

print(result[0] + "\t{:.2f}C".format(result[1]))


# Find Maximum Temperature City

parsedLines = lines.map(parseLine)

maxTemps = parsedLines.filter(lambda x: "TMAX" in x[1])

stationTemps = maxTemps.map(lambda x: (x[0], x[2]))

maxTemps = stationTemps.reduceByKey(lambda x, y: max(x,y))

results = maxTemps.collect();


# Show the result for TMAX Entry Type

for result in results:

print(result[0] + "\t{:.2f}C".format(result[1]))
```

Output:

```
# Show the result for TMIN Entry Type

ITE00100554 5.36C

EZE00100082 7.70C


# Show the result for TMAX Entry Type

ITE00100554       90.14C

EZE00100082       90.14C
```

Practical 04

Aim: Counting Word Occurrences using flatmap()

Counting word occurrences in a large dataset is a common task in data processing, particularly in natural language processing (NLP) and big data applications.

In Apache Spark, this task can be efficiently achieved using its distributed computing framework, which allows processing large datasets in parallel across multiple nodes. Here's how the concept works and why it's beneficial:

Transformations:

- **flatMap()**: This transformation is essential for word counting because it allows us to take each line of text and break it down into individual words. It "flattens" the output into a single RDD containing all words across the entire dataset.
    - Example: Splitting each line into words (x.split()), which may return multiple words for each input line.
- **map()**: Could also be used for word counting, but it doesn't flatten the result, hence flatMap() is preferred when you expect a variable number of outputs from each input element

Actions:

- **countByValue()**: This action computes the frequency of each word in the dataset by counting how many times each unique word appears. It returns the result as a dictionary of word-count pairs, where the key is the word, and the value is its count.

Practical 04:

```
from pyspark import SparkConf, SparkContext


conf = SparkConf().setMaster("local").setAppName("WordCount")
sc = SparkContext(conf = conf)


input = sc.textFile("file:///sparkcourse/book.txt")
words = input.flatMap(lambda x: x.split())
wordCounts = words.countByValue()


for word, count in wordCounts.items():
   cleanWord = word.encode('ascii', 'ignore')
   if (cleanWord):
      print(cleanWord.decode() + " " + str(count))
```

Output

```
campaign. 1 relatively 5 inexpensive 1 tests 1 best. 1 Whenever 1 investment 1 Channels 1 countless
1 Twitter, 1 Instagram, 1 LinkedIn, 2 Bing, 1 Yahoo, 1 advertiser. 1 places. 1 consumer-oriented 1
Twitter 7 presence. 1 backwards 1 Target 1 gambling 1 $100, 1 lowest-cost 1
……….
consideration. 1 back-of-the-envelope 1 calculations 1 worthwhile. 2 printed, 1 returned 1unsold. 1
```

Practical 05

Aim: Executing SQL Commands and SQL-Style Functions on a DataFrame in Spark

In Spark, the **DataFrame API** provides a convenient interface for handling structured data, and SQL-style queries can be executed on DataFrames to perform a variety of operations.

These operations enable powerful and efficient data processing with familiar SQL syntax, which can be especially useful for users transitioning from traditional databases or those comfortable with SQL queries.

Few key concepts to use SQL Command in Spark

1. **SparkSession**:
   a. **SparkSession** is the entry point for working with structured data in Spark. It allows you to interact with the Spark SQL engine and execute SQL commands on DataFrames.
   b. You can use SparkSession.sql() to run SQL queries directly on DataFrames or temporary views
2. **Creating a DataFrame:**
   a. DataFrames can be created from various data sources such as CSV, Parquet, JSON, and more. Once created, they can be used for SQL queries.
   b. DataFrames also support transformation operations using methods like `.select()`,`.filter()`, `.groupBy()`, etc.
3. **Temporary Views**:
   a. Before executing SQL queries, DataFrames can be registered as temporary views, which are treated as tables in SQL queries.
   b. Temporary views exist only during the Spark session, meaning they are session-scoped and not persistent.
4. **SQL Queries on DataFrames**:
   a. After creating a temporary view, you can execute SQL queries on the DataFrame using spark.sql(). This allows for traditional SQL operations like SELECT, WHERE, GROUP BY, ORDER BY, and joins on DataFrames.
5. **SQL-Style Functions**:
   a. Spark DataFrames support a variety of SQL-style functions that mimic SQL commands and enable complex data processing. These include:
      i. **Aggregation functions** like `COUNT()`, `SUM()`, `AVG()`, `MAX()`, `MIN()`.
      ii. **String functions** like `CONCAT()`, `LOWER()`, `UPPER()`, `LENGTH()`.
      iii. **Conditional functions** like `IFNULL()`, `COALESCE()`, `CASE WHEN`.
      iv. **Window functions** for operations over partitions of data, such as `ROW_NUMBER()`, `RANK()`, `LEAD()`, and `LAG()`.
      v. **Mathematical functions** like `ABS()`, `ROUND()`, `POW()`, `EXP()`.

Practical 05:

```
from pyspark.sql import SparkSession

from pyspark.sql import Row

import collections


# Create a SparkSession (Note, the config section is only for Windows!)

spark = SparkSession.builder.config("spark.sql.warehouse.dir",
"file:///C:/temp").appName("SparkSQL").getOrCreate()


def mapper(line):

   fields = line.split(',')

   return Row(ID=int(fields[0]), name=str(fields[1].encode("utf-8")), age=int(fields[2]),
numFriends=int(fields[3]))


lines = spark.sparkContext.textFile("fakefriends.csv")

people = lines.map(mapper)


# Infer the schema, and register the DataFrame as a table.

schemaPeople = spark.createDataFrame(people).cache()

schemaPeople.createOrReplaceTempView("people")


# SQL can be run over DataFrames that have been registered as a table.

teenagers = spark.sql("SELECT * FROM people WHERE age >= 13 AND age <= 19")


# The results of SQL queries are RDDs and support all the normal RDD operations.

for teen in teenagers.collect():

  print(teen)


# We can also use functions instead of SQL queries:

schemaPeople.groupBy("age").count().orderBy("age").show()


spark.stop()
```

Output:

```
# Output 1
Row(ID=21, name="b'Miles'", age=19, numFriends=268)
Row(ID=52, name="b'Beverly'", age=19, numFriends=269)
Row(ID=54, name="b'Brunt'", age=19, numFriends=5)
Row(ID=106, name="b'Beverly'", age=18, numFriends=499)
Row(ID=115, name="b'Dukat'", age=18, numFriends=397)
Row(ID=133, name="b'Quark'", age=19, numFriends=265)
.....
Row(ID=409, name="b'Nog'", age=19, numFriends=267)
Row(ID=439, name="b'Data'", age=18, numFriends=417)
Row(ID=444, name="b'Keiko'", age=18, numFriends=472)
Row(ID=492, name="b'Dukat'", age=19, numFriends=36)
Row(ID=494, name="b'Kasidy'", age=18, numFriends=194)


# Output 2
+---+-----+
|age|count|
+---+-----+
| 18|    8|
| 19|   11|
| 20|    5|
| 21|    8|
| 22|    7|
| 30|   11|
....
| 31|    8|
| 32|   11|
| 33|   12|
| 36|   10|
| 37|    9|
+---+-----+
```

Practical 06

Aim: Implement Total Spent by Customer with DataFrames

The task of calculating the total spent by each customer involves transforming raw data into structured, aggregated results. Using **PySpark** to implement this operation allows leveraging distributed computing for big data processing

Few key concepts to by find total spent

Key-Value Pair RDD

1. **reduceByKey():** is an **aggregation** operation on RDDs. It is used to combine values with the same key. In this case, for each customer (identified by their unique ID), it sums the total amount spent.

   The **lambda function** lambda x, y: x + y is used to add the amounts together.

   Also `reduceByKey()` works on **key-value pairs** in the RDD. In this case, customer IDs are the keys, and the amount spent is the value.

   Key-value pairs are useful in scenarios where you need to aggregate or process data based on a common key, such as summing up expenses for each customer.

   reduceByKey() operates efficiently by performing local aggregations first (in each partition) and then performing a global aggregation.

Practical No: 06:

```
from pyspark import SparkConf, SparkContext


conf = SparkConf().setMaster("local").setAppName("SpendByCustomer")

sc = SparkContext(conf = conf)


def extractCustomerPricePairs(line):

   fields = line.split(',')

   return (int(fields[0]), float(fields[2]))


input = sc.textFile("file:///sparkcourse/customer-orders.csv")

mappedInput = input.map(extractCustomerPricePairs)

totalByCustomer = mappedInput.reduceByKey(lambda x, y: x + y)


results = totalByCustomer.collect();

for result in results:

   print(result)
```

Output:

```
(44, 4714) (35, 5106) (2, 5940) (47, 4275) (29, 4979) (91, 4599) (70, 5318) (85, 5455) (53, 4898) (14,
4690) (51, 4929) (42, 5651) (79, 3750) (50, 4462) (20, 4790) (15, 5364) (5, 4517) (48, 4341) (31, 4717)
(4, 4765) (36, 4236) (57, 4582) (12, 4617)

…

(27, 4865) (78, 4483) (83, 4586) (6, 5339) (26, 5194) (75, 4135) (25, 5008) (71, 5941) (39, 6131) (60,
4994) (97, 5921)
```

Practical 07

Aim: Use Broadcast Variables to Display Movie Names Instead of ID Numbers

PySpark provides the broadcast variables allow you to efficiently share large, read-only datasets across all worker nodes in a cluster without needing to send them multiple times.

This is useful when you need to reference a small dataset (like a mapping of movie IDs to movie names) during computations on a large distributed dataset.

To perform the experiment first understand few key concepts:

1. **Broadcast Variables**:

   A broadcast variable is a read-only variable that is distributed across all nodes in the Spark cluster. This means that instead of sending large data multiple times (once per task), the broadcasted variable is copied to each worker node once, reducing the amount of data transferred across the network

   They are useful when working with a small lookup dataset (e.g., mapping movie IDs to names) while processing a large dataset (e.g., user ratings).

   Broadcasting reduces the overhead of repeatedly sending the same small data to every task, optimizing both memory usage and network efficiency.

   The `broadcast()` method in PySpark is used to create a broadcast variable.

   To lookup values from a broadcasted variable inside an RDD transformation, you can use it in a map() or similar operation.

   The broadcast variable's value is accessed using the '.value' attribute. This is how you reference the broadcasted data in your transformations or actions.

   This access is efficient because the broadcasted data is stored in-memory on each node, so no additional network overhead is incurred when accessing it.

   While broadcast variables are efficient for small to moderately sized datasets, they should be used with caution when the broadcasted dataset grows large. Broadcasting a very large dataset could lead to memory overflow on the worker nodes.

Practical No 07:

```
from pyspark.sql import SparkSession, Row,functions


def loadMovies():
   movieNames = {}
   file_path = r"c:\Users\gagan\Downloads\ml-100k\u.item"
   with open(file_path) as f:
      for line in f:
         fields = line.split('|')
         movieNames[int(fields[0])] = fields[1]
   return movieNames


spark = SparkSession.builder.appName("BroadcastVariable").getOrCreate()


nameID = loadMovies()
nameID_broadcast = spark.sparkContext.broadcast(nameID)


lines = spark.sparkContext.textFile("u.data")
movies = lines.map(lambda x: Row(movieID= int(x.split()[1])))
movieDataset = spark.createDataFrame(movies)
topMoviesID = movieDataset.groupBy("movieID").count().orderBy("count",
ascending = False).cache()


def addMovieName(movieID):
   return nameID_broadcast.value[movieID]
addMovieNameUDF = functions.udf(addMovieName)
topMoviesWithNames = topMoviesID.withColumn("movieName",
addMovieNameUDF(functions.col("movieID")))


topMoviesWithNames.select("movieName", "count").show(10, truncate=False)


spark.stop()
```

Output:

```
+----------------------------+-----------------------+
|movieName                   |count|
+----------------------------+-----------------------+
|Star Wars (1977)            |583  |
|Contact (1997)              |509  |
|Fargo (1996)                |508  |
|Return of the Jedi (1983)   |507  |
|Liar Liar (1997)            |485  |
|English Patient, The (1996) |481  |
|Scream (1996)               |478  |
|Toy Story (1995)            |452  |
|Air Force One (1997)        |431  |
|Independence Day (ID4) (1996)|429  |
+----------------------------+-----------------------+
only showing top 10 rows
```

Practical 08

Aim: Using Spark ML to Produce Movie Recommendations

To develop a movie recommendation system using Spark MLlib by applying the ALS (Alternating Least Squares) algorithm for collaborative filtering, enabling personalized movie suggestions based on user ratings

For performing an experiment first understand few key concepts:

2. **Collaborative Filtering**:

   Collaborative filtering is a technique used in recommendation systems to predict a user's interests by collecting preferences or tastes from many users. It builds a model from the past behaviors of users, such as the ratings of items (e.g., movies) by users.

   Types:

   - **User-based Collaborative Filtering**: Recommends items based on what similar users liked.
   - **Item-based Collaborative Filtering**: Recommends items that are similar to those the user has already liked.

   In the case of Spark, **ALS** (Alternating Least Squares) is used for matrix factorization in collaborative filtering to make these predictions.

3. **ALS (Alternating Least Squares)**:

   ALS is a matrix factorization algorithm that is widely used for collaborative filtering in recommendation systems. It is designed to find a low-rank approximation of the user-item rating matrix.

4. **Data Splitting:**

   Train-Test Split: To build a recommendation model, the dataset is typically split into training and testing sets to evaluate its accuracy.

   Training Set: Used to train the model, learning patterns in user ratings.

   Testing Set: Used to evaluate the model's performance by comparing the predicted ratings with actual ratings.

5. **Evaluating Model Performance**:

   RMSE is a common metric for evaluating the performance of regression models, including recommendation systems.

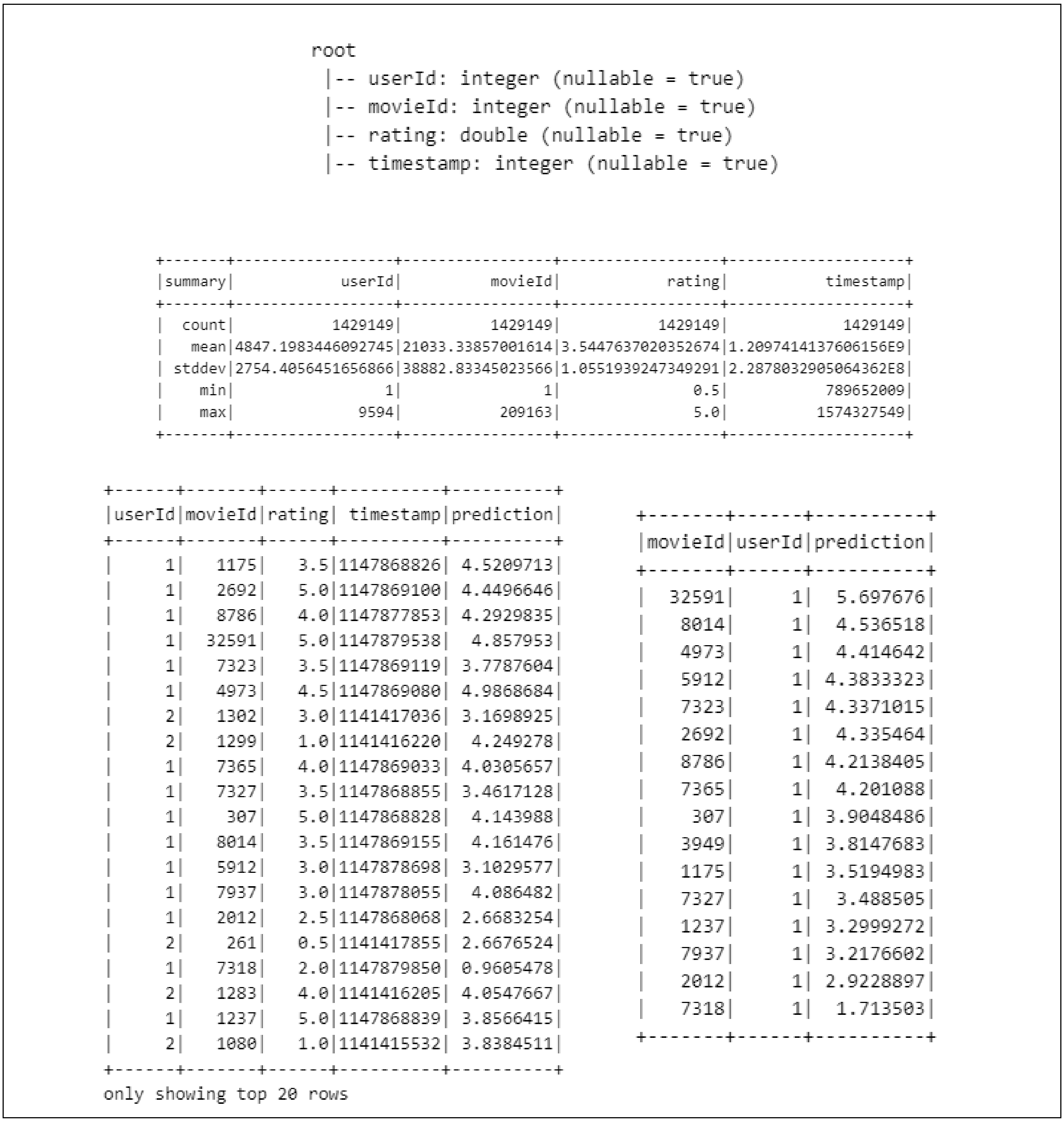
6. **Model Training**:
   The `fit()` method is used to train the ALS model on the training dataset.

7. **Model Predictions**:
   After the model is trained, it is used to generate predictions on the test data by calling the `transform()` method:

Practical No 08:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('recommendation').getOrCreate()

from pyspark.ml.evaluation import RegressionEvaluator

from pyspark.ml.recommendation import ALS


data = spark.read.csv('ratings.csv',inferSchema=True,header=True)


data.head()

data.printSchema()

data.describe().show()


(train_data, test_data) = data.randomSplit([0.8, 0.2], seed=42)


als = ALS(maxIter=5, regParam=0.01, userCol="userId", itemCol="movieId", ratingCol="rating")

model = als.fit(train_data)


# Evaluate the model by computing the RMSE on the test data

predictions = model.transform(test_data)

predictions.show()


evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating",predictionCol="prediction")

rmse = evaluator.evaluate(predictions)

print("Root-mean-square error = " + str(rmse))


single_user = test_data.filter(test_data['userId']==3).select(['movieId','userId'])

single_user.show()

reccomendations = model.transform(single_user)


reccomendations.orderBy('prediction',ascending=False).show()
```

Output:



```
root
 |-- userId: integer (nullable = true)
 |-- movieId: integer (nullable = true)
 |-- rating: double (nullable = true)
 |-- timestamp: integer (nullable = true)
```

```
+-------+-----------------+----------------+------------------+--------------------+
|summary|           userId|         movieId|            rating|           timestamp|
+-------+-----------------+----------------+------------------+--------------------+
|  count|          1429149|         1429149|           1429149|             1429149|
|   mean|4847.1983446092745|21033.33857001614|3.5447637020352674|1.209741413760615 6E9|
| stddev|2754.4056451656866|38882.83345023566|1.0551939247349291|2.2878032905064362E8|
|    min|                1|               1|               0.5|           789652009|
|    max|             9594|          209163|               5.0|          1574327549|
+-------+-----------------+----------------+------------------+--------------------+
```

```
+------+-------+------+----------+----------+
|userId|movieId|rating| timestamp|prediction|
+------+-------+------+----------+----------+
|     1|   1175|   3.5|1147868826| 4.5209713|
|     1|   2692|   5.0|1147869100| 4.4496646|
|     1|   8786|   4.0|1147877853| 4.2929835|
|     1|  32591|   5.0|1147879538|  4.857953|
|     1|   7323|   3.5|1147869119| 3.7787604|
|     1|   4973|   4.5|1147869080| 4.9868684|
|     2|   1302|   3.0|1141417036| 3.1698925|
|     2|   1299|   1.0|1141416220|  4.249278|
|     1|   7365|   4.0|1147869033| 4.0305657|
|     1|   7327|   3.5|1147868855| 3.4617128|
|     1|    307|   5.0|1147868828|  4.143988|
|     1|   8014|   3.5|1147869155|  4.161476|
|     1|   5912|   3.0|1147878698| 3.1029577|
|     1|   7937|   3.0|1147878055|  4.086482|
|     1|   2012|   2.5|1147868068| 2.6683254|
|     2|    261|   0.5|1141417855| 2.6676524|
|     1|   7318|   2.0|1147879850| 0.9605478|
|     2|   1283|   4.0|1141416205| 4.0547667|
|     1|   1237|   5.0|1147868839| 3.8566415|
|     2|   1080|   1.0|1141415532| 3.8384511|
+------+-------+------+----------+----------+
only showing top 20 rows
```

```
+-------+------+----------+
|movieId|userId|prediction|
+-------+------+----------+
|  32591|     1|  5.697676|
|   8014|     1|  4.536518|
|   4973|     1|  4.414642|
|   5912|     1| 4.3833323|
|   7323|     1| 4.3371015|
|   2692|     1|  4.335464|
|   8786|     1| 4.2138405|
|   7365|     1|  4.201088|
|    307|     1| 3.9048486|
|   3949|     1| 3.8147683|
|   1175|     1| 3.5194983|
|   7327|     1|  3.488505|
|   1237|     1| 3.2999272|
|   7937|     1| 3.2176602|
|   2012|     1| 2.9228897|
|   7318|     1|  1.713503|
+-------+------+----------+
```

Practical 09

Aim: Use Windows with Structured Streaming

The objective of this experiment is to explore and implement Windows with Structured Streaming in Apache Spark to track real-time data, such as counting the most-viewed URLs.

The experiment demonstrates how to use the powerful features of Structured Streaming to process and analyse continuous data streams, leveraging key concepts like windowing, aggregation, and SQL-style operations.

Specifically, this experiment aims to:

1. **Structured Streaming**: This is the new paradigm for processing streaming data in Spark, based on the **DataFrame API**. It allows for easier integration with batch processing, and processing is done using SQL-style operations.
2. **SocketStream**: It can be used to read streaming data from a socket, like the one in your code, which will provide continuous input to Spark for processing.
3. **GroupBy and Window Operations**: These operations can be used to calculate metrics over time windows (e.g., 1-minute window).
4. **Continuous Query**: In Structured Streaming, you define a query that continuously processes incoming data, outputs results, and optionally triggers some action (like writing to a database or console).
5. **awaitTermination()**: This keeps the stream running until it's manually terminated. The stream continuously receives data and processes it based on the logic defined in the Structured Streaming query.

Practical No 09:

```
from pyspark import SparkContext

from pyspark.streaming import StreamingContext


# Create a local StreamingContext with two working thread and batch interval of 1 second

sc = SparkContext("local[2]", "NetworkWordCount")

ssc = StreamingContext(sc, 1)


lines = ssc.socketTextStream("localhost", 9999)

words = lines.flatMap(lambda line: line.split(" "))


# Count each word in each batch

pairs = words.map(lambda word: (word, 1))

wordCounts = pairs.reduceByKey(lambda x, y: x + y)


# Print the first ten elements of each RDD generated in this DStream to the console

wordCounts.pprint()


ssc.start()             # Start the computation

ssc.awaitTermination()  # Wait for the computation to terminate


# May cause deprecation warnings, safe to ignore, they aren't errors

from pyspark import SparkContext

from pyspark.streaming import StreamingContext

from pyspark.sql import SQLContext

from pyspark.sql.functions import desc


sc.stop()

ssc.stop()

sc = SparkContext()


socket_stream = ssc.socketTextStream("127.0.0.1", 5555)
```

Output:

```
--------------------------------------------

Time: 2025-01-23 06:55:15

--------------------------------------------



--------------------------------------------

Time: 2025-01-23 06:55:16

--------------------------------------------

.....

Time: 2025-01-23 06:55:20

--------------------------------------------



--------------------------------------------

Time: 2025-01-23 06:55:21

--------------------------------------------



--------------------------------------------

Time: 2025-01-23 06:55:22

--------------------------------------------



--------------------------------------------

Time: 2025-01-23 06:55:34

--------------------------------------------
```