Practical 1

Aim: Installation of Apache Spark

Theory:

Apache Spark is an open-source, distributed computing system used for big data processing and analytics. It provides a fast and general-purpose cluster-computing framework for large-scale data processing. Spark is designed to be flexible, easy to use, and compatible with various data sources and languages.

Steps to Install Apache Spark:

Installing Apache Spark on Windows can be a bit more complex compared to other operating systems like Linux or macOS due to some compatibility issues. However, it's still possible to install and run Spark on Windows. Here's a step-by-step guide to help you get started:

1.Prerequisites:

Make sure you have Java installed on your system. Spark requires Java 8 or later. You can download and install Java from the Oracle website.

2.Download Apache Spark:

Go to the Apache Spark website (https://spark.apache.org/downloads.html) and download the latest version of Spark for Windows. Choose a pre-built package for Hadoop, as it includes the necessary Hadoop binaries.

Unzip the downloaded file to a location on your system. For example, C:\spark.

3.Set Environment Variables:

Add the following environment variables:

SPARK_HOME = C:\spark

HADOOP_HOME = C:\spark

JAVA_HOME = [Your Java installation path]

Add %SPARK_HOME%\bin and %HADOOP_HOME%\bin to your PATH variable.

4.Configuration:

Rename the file spark-env.cmd.template located in the conf directory of Spark to spark-env.cmd.

Edit spark-env.cmd and set HADOOP_HOME to your Hadoop directory.

Optionally, you can configure other Spark settings in this file as well.

5.Starting Spark:

Open a Command Prompt.

Navigate to the Spark directory (cd C:\spark).

Run bin\spark-shell.cmd to start the Spark shell or bin\pyspark.cmd for Python.

You can also start Spark applications using bin\spark-submit.cmd.

6.Testing the Installation:

Once the shell starts, you should see the Spark logo and a prompt indicating that Spark is ready.

Try running some sample Spark code to ensure everything is working as expected.

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 3

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

Theory:

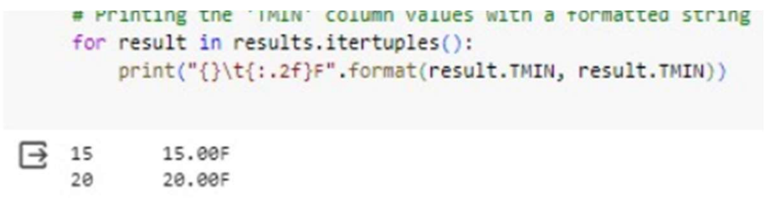
Filtering in RDD involves selecting elements that satisfy a certain condition while discarding those that do not meet the criteria. Spark provides the filter() transformation function that applies a predicate to each element of the RDD, retaining only those that evaluate to true. This operation allows for efficient data reduction, especially when dealing with large datasets distributed across a cluster. Filtering RDDs can be applied to various use cases such as data cleaning, anomaly detection, or extracting specific subsets of data. By leveraging filtering operations in Spark RDDs, users can streamline data processing workflows and focus on relevant information for subsequent analysis or processing steps.

Code:

```
from pyspark import SparkConf, SparkContext

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

sc = SparkContext(conf = conf)

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)

lines = sc.textFile("file:///SparkCourse/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();
for result in results:
print(result[0] + "\t{:.2f}F".format(result[1]))
```

Output:



```
# Printing the `TMIN` column values with a formatted string
for result in results.itertuples():
    print("{}\t{:.2f}F".format(result.TMIN, result.TMIN))
```

```
15      15.00F
20      20.00F
```

Practical 4

Aim: Counting Word Occurrences using flatmap()

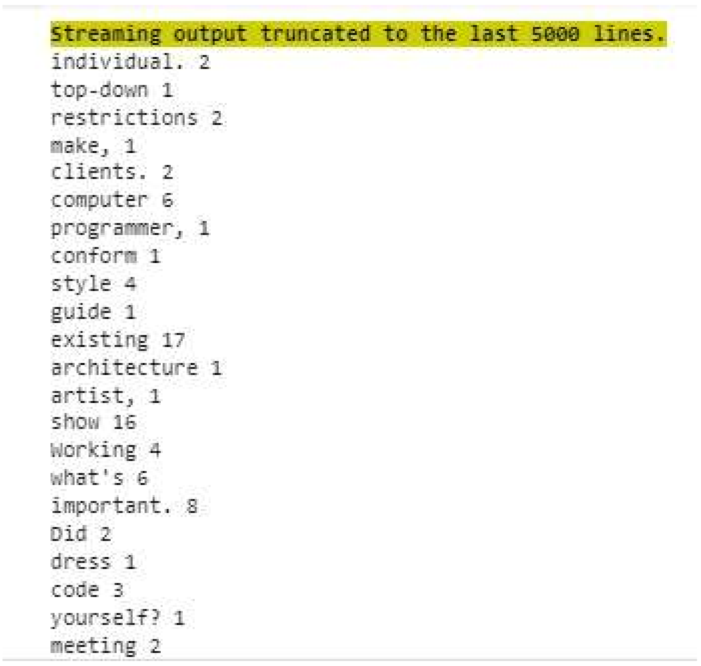
Theory:

flatMap() is a transformation function in Apache Spark that operates on RDDs (Resilient Distributed Datasets). It takes a function as input, applies it to each element of the RDD, and produces zero or more output elements for each input element. Unlike map(), flatMap() can generate multiple output elements for each input element, effectively "flattening" nested collections. This function is commonly used for operations like tokenization, where a string is split into words, resulting in a flattened collection of words. The flatMap() transformation is essential for tasks requiring the transformation of structured or nested data into a more uniform or flat format, facilitating subsequent processing steps.

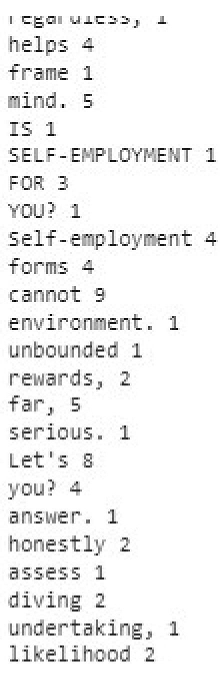
Code:

```
from pyspark import SparkConf, SparkContext

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

input = sc.textFile("book.txt") word = input.flatMap(lambda x :
x.split()) wordCounts = word.countByValue() print(wordCounts)

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

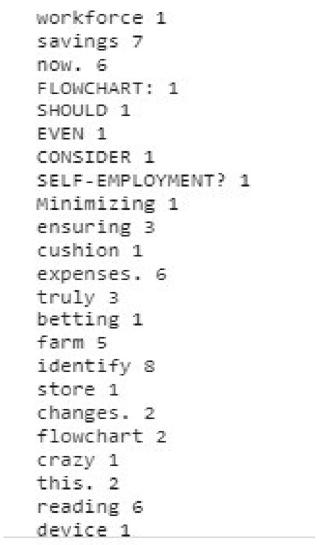
Output:



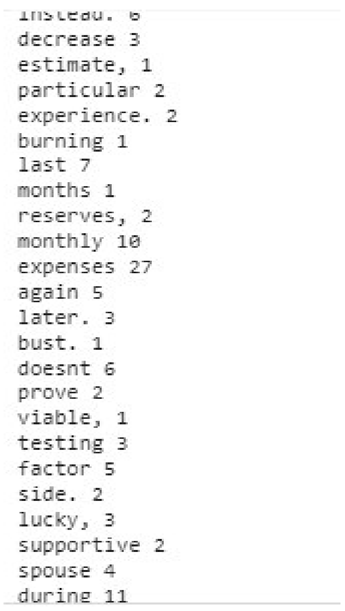
```
Streaming output truncated to the last 5000 lines.
individual. 2
top-down 1
restrictions 2
make, 1
clients. 2
computer 6
programmer, 1
conform 1
style 4
guide 1
existing 17
architecture 1
artist, 1
show 16
Working 4
what's 6
important. 8
Did 2
dress 1
code 3
yourself? 1
meeting 2
```



```
helps 4
frame 1
mind. 5
IS 1
SELF-EMPLOYMENT 1
FOR 3
YOU? 1
Self-employment 4
forms 4
cannot 9
environment. 1
unbounded 1
rewards, 2
far, 5
serious. 1
Let's 8
you? 4
answer. 1
honestly 2
assess 1
diving 2
undertaking, 1
likelihood 2
```



```
workforce 1
savings 7
now. 6
FLOWCHART: 1
SHOULD 1
EVEN 1
CONSIDER 1
SELF-EMPLOYMENT? 1
Minimizing 1
ensuring 3
cushion 1
expenses. 6
truly 3
betting 1
farm 5
identify 8
store 1
changes. 2
flowchart 2
crazy 1
this. 2
reading 6
device 1
```



```
instead. 6
decrease 3
estimate, 1
particular 2
experience. 2
burning 1
last 7
months 1
reserves, 2
monthly 10
expenses 27
again 5
later. 3
bust. 1
doesnt 6
prove 2
viable, 1
testing 3
factor 5
side. 2
lucky, 3
supportive 2
spouse 4
during 11
```

Practical 5

Aim: Executing SQL commands and SQL-style functions on a Data Frame

Theory:

Spark SQL is a module in Apache Spark that provides a high-level abstraction for working with structured and semi-structured data. It allows users to execute SQL queries and DataFrame operations on distributed datasets, seamlessly integrating SQL queries with Spark's processing engine. Spark SQL supports various data sources, including JSON, CSV, Parquet, and Hive tables, enabling unified access to different data formats. Additionally, it offers optimizations like Catalyst optimizer and Tungsten execution engine for efficient query processing. Spark SQL is widely used for data analysis, data exploration, and building data pipelines in Spark applications, offering both SQL and programmatic interfaces for data manipulation and querying.

Code:

```
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")

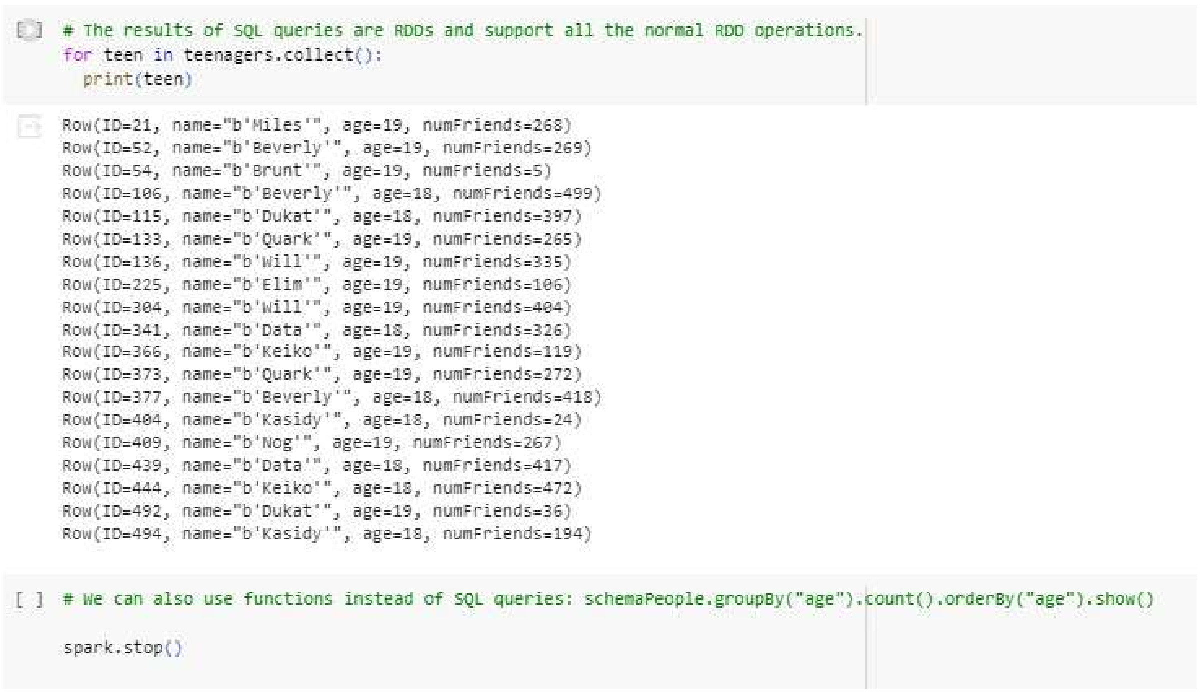
The results of SQL queries are RDDs and support all the normal RDD operations.

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:



```
# The results of SQL queries are RDDs and support all the normal RDD operations.
for teen in teenagers.collect():
  print(teen)
```

```
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=136, name="b'Will'", age=19, numFriends=335)
Row(ID=225, name="b'Elim'", age=19, numFriends=106)
Row(ID=304, name="b'Will'", age=19, numFriends=404)
Row(ID=341, name="b'Data'", age=18, numFriends=326)
Row(ID=366, name="b'Keiko'", age=19, numFriends=119)
Row(ID=373, name="b'Quark'", age=19, numFriends=272)
Row(ID=377, name="b'Beverly'", age=18, numFriends=418)
Row(ID=404, name="b'Kasidy'", age=18, numFriends=24)
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)
```

```
[ ] # We can also use functions instead of SQL queries: schemaPeople.groupBy("age").count().orderBy("age").show()

    spark.stop()
```

Practical 6

Aim: Implement Total Spent by Customer with DataFrames

Theory:

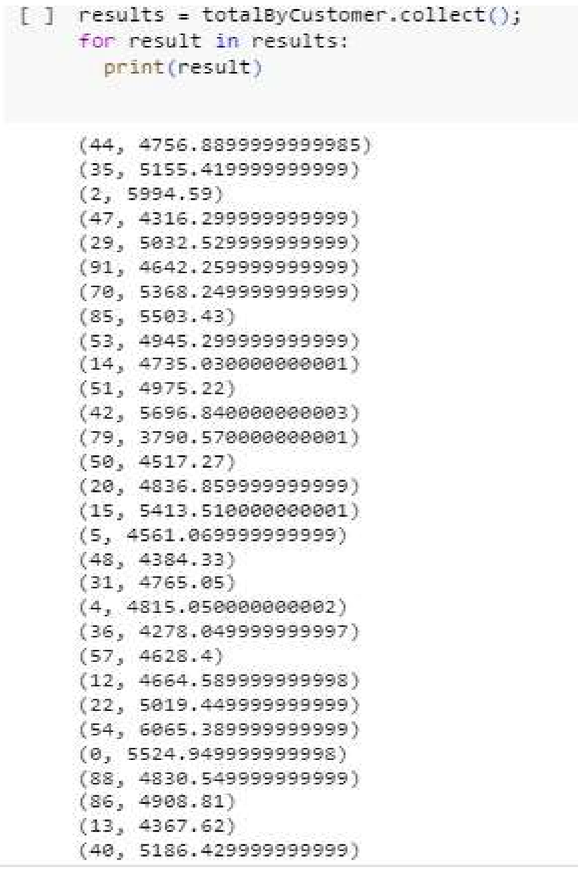
DataFrames in Apache Spark provide a structured representation of data, organizing it into rows and columns. Leveraging Spark's optimized execution engine, DataFrames offer efficient processing of data, particularly for complex transformations and aggregations. Spark DataFrames support various data formats and integrate seamlessly with Spark SQL, enabling both SQL queries and DataFrame operations. With a unified API across multiple programming languages, including Scala, Java, Python, and R, DataFrames provide flexibility in application development. DataFrames in Spark are widely used for tasks such as data cleaning, exploration, analysis, and machine learning, making them a cornerstone of Spark's data processing capabilities.

Code:

```
from pyspark import SparkConf, SparkContext

conf=SparkConf().setMaster("local").setAppName("SpendByCustom
er")

sc = SparkContext(conf = conf)

def extractCustomerPricePairs(line):

fields = line.split(',')

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

input = sc.textFile("customer-orders.csv")

mappedInput = input.map(extractCustomerPricePairs)

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

results = totalByCustomer.collect();

for result in results:
```

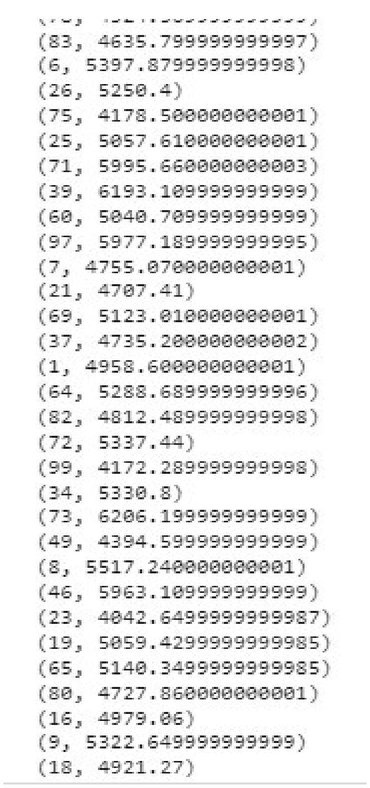
print(result)

Output:



```
[ ]  results = totalByCustomer.collect();
     for result in results:
       print(result)
```

```
     (44, 4756.8899999999985)
     (35, 5155.419999999999)
     (2, 5994.59)
     (47, 4316.299999999999)
     (29, 5032.529999999999)
     (91, 4642.259999999999)
     (70, 5368.249999999999)
     (85, 5503.43)
     (53, 4945.299999999999)
     (14, 4735.030000000001)
     (51, 4975.22)
     (42, 5696.840000000003)
     (79, 3790.570000000001)
     (50, 4517.27)
     (20, 4836.859999999999)
     (15, 5413.510000000001)
     (5, 4561.069999999999)
     (48, 4384.33)
     (31, 4765.05)
     (4, 4815.050000000002)
     (36, 4278.049999999997)
     (57, 4628.4)
     (12, 4664.589999999998)
     (22, 5019.449999999999)
     (54, 6065.389999999999)
     (0, 5524.949999999998)
     (88, 4830.549999999999)
     (86, 4908.81)
     (13, 4367.62)
     (40, 5186.429999999999)
```



```
(83, 4635.799999999997)
(6, 5397.879999999998)
(26, 5250.4)
(75, 4178.500000000001)
(25, 5057.610000000001)
(71, 5995.660000000003)
(39, 6193.109999999999)
(60, 5040.709999999999)
(97, 5977.189999999995)
(7, 4755.070000000001)
(21, 4707.41)
(69, 5123.010000000001)
(37, 4735.200000000002)
(1, 4958.600000000001)
(64, 5288.689999999996)
(82, 4812.489999999998)
(72, 5337.44)
(99, 4172.289999999998)
(34, 5330.8)
(73, 6206.199999999999)
(49, 4394.599999999999)
(8, 5517.240000000001)
(46, 5963.109999999999)
(23, 4042.6499999999987)
(19, 5059.4299999999985)
(65, 5140.3499999999985)
(80, 4727.860000000001)
(16, 4979.06)
(9, 5322.649999999999)
(18, 4921.27)
```

Practical 7

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

Theory:

Broadcast variables in Apache Spark are read-only variables distributed to each worker node in the cluster to improve task performance. They are useful for efficiently sharing large, static datasets or variables across all nodes, reducing data transfer overhead. Once broadcasted, the variable is cached in memory on each executor and reused across multiple tasks, enhancing efficiency. Broadcast variables are typically used in operations like joins or lookups, where the same data needs to be accessed by all tasks. By minimizing data shuffling and replication, broadcast variables optimize Spark's performance, especially for tasks involving large datasets or expensive operations.

Code:

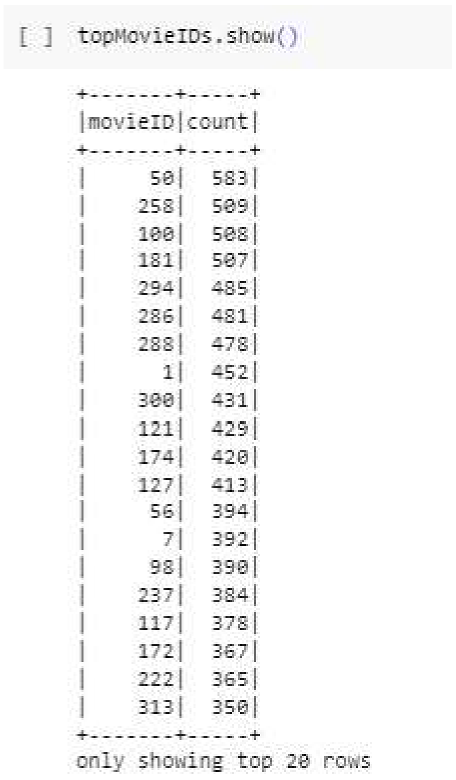
```
from pyspark.sql import SparkSession
from pyspark.sql import Row
from pyspark.sql import functions
def loadMovieNames():
movieNames = {}
with open("u.item",encoding = "ISO-8859-1") as f:
for line in f:
fields = line.split('|')
 movieNames[int(fields[0])] = fields[1]
return movieNames
# Create a SparkSession (the config bit is only for Windows!)
```

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

 # Load up our movie ID -> name dictionary nameDict =
loadMovieNames()

# Get the raw data

lines = spark.sparkContext.textFile("u.data")

# Convert it to a RDD of Row objects

movies = lines.map(lambda x: Row(movieID =int(x.split()[1])))

#Convert that to a DataFrame

movieDataset = spark.createDataFrame(movies)

# Some SQL-style magic to sort all movies by popularity in one line!
topMovieIDs =
movieDataset.groupBy("movieID").count().orderBy("count",
ascending=False).cache()

# Show the results at this point:
```

```
#|movieID|count|
#+-------+-----+
#|     50|  584|
#|    258|  509|
#|    100|  508|
```

```
topMovieIDs.show()

# Grab the top 10 top10 = topMovieIDs.take(10)

# Print the results

print("\n") for result in top10:

# Each row has movieID, count as above.

print("%s: %d" % (nameDict[result[0]], result[1]))

# Stop the session spark.stop()
```

Output:



```
[ ]  topMovieIDs.show()
```

```
+-------+-----+
|movieID|count|
+-------+-----+
|     50|  583|
|    258|  509|
|    100|  508|
|    181|  507|
|    294|  485|
|    286|  481|
|    288|  478|
|      1|  452|
|    300|  431|
|    121|  429|
|    174|  420|
|    127|  413|
|     56|  394|
|      7|  392|
|     98|  390|
|    237|  384|
|    117|  378|
|    172|  367|
|    222|  365|
|    313|  350|
+-------+-----+
only showing top 20 rows
```

Practical 8

Aim: Create Similar Movies from One Million Rating

Theory:

To create similar movies from a dataset of one million ratings using Spark:

1.Load the dataset and preprocess it.

2.Split the data into training and testing sets.

3.Train a recommendation model, such as ALS (Alternating Least Squares), using Spark's MLlib.

4.Use the trained model to generate similar movies for each movie in the dataset.

5.Evaluate the model's performance, typically using metrics like Root Mean Squared Error (RMSE).

6.Implement the recommendation system to provide similar movie suggestions based on user input or existing movie selections.

Code:

```
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('ratings1.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)
```

predictions = model.transform(test_data)

predictions.show()

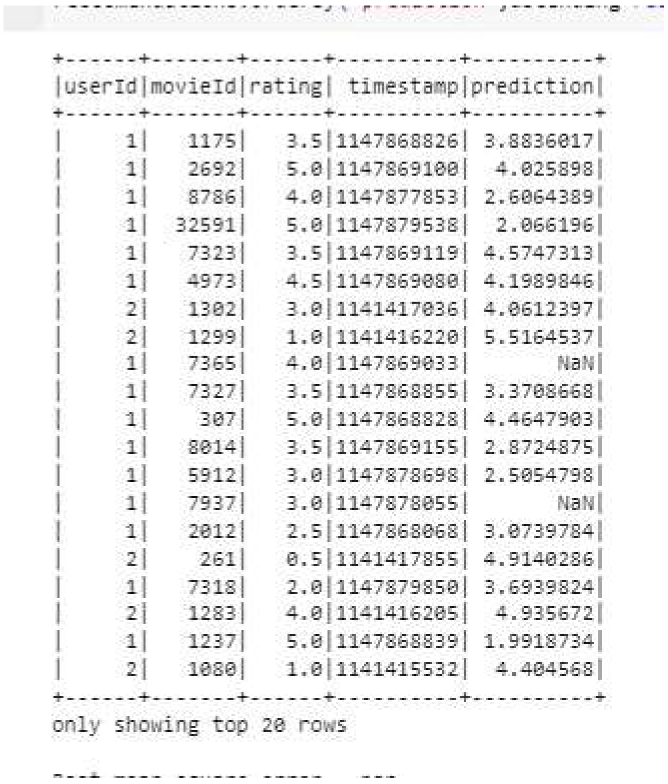
evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating",predictionCol="prediction") rmse = evaluator.evaluate(predictions)

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

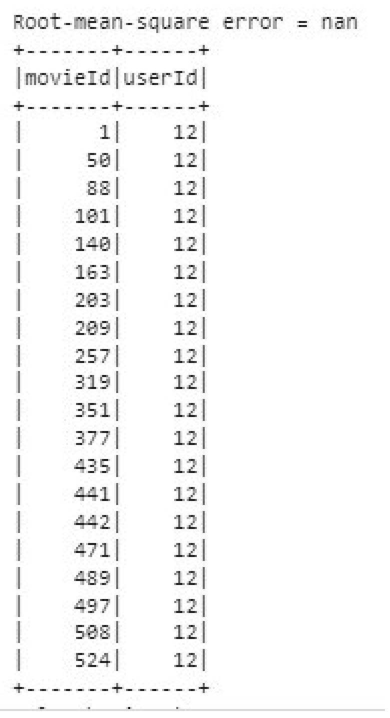
single_user = test_data.filter(test_data['userId']==12).select(['movieId','userId']) single_user.show()

reccomendations = model.transform(single_user) reccomendations.orderBy('prediction',ascending=False).show()

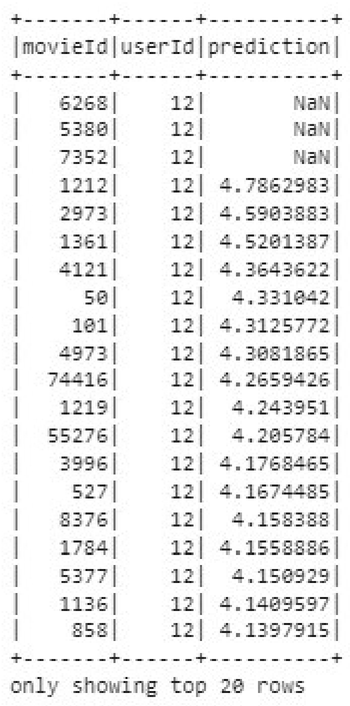
Output:



```
+------+-------+------+----------+----------+
|userId|movieId|rating| timestamp|prediction|
+------+-------+------+----------+----------+
|     1|   1175|   3.5|1147868826| 3.8836017|
|     1|   2692|   5.0|1147869100|  4.025898|
|     1|   8786|   4.0|1147877853| 2.6064389|
|     1|  32591|   5.0|1147879538|  2.066196|
|     1|   7323|   3.5|1147869119| 4.5747313|
|     1|   4973|   4.5|1147869080| 4.1989846|
|     2|   1302|   3.0|1141417036| 4.0612397|
|     2|   1299|   1.0|1141416220| 5.5164537|
|     1|   7365|   4.0|1147869033|       NaN|
|     1|   7327|   3.5|1147868855| 3.3708668|
|     1|    307|   5.0|1147868828| 4.4647903|
|     1|   8014|   3.5|1147869155| 2.8724875|
|     1|   5912|   3.0|1147878698| 2.5054798|
|     1|   7937|   3.0|1147878055|       NaN|
|     1|   2012|   2.5|1147868068| 3.0739784|
|     2|    261|   0.5|1141417855| 4.9140286|
|     1|   7318|   2.0|1147879850| 3.6939824|
|     2|   1283|   4.0|1141416205|  4.935672|
|     1|   1237|   5.0|1147868839| 1.9918734|
|     2|   1080|   1.0|1141415532|  4.404568|
+------+-------+------+----------+----------+
only showing top 20 rows
```



```
Root-mean-square error = nan
+-------+------+
|movieId|userId|
+-------+------+
|      1|    12|
|     50|    12|
|     88|    12|
|    101|    12|
|    140|    12|
|    163|    12|
|    203|    12|
|    209|    12|
|    257|    12|
|    319|    12|
|    351|    12|
|    377|    12|
|    435|    12|
|    441|    12|
|    442|    12|
|    471|    12|
|    489|    12|
|    497|    12|
|    508|    12|
|    524|    12|
+-------+------+
```



```
+-------+------+----------+
|movieId|userId|prediction|
+-------+------+----------+
|   6268|    12|       NaN|
|   5380|    12|       NaN|
|   7352|    12|       NaN|
|   1212|    12| 4.7862983|
|   2973|    12| 4.5903883|
|   1361|    12| 4.5201387|
|   4121|    12| 4.3643622|
|     50|    12|  4.331042|
|    101|    12| 4.3125772|
|   4973|    12| 4.3081865|
|  74416|    12| 4.2659426|
|   1219|    12|  4.243951|
|  55276|    12|  4.205784|
|   3996|    12| 4.1768465|
|    527|    12| 4.1674485|
|   8376|    12|  4.158388|
|   1784|    12| 4.1558886|
|   5377|    12|  4.150929|
|   1136|    12| 4.1409597|
|    858|    12| 4.1397915|
+-------+------+----------+
only showing top 20 rows
```

Practical 9

Aim: Using Spark ML to Produce Movie Recommendations

Theory:

Spark MLlib (Machine Learning Library) is a scalable machine learning library within Apache Spark that provides a wide range of algorithms and tools for various machine learning tasks. Here's some more detailed information about Spark MLlib:

1.Rich Set of Algorithms: Spark MLlib offers a comprehensive collection of machine learning algorithms and utilities for tasks such as classification, regression, clustering, collaborative filtering, dimensionality reduction, and feature engineering. It includes popular algorithms like logistic regression, decision trees, random forests, k-means clustering, and matrix factorization.

2.High-Level APIs: MLlib provides high-level APIs in Scala, Java, Python, and R, making it accessible to developers with different programming backgrounds. These APIs offer a user-friendly interface for building and deploying machine learning models, abstracting away the complexities of distributed computing.

3.Distributed Computing: MLlib leverages Spark's distributed computing framework to perform parallelized model training, evaluation, and prediction on large-scale datasets. It automatically parallelizes computations across multiple nodes in a Spark cluster, enabling efficient processing of big data.

4.Integration with Spark Ecosystem: MLlib seamlessly integrates with other components of the Spark ecosystem, such as Spark SQL, Spark Streaming, and Structured Streaming. This integration allows users to build end-to-end machine learning pipelines that span from data preprocessing and feature extraction to model training and inference.

5.Scalability and Performance: MLlib is designed for scalability and performance, enabling users to train and deploy machine learning models on datasets of any size. It optimizes distributed computations through techniques like data partitioning, in-memory caching, and pipelining, resulting in faster model training and prediction.

6.Model Persistence and Deployment: MLlib supports model persistence, allowing users to save trained models to disk in various formats (e.g., PMML, Parquet) for later use or deployment. Models can be deployed in production environments using Spark's deployment options, such as standalone mode, YARN, Mesos, or Kubernetes.

7.Community and Ecosystem: Spark MLlib benefits from a vibrant community of users and contributors who actively contribute to its development and improvement. It also has an extensive ecosystem of libraries and tools for tasks like hyperparameter tuning (e.g., MLflow), model serving (e.g., TensorFlow Serving), and model monitoring (e.g., Prometheus).

Practical 10

Aim: Use Windows with Structured Streaming to Track Most-Viewed URLs (Spark Streaming)

Theory:

Spark Streaming is a real-time data processing framework within Apache Spark that enables high-throughput, fault-tolerant stream processing of live data streams. Here's some more detailed information about Spark Streaming:

1.Micro-batch Processing: Spark Streaming adopts a micro-batch processing model where incoming data streams are divided into small, discrete batches of data. These batches are then processed using the same RDD (Resilient Distributed Dataset) abstraction as batch processing in Spark.

2.DStream API: Spark Streaming introduces a concept called Discretized Streams (DStreams), which represent a continuous stream of data divided into small RDDs. DStreams can be created from various input sources such as Kafka, Flume, Kinesis, or TCP sockets.

3.Windowed Operations: Spark Streaming provides support for windowed operations, allowing developers to perform computations over a sliding window of data. This enables analyzing data over specific time intervals or event counts, facilitating tasks like sessionization or trend analysis.

4.Exactly-once Semantics: Spark Streaming ensures exactly-once processing semantics through checkpointing and transactional updates. This guarantees that each record in the input stream is processed exactly once, even in the presence of failures or retries.

5.Integration with Spark Ecosystem: Spark Streaming seamlessly integrates with other components of the Spark ecosystem, including Spark SQL, MLlib, GraphX, and Structured Streaming. This allows developers to build end-to-end data processing pipelines that span from real-time stream processing to batch processing and machine learning.

6.Fault Tolerance and Scalability: Spark Streaming inherits the fault tolerance and scalability features of Apache Spark. It automatically handles failures by re-executing lost tasks on resilient worker nodes, ensuring continuous data processing even in the face of node failures or network issues.

7.Use Cases: Spark Streaming is used in a wide range of real-time analytics applications, such as monitoring system logs, processing sensor data from IoT devices, analyzing social media streams, detecting anomalies in financial transactions, and performing real-time recommendation systems.