**INDEX**

| Sr. No | Practical List Date Sign |
| --- | --- |
| 1. | Installation of Apache Spark.  24/09/2024 |
| 2. | Spark Basics and RDD interface 01/10/2024 |
| 3. | Filtering RDDs, and the Minimum  Temperature by Location Example 08/10/2024 |
| 4. | Counting Word Occurrences using flatmap() 15/10/2024 |
| 5.  6. | Executing SQL commands and SQL-style  functions on a Data Frame  22/10/2024  DataFrames19/11/2024  Implement Total Spent by Customer with |
| 7. | Use Broadcast Variables to Display Movie  Names Instead of ID Numbers 26/11/2024 |
| 8. | Create Similar Movies from One Million  Rating 03/12/2024 |
| 9. | Using Spark ML to Produce Movie  Recommendations10/12/2024 |
| 10. | Use Windows with Structured Streaming to  17/12/2024  Track Most-Viewed URLs (Spark  Streaming) |

**Practical-1**

**Aim: Installation of Apache Spark**

**Theory:**

**1. Introduction to Apache Spark**

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.

**2. Steps to Install Apache Spark on Windows**

Installing Apache Spark on Windows can be more complex compared to other operating systems like Linux or macOS due to 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:

***2.1 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.

***2.2 Download Apache Spark***

● Go to the Apache Spark website and download the latest version of Spark for Windows. ● Choose a pre-built package for **Hadoop** as it includes the necessary Hadoop binaries. ● After downloading, unzip the file to a location on your system (e.g., C:\spark).

***2.3 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. ***2.4 Configuration***

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

● Edit the spark-env.cmd file and set HADOOP\_HOME to your Hadoop directory (if applicable). ● Optionally, you can configure other Spark settings in this file as well.

***2.5 Starting Spark***

● Open a **Command Prompt**.

● Navigate to the Spark directory:

cd C:\spark

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

● You can also start Spark applications using bin\spark-submit.cmd. ***2.6 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:**

**1. 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.

**2. 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**: Create new RDDs from existing ones.

● **Actions**: 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**

**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]))



**Practical-4**

**Aim: Counting Word Occurrences using flatMap() 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)

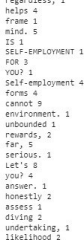
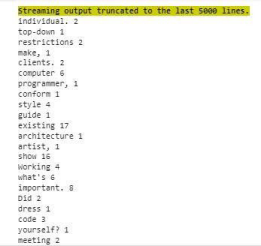
for word, count in wordCounts.items():

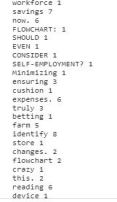
cleanWord = word.encode('ascii', 'ignore')

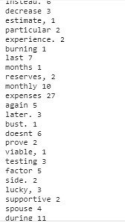
if cleanWord:

print(cleanWord.decode() + " " + str(count))

**Output:**

****

****

****

**Practical-5**

**Aim:Executing SQL commandsand SQL-style functionsona Data Frame. Code:**

from pyspark.sql import SparkSession, Row

import collections

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)

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

schemaPeople.createOrReplaceTempView("people")

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

for teen in teenagers.collect():

print(teen)

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

spark.stop()

**Output:**

****

**Practical-6**

**Aim:Implement Total Spent by Customer with Data Frames. Code:**

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("customer-orders.csv")

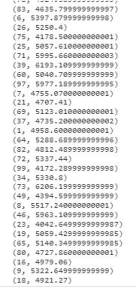
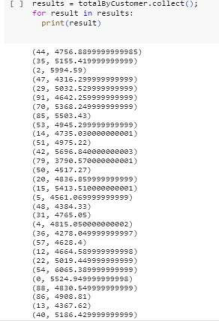
mappedInput = input.map(extractCustomerPricePairs)

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

for result in results:

print(result)

**Output:**

****

**Practical-7**

**Aim:UseBroadcastVariablestoDisplayMovieNamesInsteadofID Numbers**

**Code:**

from pyspark.sql import SparkSession, 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

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

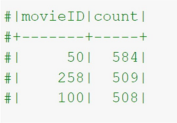
movieNames = loadMovieNames()

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

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

topMovieIDs = movieDataset.groupBy("movieID").count().orderBy("count", ascending=False).cache()

topMovieIDs.show()



topMovieIDs.show()

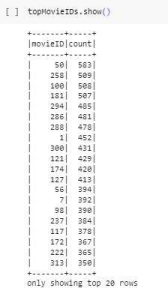
top10 = topMovieIDs.take(10)

print("\n")

for result in top10:

print("%s:%d" % (movieNames[result[0]], result[1])) spark.stop()

**Output:**

****

**Practical-8**

**Aim:Create Similar Movies from One Million Rating.**

**Code:**

from pyspark.sql import SparkSession

from pyspark.ml.evaluation import RegressionEvaluator

from pyspark.ml.recommendation import ALS

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

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)

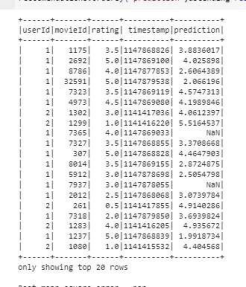
print("Root-mean-square error = " + str(rmse))

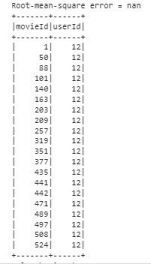
single\_user = test\_data.filter(test\_data['userId'] == 12).select(['movieId', 'userId']) single\_user.show()

recommendations = model.transform(single\_user)

recommendations.orderBy('prediction', ascending=False).show()

**Output:**

****

****

****

**Practical-9**

**Aim: Using Spark MLlib to Produce Movie Recommendations**

**Theory:**

**Spark MLlib Overview**

**Spark MLlib** is a scalable machine learning library within Apache Spark that provides a wide range of algorithms and tools for various machine learning tasks. Below is a detailed explanation of Spark MLlib:

**1. Rich Set of Algorithms**

● **Spark MLlib** offers a comprehensive collection of machine learning algorithms and utilities for tasks such as:

o **Classification**

o **Regression**

o **Clustering**

o **Collaborative filtering**

o **Dimensionality reduction**

o **Feature engineering**

● It includes popular algorithms like:

o **Logistic regression**

o **Decision trees**

o **Random forests**

o **K-means clustering**

o **Matrix factorization**

**2. High-Level APIs**

● MLlib provides **high-level APIs** in **Scala**, **Java**, **Python**, and **R**, making it accessible to developers from 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**, including: o **Spark SQL**

o **Spark Streaming**

o **Structured Streaming**

● This integration allows users to build **end-to-end machine learning pipelines**, spanning tasks 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:

o **Data partitioning**

o **In-memory caching**

o **Pipelining**

● These optimizations result 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:

o **Standalone mode**

o **YARN**

o **Mesos**

o **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:

o **Hyperparameter tuning** (e.g., **MLflow**)

o **Model serving** (e.g., **TensorFlow Serving**)

o **Model monitoring** (e.g., **Prometheus**)

**Practical-10**

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

**Theory:**

**Spark Streaming Overview**

**Spark Streaming** is a real-time data processing framework within Apache Spark that enables high-throughput, fault-tolerant stream processing of live data streams. Below is a detailed explanation of Spark Streaming:

**1. Micro-batch Processing**

● Spark Streaming uses a **micro-batch processing model**, where incoming data streams are divided into small, discrete batches.

● These batches are then processed using the same **RDD (Resilient Distributed Dataset)** abstraction as batch processing in Spark.

**2. DStream API**

● **DStreams (Discretized Streams)** represent a continuous stream of data divided into small RDDs.

● DStreams can be created from various input sources, including:

o **Kafka**

o **Flume**

o **Kinesis**

o **TCP sockets**

**3. Windowed Operations**

● Spark Streaming supports **windowed operations**, enabling computations over a sliding window of data.

● This feature allows developers to analyze data over specific time intervals or event counts, which is useful for tasks like:

o **Sessionization**

o **Trend analysis**

**4. Exactly-once Semantics**

● Spark Streaming ensures **exactly-once processing semantics** through:

o **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**, such as:

o **SparkSQL**

o **MLlib**

o **GraphX**

o **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 widely used in various real-time analytics applications, such as:

● **Monitoring system logs**

● **Processing sensor data from IoT devices**

● **Analyzing social media streams**

● **Detecting anomalies in financial transactions**

● **Performing real-time recommendation systems**