REVA UNIVERSITY
Bengaluru, India

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

BIG DATA AND ANALYTICS LAB
M22TC0207

Second Semester

AY-2024-25

(Prepared in Aug-2024)

Composed by: Dr. Shantala Devi Patil

INDEX

SL. No	Contents	Page. no
1	Lab Objectives	3
2	Lab Outcomes	3
3	Lab Requirements	3
4	Guidelines to Students	4
5	List of Lab Exercises and Mini-Project	4
6	Lab Exercises' Solutions:	7
PART – A: LAB EXERCISES		
	Session : Introduction: Installing PySpark on Colab	7
	Session – 1: Lab Exercise	13
	Session – 2: Lab Exercise	16
	Session – 3: Lab Exercise	18
	Session – 4: Lab Exercise	24
	Session – 5: Lab Exercise	25
	Session – 6: Lab Exercise	29
	Session – 7: Lab Exercise	33
	Session – 8: Lab Exercise	36
PART – B: MINI-PROJECT		
7	Project Report Format	41
8	Evaluation Rubrics	43
9	Learning Resources	48

1. Lab Objectives:

The objectives of this course are to

2. Discuss the fundamentals of Hadoop distributed file system and Big Data Analytics.
3. Demonstrate Big Data Processing with MapReduce and Batch Analytics.
4. Describe the implementation of Real-Time Analytics with Apache Hadoop in real world Applications.
5. Illustrate the working of Pig, Hive and Stream Processing and also discuss the fundamentals of Flume.

6. Lab Outcomes:

On successful completion of this course; student shall be able to:

CO#	Course Outcomes	POs	PSOs
CO1	Illustrate the fundamentals of Hadoop distributed file system and Big Data Analytics	1 to 5,9,10,11	1,2,3
CO2	Demonstrate Big Data Processing with MapReduce and Batch Analytics with Apache Hadoop to simple real world problems. .	1 to 5,9,10,11	1,2,3
CO3	Design Real-Time Analytics with Apache Pig and Hive for real world Applications.	1 to 5,9,10,11	1,2,3
CO4	Develop data and processing models using Hadoop eco-system for real world Big data Applications	1 to 5,9,10,11	1,2,3
CO5	Design Real-Time Analytics incorporating the structured data model using Apache Hive to solve real world Big Data Analytics Applications.	1 to 5,9,10,11	1,2,3
CO6	Develop data and processing models using Hadoop eco-system for real world Big data Applications	1 to 5,9,10,11	1,2,3

7. Lab Requirements:

The following are the required hardware and software for this lab.

Hardware Requirements: A standard personal computer or laptop with the following minimum specifications:

1. **Processor**: Any modern multi-core processor (e.g., Intel Core i5 or AMD Ryzen series).
2. **RAM**: At least 4 GB of RAM for basic programs; 8 GB or more is recommended for larger data structures and complex algorithms.
3. **Storage**: A few gigabytes of free disk space for the development environment and program files.
4. **Display**: A monitor with a resolution of 1024x768 or higher is recommended for comfortable coding.
5. **Input Devices**: Keyboard and mouse (or other input devices) for coding and interacting with the development environment.

Software Requirements:

1. **Operating System**: You can develop and execute programs for Big Data Analytics Lab on various operating systems, including Windows, macOS, and Linux. In our lab we are using **Ubuntu operating system.**
2. **Spark:** Apache Spark is an open-source, distributed processing system used for big data workloads. It utilizes in-memory caching, and optimized query execution for fast analytic queries against data of any size.
3. **Pyspark:** PySpark is the Python API for Apache Spark. It enables you to perform real-time, large-scale data processing in a distributed environment using Python. It also provides a PySpark shell for interactively analyzing your data.

8. Guidelines to Students:

- Equipment in the lab for the use of the student community. Students need to maintain a proper decorum in the computer lab. Students must use the equipment with care. Any damage caused is punishable.
- Students are required to carry their observation / programs book with completed exercises while entering the lab.
- Students are supposed to occupy the machines allotted to them and are not supposed to talk or make noise in the lab. The allocation is put up on the lab notice board.
- The lab can be used in free time / lunch hours by the students who need to use the systems should get prior permission from the lab in-charge.
- Lab records need to be submitted on or before the date of submission.
- Students are not supposed to use flash drives.

9. List of Lab Exercises and Mini-Project:

Sl No.	Title of the Exercise
	Part A
	Introduction: Installing PySpark on Colab
1	Create a Python script to test PySpark. a. Create a SparkContext object b. Create an RDD from for the input file “Sample.txt” c. Find the total count of the words in the RDD Print the string with highest occurrence**.**
2	Given the two RDDs: a. x created from the ordered pairs: ("spark", 1) and ("hadoop", 4) b. y created from the ordered pairs: ("spark", 2), ("hadoop", 5). c. Perform the join operation on the RDDs created above, and print the resulting RDD. Run the Usecases for Right Join, Left Join, Inner Join, Outer Join
3	a.Create an RDD of set of numbers and perform the sum of these numbers using an *accumulator()* function in Spark context. b.Create an RDD from the existing file having CSV data, using *read()* and *load()* functions and display the top 5 rows of the data set. And also display the statistical results from the data frame (Note: It only works for numerical values).
4	Given the following tables, Perform Full outer join on dataframe **Age** | **Name** **2** | **Alice** **5** | **Bob**

Height	Name
80	Tom
85	Bob

Age	Name
2	Alice
5	Bob

Age	Height	Name
10	80	Alice
5	None	Bob
None	None	Tom
None	None	None

a. Write query and Perform Join of df1 and df2 to find the height
b. Write query and Perform outer join of df1 and df2 to find the heights
c. Write query and Perform outer join of df1 and df2 to find the age

5 Given the following data
data = [("1", "john jones"), ("2", "tracey smith"), ("3", " amy sanders")], along with the following schema of the data
columns = ["Seqno","Name"]
Perform the following using the afore mentioned data.
i. create an RDD from the above data using its schema
ii. create the PySparkdataframe from the RDD created.
iii. Write python functions to convert the first letter of every string into upper case.
iv. Use the above python function as udf in pySpark to convert the data in the dataframe and display the result.

6 Given the following data.
Data = [("James","Sales","NY",90000,34,10000),
("Michael","Sales","NV",86000,56,20000),
("Robert","Sales","CA",81000,30,23000),
("Maria","Finance","CA",90000,24,23000),
("Raman","Finance","DE",99000,40,24000),
("Scott","Finance","NY",83000,36,19000),
("Jen","Finance","NY",79000,53,15000),
("Jeff","Marketing","NV",80000,25,18000),
("Kumar","Marketing","NJ",91000,50,21000)
], with the following schema schema =
["employee_name","department","state","salary","age","bonus"]
Perform the following using the aforementioned data.
i. create an RDD from the above data using its schema
ii. create the PySparkdataframe from the RDD created.
iii. Using groupBy() function, display the salaries of the employees state-wise.
iv. Display the state-wise salaries that are greater than 1 lakh
V. Display the state-wise salaries in descending order.

7 Create a dataframe and then perform the following operations
i. Check the lifestage of each person into Adult, Child and Teenager
ii. Write query to Display entries of teenager and adult only
iii. Write query to average age
iv. Write query to group by entries by life_stage
v. Insert a record “Frank,4, Child) into new data frame
vi. Write a query to display teenage entries

8 Write a Word Count Map Reduce program to understand Map Reduce Paradigm.

	Part B
1	Implement and demonstrate any real life big data problem using any of the publicly available big data sets.

PART – A

LAB EXERCISES

10. Lab Exercises' Solutions:

Solutions (Part A)

Introduction: Installing PySpark on Colab

Introduction to Apache Spark

Apache Spark is an open-source, distributed computing system that provides an easy-to-use platform for large-scale data processing. Spark is designed to be fast, general-purpose, and easy to use, making it one of the most popular frameworks for big data analytics.

Key Features of Apache Spark:

- **Speed:** Spark processes data in-memory, which makes it much faster than traditional MapReduce, especially for iterative algorithms.
- **Ease of Use:** Spark has simple APIs for operating on large datasets, in multiple languages such as Python, Java, Scala, and R.
- **General-purpose:** Spark supports various types of data processing, including batch processing, real-time stream processing, and interactive queries.
- **Unified Engine:** It provides a unified engine for processing large datasets, supporting SQL, streaming data, machine learning, and graph processing.

What is RDD?

A Resilient Distributed Dataset (RDD) is the most fundamental data object used in Spark programming.

RDDs are datasets within a Spark application, including the initial dataset(s) loaded, any intermediate dataset(s), and the final resultant dataset(s).

Most Spark applications load an RDD with external data and then create new RDDs by performing operations on the existing RDDs; these operations are transformations.

This process is repeated until an output operation is ultimately required—for instance, to write the results of an application to a filesystem; these types of operations are actions

The term Resilient Distributed Dataset is an accurate and succinct descriptor for the concept. Here's how it breaks down:

- **Resilient:** RDDs are resilient, meaning that if a node performing an operation in Spark is lost, the dataset can be reconstructed.
- **Distributed:** RDDs are distributed, meaning the data in RDDs is divided into one or many partitions and distributed as in-memory collections of objects across Worker nodes in the cluster.
- **Dataset:** RDDs are datasets that consist of records. Records are uniquely identifiable data collections within a dataset. A record can be a collection of fields similar to a row in a table in a relational database, a line of text in a file, or multiple other formats

Support of Different Languages in Apache Spark

Apache Spark provides APIs in multiple programming languages, which allows developers and data scientists to use the tools they are most comfortable with:

- **Scala:** The original language in which Spark was written. It is preferred for its tight integration with the Spark core.
- **Java:** Spark provides full support for Java, making it accessible to a wide range of developers.
- **Python (PySpark):** PySpark is the Python API for Spark. It is widely used in data science and machine learning.
- **R:** SparkR is the R API for Spark, used primarily for statistical computing and graphics.

Introduction to PySpark:

PySpark is the Python API for Apache Spark, an open-source distributed computing system that provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. PySpark allows Python developers to interface with Spark's features, making it easier to leverage Spark's power for big data processing tasks. Familiarity, Rich Ecosystem and Scalability are the key benefits of PySpark. Here are some key PySpark methods and functions that are essential for working with data in Spark.

1. SparkSession

A SparkSession is the entry point to any PySpark functionality. It allows you to create DataFrame and execute SQL queries.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("PySpark Introduction").getOrCreate()
```

RDD Transformations

Transformations create a new RDD from an existing one. They are lazy, meaning they do not execute until an action is called.

1. Map
The map method applies a function to each element of the RDD.

```
rdd = sc.parallelize([1, 2, 3, 4])
squared_rdd = rdd.map(lambda x: x * x)
print(squared_rdd.collect())
```

2. Filter
The filter method filters the elements of the RDD based on a predicate function.

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
even_rdd = rdd.filter(lambda x: x % 2 == 0)
print(even_rdd.collect())
```

3. FlatMap
The flatMap method applies a function that returns a sequence for each element in the RDD and flattens the results.

```
rdd = sc.parallelize(["hello world", "hi"])
words_rdd = rdd.flatMap(lambda x: x.split(" "))
print(words_rdd.collect())
```

4. Union

The union method returns the union of two RDDs.

```
rdd1 = sc.parallelize([1, 2, 3])
rdd2 = sc.parallelize([3, 4, 5])
union_rdd = rdd1.union(rdd2)
print(union_rdd.collect())
```

5. Intersection

The intersection method returns the intersection of two RDDs.

```
rdd1 = sc.parallelize([1, 2, 3])
rdd2 = sc.parallelize([3, 4, 5])
intersection_rdd = rdd1.intersection(rdd2)
print(intersection_rdd.collect())
```

RDD Actions

Actions trigger the execution of RDD transformations.

1. Collect

The collect method returns all the elements of the RDD as a list.

```
rdd = sc.parallelize([1, 2, 3, 4])
print(rdd.collect())
```

2. Count

The count method returns the number of elements in the RDD.

```
rdd = sc.parallelize([1, 2, 3, 4])
print(rdd.count())
```

3. Take

The take method returns the first n elements of the RDD.

```
rdd = sc.parallelize([1, 2, 3, 4])
print(rdd.take(2))
```

4. Reduce

The reduce method reduces the elements of the RDD using a specified function.

```
rdd = sc.parallelize([1, 2, 3, 4])
sum = rdd.reduce(lambda a, b: a + b)
print(sum)
```

2. SparkContext

SparkContext is the entry point to any Spark functionality. It allows your Spark application to connect to the Spark cluster. Before you can create DataFrames or RDDs, you need to create a SparkContext.

Creating SparkContext

In PySpark, a SparkContext is automatically created when you create a SparkSession. However, if you need to manually create a SparkContext, you can do it as follows:

```
from pyspark import SparkContext

# create a Spark context

sc = SparkContext.getOrCreate()
```

Basic SparkContext Operations

Parallelize

The parallelize method is used to create an RDD from a Python list. This is useful for testing and experimenting with RDD operations.

```
data = [1, 2, 3, 4, 5]

rdd = sc.parallelize(data)
```

TextFile

The textFile method is used to create an RDD from a text file. Each line in the text file becomes an element in the RDD.

```
rdd = sc.textFile("path/to/textfile.txt")
```

WholeTextFiles

The wholeTextFiles method is used to read multiple whole text files into an RDD, where each file is represented as a tuple (filename, content).

```
rdd = sc.wholeTextFiles("path/to/directory")
```

Introduction to Installation of pysprak

1. Introduction to Google Colab

Google Colab is a free, cloud-based platform that provides a Jupyter notebook environment for running Python code. It is widely used for data science, machine learning, and deep learning tasks.

Use the link https://colab.research.google.com/ and create a new notebook.

```
pip install pyspark
```

Create a SparkSession: Import SparkSession from pyspark.sql and create a session, the entry point to your PySpark journey:

That's it! You're now ready to explore PySpark functionalities on Colab.

2. Installation on Ubuntu

Step 1: Update and Upgrade the System

First, update and upgrade your system packages to ensure you have the latest versions.

```
sudo apt update
sudo apt upgrade –y
```

Step 2: Install Java

Apache Spark requires Java to run. Install OpenJDK (Java Development Kit).

```
sudo apt install -y openjdk-11-jdk
```

Verify the installation:

```
java –version
```

Step 3: Install Hadoop

Although Spark can run without Hadoop, it's often useful to install Hadoop for HDFS (Hadoop Distributed File System).

Download Hadoop: wget https://downloads.apache.org/hadoop/common/hadoop-3.3.5/hadoop-3.3.5.tar.gz

Extract the tar file:

```
tar -xzf hadoop-3.3.5.tar.gz
```

Move Hadoop to a suitable directory:

```
sudo mv hadoop-3.3.5 /usr/local/hadoop
```

Set up Hadoop environment variables by editing the ~/.bashrc file:

```
nano ~/.bashrc
```

Add the following lines at the end of the file:

```
export HADOOP_HOME=/usr/local/hadoop
export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
```

Source the .bashrc file to apply changes:

```
source ~/.bashrc
```

Step 4: Install Spark

Download Apache Spark:

wget https://downloads.apache.org/spark/spark-3.5.0/spark-3.5.0-bin-hadoop3.tgz

Extract the tar file:

```
tar -xzf spark-3.5.0-bin-hadoop3.tgz
```

Move Spark to a suitable directory:

```
sudo mv spark-3.5.0-bin-hadoop3 /usr/local/spark
```

Set up Spark environment variables by editing the ~/.bashrc file:

```
nano ~/.bashrc
```

Add the following lines at the end of the file:

```
export SPARK_HOME=/usr/local/spark
export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
```

Source the .bashrc file to apply changes:

```
source ~/.bashrc
```

Step 5: Install PySpark

Install PySpark using pip:

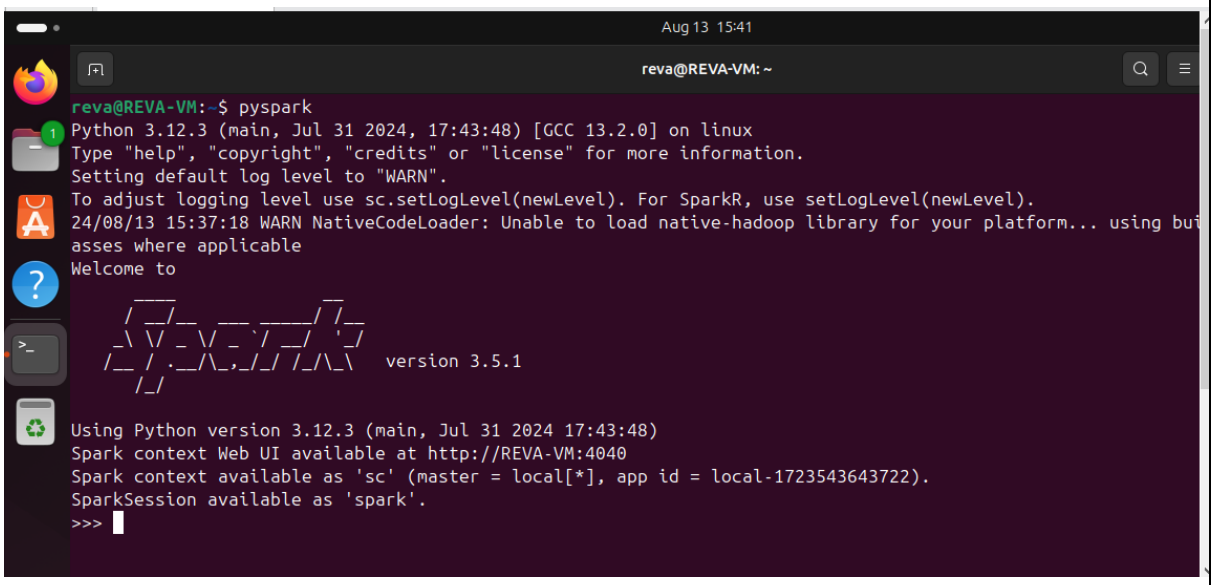
```
pip install pyspark
```

Step 6: Verify the Installation

To verify that PySpark is correctly installed, run a simple PySpark shell command:

```
pyspark
```

You should see an interactive PySpark shell, indicating that PySpark is correctly installed.



Step 7: Run a Simple PySpark Application

Create a Python script to test PySpark.

1 Create a Python script to test PySpark.

a. **Create a SparkContext object**
b. **Create an RDD from for the input file "Sample.txt"**
c. **Find the total count of the words in the RDD**
d. **Print the string with highest occurrence.**

Problem Statement:
The task is to perform a series of operations using Apache Spark to manipulate and analyze a set of words.

Solution Overview:

a) Initialize the SparkContext Object:
   - The SparkContext is the main gateway for any Spark application. It provides the interface to interact with the Spark cluster and is used to create RDDs.

b) Read the file "Sample.txt" from a List of Words:

c) Calculate the Total Number of Words using Lambda Function:
   - The count() action computes and returns the total number of elements in Lambda function.

d) Filter and Display Strings That Include the Word with highest occurrence
   - The filter() transformation is used to extract strings containing the word "spark," and the collect() action retrieves these filtered results to display them.

Intuition:

a) Creating a SparkContext Object:
   - Intuition: The SparkContext serves as the main interface for interacting with Spark, allowing you to leverage its distributed computing capabilities.

b) Creating an RDD from a Set of Words:
   - Intuition: Enable Spark to process each item concurrently, which enhances processing efficiency.

c) Finding the Total Count of Words in the RDD:
   - Intuition: Spark can easily aggregate data across nodes, allowing you to determine the total number of elements in the distributed dataset.

e) Filtering and Printing Strings That Include the Word with highest occurrence:
   - Intuition: Spark efficiently filters the dataset, extracting and returning only those elements that meet specific criteria, such as containing the word "spark."

Code Implementation:

Step 1: Setting Up PySpark Environment

```
pip install pyspark
```

Step 2: Create a SparkContext Object

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

Step 3: Create a RDD called by using sparkcontext object with parallelize method

```
text_file = sc.textFile("sample2 (1).txt")

counts = text_file.flatMap(lambda line: line.split(" ")) \

.map(lambda word: (word, 1)) \

.reduceByKey(lambda x, y: x + y)
```

Step 4: Find the total count of words in the RDD using Count( ) method

```
total_count = text_file.count()
print(total_count)
```

Step 5: Filter the RDD to Create spark_words_rdd, Including Only Elements Containing the Substring "spark"

```
spark_words_rdd = rdd.filter(lambda x: "spark" in x)
```

Complete Code without Explanation:

```
# To find out path where pyspark installed
import findspark
findspark.init()


# Create SparkSession and sparkcontext
from pyspark.sql import SparkSession
spark = SparkSession.builder\
            .master("local")\
            .appName('Firstprogram')\
            .getOrCreate()
sc=spark.sparkContext
# Read the input file and Calculating words count
text_file = sc.textFile("sample2 (1).txt")
counts = text_file.flatMap(lambda line: line.split(" ")) \
                .map(lambda word: (word, 1)) \
               .reduceByKey(lambda x, y: x + y)


# Printing each word with its respective count
output = counts.collect()
for (word, count) in output:
   print("%s: %i" % (word, count))

sc.stop()
spark.stop()
```

Sample Output:

Big: 4
Data: 5
analysis: 2
involves: 1
examining: 1
and: 12
processing: 3

Outcome of the Exercise:

After working with this code snippet, the learner will be able to:

- Create and handle RDDs: Understand how to create and manage RDDs in Spark.
- Count elements: Use Spark operations to determine the total number of items in an RDD.
- Filter and collect data: Apply filters to extract specific data based on criteria and gather the results.

This will enhance students practical understanding of distributed data processing with Spark.

Viva/Interview Questions:

1. What is the purpose of a SparkContext object in a Spark application?

Answer:The SparkContext object is the entry point to any Spark application. It allows your program to connect to a Spark cluster, either locally or in a distributed environment. Through the SparkContext, you can create RDDs, broadcast variables, and access Spark's core functionalities. Essentially, it sets up the environment in which your Spark job will run.

2. How do you create a SparkContext object in a Spark application?

Answer:You create a SparkContext object by first setting up a SparkConf object, which contains configurations for your Spark application like the application name and master URL. Then, you pass this SparkConf object to the SparkContext constructor. Here's a simple example in Python:

```
from pyspark import SparkConf, SparkContext

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

sc = SparkContext(conf=conf)
```

3. What does the parallelize() function do in Spark?

Answer:The parallelize() function in Spark is used to create an RDD from a list or any collection of objects. This function distributes the data across the nodes in a cluster, enabling parallel processing. For example, if you have a list of strings and you want to distribute this data for parallel operations, you would use parallelize().

4. How do you create an RDD from a list of words using the parallelize() function?

Answer:To create an RDD from a list of words, you use the parallelize() function provided by the SparkContext. Here's how you can do it:

python

Copy code

```
words = ["scala", "java", "hadoop", "spark", "akka", "spark vs hadoop", "pyspark", "pyspark and spark"]

rdd = sc.parallelize(words)
```

5. How do you find the total number of elements in an RDD?

Answer:You can find the total number of elements in an RDD by using the count() action. This function counts and returns the number of elements in the RDD.

Example:

```
total_count = rdd.count()
```

If the RDD contains 8 elements, total_count will be 8.

6. How would you filter strings containing the word "spark" from an RDD?

Answer:To filter strings containing the word "spark" from an RDD, you can use the filter() transformation. The filter() function takes a lambda function as a condition and returns a new RDD containing only the elements that meet this condition.

Example:

```
spark_rdd = rdd.filter(lambda x: "spark" in x)
```

This will return an RDD containing only the strings that have the substring "spark".

7. What is the output of the filter() operation when applied to the RDD containing the list of words?

Answer:The filter() operation will return an RDD with the following strings:

- "spark"
- "spark vs hadoop"
- "pyspark"
- "pyspark and spark"

These are the strings in the original RDD that contain the word "spark".

8. What is lazy evaluation in Spark, and how does it affect the execution of transformations like filter()?

Answer:Lazy evaluation means that Spark does not immediately execute transformations when they are called. Instead, it builds up a logical plan (DAG) of transformations, which is only executed when an action (like count() or collect()) is called. This optimization allows Spark to combine transformations and reduce the amount of data shuffled across the cluster, improving efficiency.

9. What is the difference between the count() and collect() actions in Spark?

Answer:

- count(): This action returns the number of elements in the RDD.
- collect(): This action retrieves the entire dataset from the RDD and brings it to the driver program as a list. While count() only returns a number, collect() can be expensive for large datasets as it pulls all data to the driver.

10. What happens if you don't apply an action after performing transformations on an RDD?

Answer:If you don't apply an action after performing transformations, the transformations are not executed. This is due to Spark's lazy evaluation. Without an action to trigger the execution, Spark will simply keep building up the execution plan without actually processing the data. This means no results will be returned or computed until an action is called.

Additional Questions

1. What does the SparkContext object do in Spark?
2. What does the filter() transformation do in Spark?
3. What is the parallelize() function used for in Spark?
4. How can you determine the number of elements in an RDD?
5. What is Apache Spark, and how does it differ from Hadoop?
6. Can you explain the role of the SparkContext object in a Spark application?
7. What are RDDs in Spark, and why are they important?
8. What is the difference between actions and transformations in Spark?
9. How do you create a SparkContext object, and what are the necessary configurations?
10. What does the parallelize () function do in Spark, and when would you use it?
11. How would you calculate the total number of elements in an RDD, and which function is used for this purpose?
12. How can you filter elements in an RDD based on a condition, and what function is used to achieve this?
13. What is the significance of lazy evaluation in Spark, particularly with transformations like filter ()?
14. In the context of the given problem, what would happen if you didn't apply an action after filtering the RDD?
15. How does Spark handle case sensitivity when filtering strings in an RDD?
16. What are some other common transformations in Spark, besides filter(), that can be applied to RDDs?

2

Given the two RDDs:
a. x created from the ordered pairs: ("spark", 1) and ("hadoop", 4)
a. b. y created from the ordered pairs: ("spark", 2), ("hadoop", 5).

b. Perform the join operation on the RDDs created above, and print the resulting RDD.
c. Run the Usecases for Right Join, Left Join, Inner Join, Outer Join

Problem Statement:

The task is to perform the join operation on RDDs and to print the resultant RDD.

Solution Overview:

- Spark: Apache Spark is an open-source Distribution Processing System used for bigdata workloads
- RDD (Resilient Distributed Datasets) in Spark is a fundamental data structure that represents an immutable, fault-tolerant collection of data that can be processed in parallel across a cluster of machines.
- RDDs can be created from various data sources, including Hadoop Distributed File System (HDFS), local file systems, and external databases, and can be transformed using various operations such as map, filter, and reduce to perform complex computations.
- RDDs are the basic building blocks of Spark applications and provide a high-level API for performing distributed data processing.
- Join is a transformation, and it is available in the package **org.apache.spark.rdd.pairRDDFunction**

Intuition:

To solve the problem, make the environment ready by installing spark and Pyspark. Create context and session for spark. Create RDD and perform join operation.

Code Implementation:

```
# Install Spark
 pip install spark

# Install Pyspark
 Pip install pyspark

# Create Context and Session for Spark
from pyspark.context import SparkContext
from pyspark.sql.session import SparkSession
sc=SparkContext.getOrCreate()
spark=SparkSession(sc)
#Using parallelize method, create RDD
rdd1=sc.parallelize([("spark", 1),("hadoop", 4)])
rdd2=sc.parallelize([("spark", 2),("hadoop", 5)])

#Perforn Join Operation on the created RDDs
rdd=sorted(rdd1.join(rdd2).collect())

#Print the Result
Print(rdd)
```

Sample Output:

[('hadoop', (4, 5)), ('spark', (1, 2))]

Usecases:

```
 rdd1=sc.parallelize([("spark", 1),("hadoop", 4)])
rdd2=sc.parallelize([("spark", 2),("hadoop", 5)])
print(sorted(rdd1.join(rdd2).collect()))
```

Output

[('hadoop', (4, 5)), ('spark', (1, 2))]

Left Outer Join

```
rdd1=sc.parallelize([("spark", 1),("hadoop", 4)])
```

```
rdd2=sc.parallelize([("spark", 2)])
sorted(rdd1.leftOuterJoin(rdd2).collect())
```

Output

```
[('hadoop', (4, None)), ('spark', (1, 2))]
```

```
rdd1=sc.parallelize([("spark", 1),("hadoop", 4)])
rdd2=sc.parallelize([("spark", 2)])
sorted(rdd2.rightOuterJoin(rdd1).collect())
```

Output

```
[('hadoop', (None, 4)), ('spark', (2, 1))]
```

```
rdd1=sc.parallelize([("spark", 1),("hadoop", 4)])
rdd2=sc.parallelize([("spark", 2),("hadoop", 5)])
sorted(rdd1.fullOuterJoin(rdd2).collect())
```

Output

```
[('hadoop', (4, 5)), ('spark', (1, 2))]
```

Outcome of the Exercise:

After the learning this code snippet, the learner will able to understand about the concept of RDD, How to use Join Operation in Spark context in RDD.

Interview Questions:

1. List all Spark RDD Join Types
2. Explain Inner join
3. Explain Left outer join
4. Explain Right outer join
5. Explain cartesian join

3

a. Create an RDD of set of numbers and perform the sum of these numbers using an *accumulator()* function in Spark context.

b.Create an RDD from the existing file having CSV data, using *read()* and *load()* functions and display the top 5 rows of the data set. And also display the statistical results from the data frame (Note: It only works for numerical values).

a. Problem Statement:

The problem is to create an RDD on set of numbers and perform sum using accumulator() in spark context.

Solution Overview:

When dealing with large-scale data processing in Apache Spark, an accumulator can be useful for tracking aggregated information, such as sums, across distributed data. Here's an overview of how to create an RDD, sum the numbers using an accumulator, and ensure the operation is efficiently and correctly executed.

1. Spark Context Initialization: Set up the environment for Spark operations.

Create a SparkContext object, which is the entry point for Spark functionality.

- The local argument specifies that Spark runs locally with one thread. The second argument is the name of the application.

2. Creating an RDD

- Create a resilient distributed dataset (RDD) from a collection of numbers.
- Use the parallelize() method on the Spark context to distribute the dataset across the cluster.

3. Accumulator Initialization

- Accumulate the sum of the numbers in the RDD.
- Create an accumulator using the accumulator() function with an initial value of 0.
- The accumulator will be used to add values as the RDD is processed.

4. Applying the Accumulator to the RDD

- Sum each element of the RDD by adding it to the accumulator.
- Define a function that adds each element of the RDD to the accumulator, and then apply this function to the RDD using foreach.
- The foreach action applies the function to each element in the RDD, adding the numbers to the accumulator.

5. Retrieving the Result

- Retrieve the final sum after all numbers in the RDD have been processed.
- Access the value property of the accumulator.
- This prints the total sum of the numbers in the RDD.

6. Stopping the Spark Context

- Clean up the Spark environment after the operation is complete.
- Use the stop() method on the SparkContext object.

Key Concepts

- **RDD (Resilient Distributed Dataset)**: A fault-tolerant collection of elements that can be operated on in parallel.
- **Accumulator**: A variable that workers can only add to using an associative operation. It is used to implement counters or sums across the cluster.
- **foreach Action**: An action that applies a function to each element in an RDD.

Intuition:

When working with large datasets in a distributed computing environment like Apache Spark, you often need to aggregate data spread across multiple nodes. Summing values across these nodes while ensuring efficiency and fault tolerance can be challenging. This is where accumulators come into play.

Understanding the Problem:

You have a collection of numbers, and you want to compute their sum. In a single-machine environment, this is a straightforward operation, but in a distributed system like Spark, the data is spread across multiple nodes. The challenge is to aggregate these distributed numbers efficiently and reliably.

Why Use an Accumulator?

- **Parallel Computation**: In Spark, data is processed in parallel across multiple nodes. If you try to sum numbers directly using operations like reduce, each node will compute part of the sum independently. However, if you want a running total that can be accessed across different parts of your program, you need something more versatile.
- **Global State Management**: Accumulators allow you to maintain a global state (like a sum) that can be safely updated by multiple nodes during parallel operations. This is particularly useful for operations where you need to track progress or aggregate results from across the cluster.

How Accumulators Work:

- An accumulator is a special variable in Spark that workers can increment (or perform other associative operations on) as they process data.
- Unlike other variables, the updates to an accumulator are automatically aggregated back to the driver program, even when tasks are distributed across different nodes.
- This makes accumulators ideal for counting events (like errors) or summing values across a large dataset.

The Process:

1. **Initialize the Spark Context**: This sets up the environment where Spark operations can be executed.
2. **Create an RDD**: This RDD represents the distributed dataset, where each element (number) will be processed across the nodes.
3. **Initialize an Accumulator**: The accumulator starts with an initial value (e.g., 0 for sum) and will hold the running total as the RDD is processed.
4. **Process the RDD**: Use an action like foreach to iterate through the RDD, updating the accumulator with each number.
5. **Retrieve the Accumulated Value**: After all elements have been processed, the total sum can be accessed via the accumulator.

Code Implementation:

```
from pyspark import SparkContext
# create a Spark context
  sc = SparkContext()
```

```
# create an RDD of set of numbers
    rdd = sc.parallelize([{1, 2, 3}, {4, 5, 6}, {7, 8, 9}])
# define an accumulator
    acc = sc.accumulator(0)
# use the accumulator to sum the numbers in the RDD
    def add_to_acc(x):
    global acc
    acc += sum(x)
     rdd.foreach(add_to_acc)
# print the result
     print("Sum of numbers in RDD: ", acc.value)
# stop the Spark context
      sc.stop()
```

Sample output:

Sum of numbers in RDD: 45

Outcome of the Exercise:
After the learning this code snippet, the learner will able to understand about the concept of RDD and find the Sum of the Numbers using accumulator() function in Spark context in RDD.

Interview Questions:

1. What is an accumulator in Spark, and how does it differ from broadcast variables?
   An accumulator is a variable that workers can "add" to using an associative operation, primarily used for counters or sums. Unlike broadcast variables, which are read-only and shared across tasks, accumulators are used to aggregate information across tasks.
2. Why might one prefer using an accumulator over traditional RDD operations for summing values?
   Accumulators are preferred over traditional RDD operations for summing values because they provide a simple and efficient way to perform distributed aggregation without needing to shuffle data or modify the RDD.
3. Can accumulators be used to accumulate non-numeric data?
   Yes, accumulators can be used to accumulate non-numeric data, but they are most commonly used for numeric aggregations. Custom accumulator types can be created for other data types.
4. What are some potential pitfalls when using accumulators in Spark?
   Potential pitfalls include:
   Accidental Updates: Multiple updates from the same task may lead to incorrect results.
   Driver Visibility: Accumulator updates are only visible to the driver after the action is executed, not during lazy evaluation.
5. How does Spark ensure the reliability of accumulator updates, especially in the face of task failures and retries?
   Spark ensures reliability by tracking task attempts. If a task fails and is retried, the accumulator's update from the failed task is not counted again, ensuring accurate results.
6. What is SparkContext, and why is it important?
   SparkContext is the entry point for any Spark application. It initializes the core components and connects to the cluster manager to distribute and manage tasks. It's crucial because it allows Spark to interact with the cluster and access resources.
7. How do you create a SparkContext in a Spark application?
   In most modern Spark applications, you typically create a SparkContext indirectly by using the SparkSession.builder.getOrCreate() method, which internally initializes a SparkContext. In older versions, you would directly create a SparkContext by passing a SparkConf object to it.
8. Can you have multiple SparkContexts in a single Spark application?
   No, you cannot have multiple active SparkContexts in a single Spark application. However, you can stop the existing SparkContext and create a new one if needed.
9. What happens if you don't stop a SparkContext?
   If you don’t stop a SparkContext, the resources allocated to the application remain occupied, potentially leading to resource leaks and preventing the Spark application from shutting down properly.
10. How does SparkContext connect with the cluster manager?

SparkContext connects with the cluster manager (like YARN, Mesos, or Kubernetes) using the cluster manager's URL provided in the Spark configuration. The cluster manager then allocates resources (executors) to the Spark application.

b. Problem Statement:

The task is to perform read() and load() function on CSV data and display top 5 rows and to display the statistical results from the data frame.

Solution Overview:

After the learning this code snippet, the learner will able to understand about the concept of RDD and Creation of .CSV file and insert 10 rows and 3 columns of numerical value and upload in to Files in colab. Students will also know about the data frame creation.

Intuition:

RDD (Resilient Distributed Datasets) in Spark is a fundamental data structure that represents an immutable, fault-tolerant collection of data that can be processed in parallel across a cluster of machines.

- RDDs can be created from various data sources, including Hadoop Distributed File System (HDFS), local file systems, and external databases, and can be transformed using various operations such as map, filter, and reduce to perform complex computations.

RDDs are the basic building blocks of Spark applications and provide a high-level API for performing distributed data processing.

Code Implementation:

Sample Code:
Step1:
pip install spark

step2:
pip install pyspark

Step3:
Create one .CSV file and insert 10 rows and 3 cloumns of numerical value and uplaod in to Files in colab.

Step4: Code

```
from pyspark.sql import SparkSession

# create a SparkSession object
spark = SparkSession.builder.appName("CSV RDD").getOrCreate()

# create an RDD from a CSV file
rdd = spark.read.format("csv").option("header", "true").load("/Book1.csv").rdd

# display the top 5 rows of the RDD
print(rdd.take(5))
# convert the RDD to a DataFrame
df = rdd.toDF()

# display the statistical results
df.describe().show()
```

Sample output:

[Row(radio='99.1', Newspaper='22.1', Sales='44.1'), Row(radio='20.1', Newspaper='23.4', Sales='55.1'), Row(radio='33.1', Newspaper='26.4', Sales='66.1'), Row(radio='44.1', Newspaper='123.1', Sales='88.1'), Row(radio='88.1', Newspaper='20.1', Sales='12.1')]

```
+-------+-----------------+-----------------+-----------------+
|summary|          radio|        Newspaper|           Sales|
+-------+-----------------+-----------------+-----------------+
|  count|               8 |               8 |               8|
|   mean|           48.35 |          39.325 | 39.35000000000001|
```

```
| stddev|35.30746582321001|34.75755827360226|28.49937342669634|
|   min|          10      |          123.1  |          12.1|
|   max|          99.1    |          44.1   |          88.1|
+-------+-----------------+-----------------+-----------------+
```

Outcome of the Exercise:

When you run the provided PySpark code, the CSV data is first loaded into a DataFrame using the read() and load() functions. The DataFrame contains rows and columns structured according to the CSV file, with the schema inferred automatically if specified. After loading the data, the DataFrame is converted into an RDD, which allows for distributed data processing across multiple nodes.

When you display the top 5 rows of the DataFrame using df.show(5), you'll see a table that lists the first 5 entries from your dataset. Each row corresponds to a record in your CSV file, with values displayed under each column header. This helps in quickly inspecting the contents of the dataset.

The statistical summary generated by df.describe().show() provides a descriptive overview of the numeric columns in your DataFrame. It includes statistics such as count (number of non-null values), mean (average), standard deviation (measure of variability), minimum, and maximum values for each numeric column. This summary is useful for understanding the distribution and range of the data at a glance.

Overall, these operations allow you to efficiently load, inspect, and analyze the structure and basic statistics of your CSV data in a distributed computing environment.

Interview Questions:

1. What is a SparkSession and why do you need it?
   - Answer: A SparkSession is the entry point to using Spark's capabilities, providing a unified interface to manage and interact with various Spark functionalities. It is required to create DataFrames, manage resources, and execute SQL queries. Essentially, it initializes the Spark context and acts as the environment for Spark applications.
2. How does Spark infer the schema of a DataFrame when loading a CSV file?
   - Answer: Spark infers the schema by scanning a sample of the data in the CSV file and analyzing the data types of each column. If inferSchema is set to true, Spark will automatically determine the most appropriate data type (e.g., Integer, Double, String) for each column based on the sampled data.
3. What is the difference between a DataFrame and an RDD in Spark?
   - Answer: A DataFrame is a higher-level abstraction built on top of RDDs, designed to handle structured data with schema (similar to a table in a relational database). DataFrames allow for optimizations and SQL-like operations, making them more efficient and easier to use than RDDs. RDDs (Resilient Distributed Datasets) are lower-level data structures that are more flexible but require more manual control over operations and optimizations.
4. Why would you convert a DataFrame to an RDD in Spark?
   - Answer: You might convert a DataFrame to an RDD if you need to perform operations that are not supported by DataFrames or if you require more fine-grained control over the data and processing steps. RDDs allow for custom transformations and actions that may not be easily expressed using DataFrames.
5. What does the df.show(5) command do in Spark?
   - Answer: The df.show(5) command displays the first 5 rows of the DataFrame in a tabular format. This command is useful for quickly inspecting the contents and structure of your dataset to ensure that the data has been loaded correctly and to understand its layout.
6. What kind of information does df.describe().show() provide?
   - Answer: The df.describe().show() command provides a statistical summary of the numeric columns in the DataFrame. It includes metrics such as count (number of non-null entries), mean (average value), standard deviation (variability), minimum, and maximum values. This summary helps in understanding the distribution and central tendencies of the data.
7. What are some advantages of using DataFrames over RDDs in Spark?
   - Answer: DataFrames offer several advantages over RDDs, including:
     - Performance Optimization: DataFrames benefit from Spark's Catalyst optimizer, which can significantly improve execution speed through query planning and optimization.
     - Ease of Use: DataFrames provide a higher-level API with SQL-like operations, making it easier to write and read code.
     - Interoperability: DataFrames can easily integrate with other data sources and formats, such as Parquet, ORC, and Hive, as well as support for SQL queries.
     - Less Boilerplate Code: DataFrame operations are generally more concise and require less boilerplate code compared to RDDs.

8. What happens if you try to load a CSV file without setting the inferSchema option?
   - Answer: If the inferSchema option is not set (or set to false), Spark will treat all columns in the CSV file as strings (StringType). This may lead to incorrect data types being assigned, such as numeric data being interpreted as strings, which could affect the accuracy of subsequent operations like aggregations and calculations.
9. How does Spark handle missing or null values in a DataFrame when generating statistical summaries?
   - Answer: When generating statistical summaries with df.describe(), Spark automatically excludes null values from the calculations. The count in the summary represents the number of non-null entries for each column. If a column has missing values, the count will reflect the number of rows with actual (non-null) data for that column.
10. What are the key components of Apache Spark?
    - Answer: The key components of Apache Spark include:
      - Spark Core: The engine that handles basic I/O functions, task scheduling, and memory management.
      - Spark SQL: Provides support for structured data processing and allows running SQL queries.
      - Spark Streaming: Enables real-time data stream processing.
      - MLlib: A machine learning library for Spark.
      - GraphX: A library for graph processing.
11. How does Spark achieve fault tolerance?
    - Answer: Spark achieves fault tolerance through two main mechanisms:
      - Lineage Graph: Each RDD maintains a lineage graph that tracks the sequence of transformations used to build it. If an RDD partition is lost, Spark can recompute it using the lineage information.
      - Data Replication: In some cases, data is replicated across multiple nodes in the cluster, ensuring that if one node fails, another can provide the required data.
12. What is a DAG (Directed Acyclic Graph) in Spark?
    - Answer: A DAG in Spark represents the sequence of computations to be performed on the data, where each node corresponds to an RDD and edges represent transformations applied to the data. Spark uses the DAG to optimize and execute the jobs in parallel, ensuring efficient resource utilization.
13. What are the types of cluster managers Spark can run on?
    - Answer: Spark can run on several cluster managers, including:
      - Standalone Cluster Manager: A simple built-in cluster manager.
      - Apache Mesos: A general cluster manager that can also run Hadoop jobs.
      - Hadoop YARN: The resource manager in the Hadoop ecosystem.
      - Kubernetes: A container orchestration system that can manage Spark jobs in a containerized environment.
14. Explain the difference between transformations and actions in Spark.
    - Answer:
      - Transformations: These are operations that create a new RDD/DataFrame from an existing one. They are lazy, meaning they don't compute results immediately but rather create a plan for how to compute the data when an action is called (e.g., map(), filter(), groupBy()).
      - Actions: These operations trigger the actual computation and return a result to the driver or write it to storage (e.g., collect(), count(), saveAsTextFile()).

Data Processing

15. How do you handle skewed data in Spark?
    - Answer: To handle skewed data in Spark, you can:
      - Salting: Add random keys to data to distribute it more evenly across partitions.
      - Increase Partitioning: Increase the number of partitions to distribute data more evenly.
      - Use Skewed Join Handling: Use built-in functions like broadcast() for small tables or perform a map-side join.
16. What is a wide transformation and how does it impact performance?
    - Answer: A wide transformation is a transformation that requires shuffling data across the network between nodes, such as groupByKey(), reduceByKey(), or join(). These operations are more expensive because they involve data movement and repartitioning, potentially leading to increased network I/O and longer job execution times.
17. What is the difference between map() and flatMap() transformations?
    - Answer:
      - map() applies a function to each element in the RDD and returns a new RDD with the same number of elements.

- flatMap() also applies a function to each element, but the function can return multiple elements (e.g., splitting a sentence into words), resulting in a potentially larger RDD.

18. How do you perform a join operation in Spark?
    - Answer: Join operations in Spark can be performed using the join() method on DataFrames or RDDs. You can perform various types of joins, such as inner joins, left outer joins, right outer joins, and full outer joins. It is often recommended to use broadcast() to optimize joins when one of the tables is small enough to fit in memory.
19. Explain the concept of caching/persisting in Spark. When should you use it? –
    - Answer: Caching or persisting an RDD/DataFrame stores the dataset in memory (or disk) across operations, allowing faster access to the data in subsequent actions. It should be used when you have multiple actions that need to be performed on the same dataset, thereby avoiding the recomputation of the data each time. You can use cache() for memory storage or persist() with a specified storage level (e.g., memory and disk).
20. What are some techniques for optimizing Spark jobs?
    - Answer: Techniques for optimizing Spark jobs include: - Avoiding Shuffles: Minimize wide transformations and use techniques like map-side reductions. - Broadcast Joins: Use broadcast() to join large datasets with small ones efficiently. - Caching/Persisting: Cache/persist intermediate results if used multiple times. - Partitioning: Ensure proper partitioning of data to avoid skew and to ensure parallelism. - Use of DataFrames/Datasets: Prefer DataFrames/Datasets over RDDs as they benefit from Spark's Catalyst optimizer. - Reducing Data Size: Filter out unnecessary data early in the pipeline and select only the columns needed.
21. What are the benefits of using DataFrames over RDDs in Spark?
    - Answer: Benefits of using DataFrames over RDDs include: - Optimized Execution: DataFrames are optimized by the Catalyst query optimizer and benefit from Tungsten's in-memory computation engine. - Ease of Use: DataFrames offer higher-level abstractions with SQL-like syntax, making them easier to use. - Reduced Memory Footprint: DataFrames use an optimized, columnar format for storage, reducing the memory footprint compared to RDDs. - Interoperability: DataFrames can be easily integrated with other components in the Spark ecosystem, such as Spark SQL.

4

Given the following tables, Perform Full outer join on dataframe.

Age	Name
2	Alice
5	Bob

Height	Name
80	Tom
85	Bob

Age	Name
2	Alice
5	Bob

Age	Height	Name
10	80	Alice
5	None	Bob
None	None	Tom
None	None	None

a. Write query and Perform Join of df1 and df2 to find the height
b. Write query and Perform outer join of df1 and df2 to find the heights
c. Write query and Perform outer join of df1 and df2 to find the age

```
#performs a full outer join between df1 and df2.
from pyspark.sql import Row
from pyspark.sql.functions import desc
df = spark.createDataFrame([(2, "Alice"), (5, "Bob")]).toDF("age", "name")
```

```
df2 = spark.createDataFrame([Row(height=80, name="Tom"), Row(height=85, name="Bob")])
df3 = spark.createDataFrame([Row(age=2, name="Alice"), Row(age=5, name="Bob")])
df4 = spark.createDataFrame([
    Row(age=10, height=80, name="Alice"),
    Row(age=5, height=None, name="Bob"),
    Row(age=None, height=None, name="Tom"),
    Row(age=None, height=None, name=None),
])
df.join(df2, 'name').select(df.name, df2.height).show()
```

Sample output:

```
+----+------+
|name|height|
+----+------+
| Bob|    85|
+----+------+
```

```
df.join(df2, df.name == df2.name, 'outer').select(
    df.name, df2.height).sort(desc("name")).show()
```

```
+-----+------+
| name|height|
+-----+------+
|  Bob|    85|
|Alice|  NULL|
| NULL|    80|
+-----+------+
```

```
#Outer join for both DataFrames with multiple columns.
df.join(
    df3,
    [df.name == df3.name, df.age == df3.age],
    'outer'
).select(df.name, df3.age).show()
```

```
+-----+---+
| name|age|
+-----+---+
|Alice|  2|
|  Bob|  5|
+-----+---+
```

5

Given the following data
data = [("1", "john jones"), ("2", "tracey smith"), ("3", " amy sanders")], along with the following schema of the data
columns = ["Seqno","Name"]
Perform the following using the afore mentioned data.
i. create an RDD from the above data using its schema
ii. create the PySparkdataframe from the RDD created.
iii. Write python functions to convert the first letter of every string into upper case.
iv. Use the above python function as udf in pySpark to convert the data in the dataframe and display the result.

Problem Statement:
The task is to perform different operations for the mentioned data.
Solution Overview:

- **DD Creation**: spark.sparkContext.parallelize(data) creates an RDD from the provided data.
- **DataFrame Creation**: spark.createDataFrame(rdd, schema) creates a DataFrame with the given schema.
- **Python Function**: capitalize_first_letter changes the first letter of a string to uppercase.

- **UDF Registration and Application**: The function is registered as a UDF and applied to the DataFrame column using withColumn.

Intuition:
To solve this problem, you'll first create a basic data structure (RDD) from the provided data using the schema. Convert this RDD into a DataFrame so that you can leverage PySpark's powerful DataFrame operations. By using capitalize it will convert the first letter of every string into upper case. Then, by importing udf in pyspark will convert the data in the dataframe.

Code Implementation:

i. Create an RDD from the above data using its schema

```
from pyspark.sql import SparkSession

# Initialize Spark session

spark = SparkSession.builder \

.appName("RDD and DataFrame Example") \

.getOrCreate()

# Data and schema

data = [("1", "john jones"), ("2", "tracey smith"), ("3", "amy sanders")]

columns = ["Seqno", "Name"]

# Create RDD

rdd = spark.sparkContext.parallelize(data)
```

Sample Output:

```
+----+-------------+

|Seqno|        Name|

+----+-------------+

|   1| john jones  |

|   2| tracey smith|

|   3| amy s anders|

+----+-------------+
```

Description:

- **Column Seqno**: Contains the sequence numbers as strings ("1", "2", "3").
- **Column Name**: Contains the names as strings ("john jones", "tracey smith", "amy s anders").

Each row in the DataFrame represents a tuple from the original data with the Seqno and Name columns correctly populated according to the given schema. The output is displayed without truncation, meaning the full content of each field is shown.

ii. Create the PySparkdataframe from the RDD created.

```
from pyspark.sql import Row

# Convert RDD to DataFrame
df = rdd.map(lambda x: Row(Seqno=x[0], Name=x[1])).toDF()

# Show the DataFrame
df.show()
```

Sample Output:

The DataFrame will be displayed as follows:

plaintext
Copy code

```
+----+-------------+
|Seqno|        Name|
+----+-------------+
|   1| john jones  |
|   2| tracey smith|
|   3| amy s anders|
+----+-------------+
```

This output shows each row of the DataFrame with the specified schema applied, where Seqno and Name columns are populated according to the provided data.

iii: Write a Python functions to convert the first letter of every string into upper case.

```
def capitalize_first_letter(s):

return s.title() if s else s
```

Sample Output:

After running the above code, the DataFrame will have the "Name" column with each word's first letter capitalized:

plaintext
Copy code

```
+----+-------------+
|Seqno|        Name|
+----+-------------+
|   1| John Jones  |
|   2| Tracey Smith|
|   3| Amy S Anders|
+----+-------------+
```

This output shows that the first letter of each word in the "Name" column has been capitalized correctly.

iv. Use the above python function as udf in pySpark to convert the data in the dataframe and display the result.

```
from pyspark.sql.functions

import udf from pyspark.sql.types import StringType

# Register the UDF

capitalize_udf = udf(capitalize_first_letter, StringType())

# Apply the UDF to the 'Name' column

df_transformed = df.withColumn("Name", capitalize_udf(df["Name"]))

# Show the transformed DataFrame

df_transformed.show()
```

Sample Output:

```
+----+-------------+

|Seqno|        Name|

+----+-------------+

|   1| John Jones  |

|   2| Tracey Smith|

|   3| Amy S Anders|

+----+-------------+
```

Outcome of the Exercise:

The outcomes of the program are:

- The DataFrame is successfully created from the RDD.
- The Python function capitalizes the first letter of each word in the "Name" column.
- The resulting DataFrame has names with properly capitalized words, improving readability and consistency.

This approach ensures that names and similar data entries are formatted with proper capitalization, which is useful for standardizing textual data.

Interview Questions:

1. What is SparkSession and why is it important?
   Answer: SparkSession is the entry point for programming with Spark in Python. It provides a unified interface for interacting with different Spark components, including DataFrames and SQL. It is essential because it allows you to create RDDs, DataFrames, and Datasets, and run SQL queries.
2. What is an RDD in Spark?

Answer: RDD (Resilient Distributed Dataset) is the fundamental data structure in Spark. It represents a distributed collection of objects that can be processed in parallel. RDDs are immutable and support operations like map, filter, and reduce.

3. What is a DataFrame in Spark?
   Answer: A DataFrame is a distributed collection of data organized into named columns. It is conceptually similar to a table in a relational database or a data frame in Python's pandas library. DataFrames provide an abstraction for working with structured data and support a wide range of operations, including filtering, aggregation, and joining.
4. How do you create an RDD from a list of tuples in PySpark?
   Answer: You can create an RDD from a list of tuples using the parallelize method of the 'SparkContext'.
5. What is the purpose of defining a schema in Spark DataFrames?
   Answer: Defining a schema ensures that the DataFrame has a specific structure with defined column names and data types. This helps Spark understand how to interpret and process the data, providing better optimization and validation.
6. How do you define and apply a UDF in PySpark?
   Answer: To define a UDF, you first create a Python function that performs the desired transformation. Then, you use udf to register this function with Spark, specifying the return type. After registering the UDF, you can apply it to DataFrame columns using withColumn.
7. Explain how the UDF capitalize_first_letter works.
   Answer: The capitalize_first_letter function splits the input string into words, capitalizes the first letter of each word, and then joins the words back together with spaces. This ensures that each word in the string starts with an uppercase letter.
8. What does the df.show(truncate=False) method do?
   Answer: The df.show(truncate=False) method displays the contents of the DataFrame. The truncate=False argument ensures that the full content of each column is shown without truncation, which is useful for viewing the complete data.
9. What would happen if you applied the UDF to a column with null values?
   Answer: If the column contains null values, the UDF will return null for those rows. The capitalize_first_letter function handles null values by checking if the input is non-null before processing.
10. Why is it important to use UDFs for custom transformations?
    Answer: UDFs (User Defined Functions) allow you to perform custom transformations that are not available through built-in functions in Spark SQL. They provide flexibility to apply complex logic or transformations to DataFrame columns that are specific to your use case.
11. How does Spark handle the parallelization of operations on DataFrames?
    Answer: Spark automatically handles the parallelization of operations on DataFrames. When you perform transformations or actions on a DataFrame, Spark distributes the computation across the cluster, executing tasks in parallel to improve performance and scalability.
12. What are some best practices when working with UDFs in PySpark?
    Answer: Best practices for working with UDFs in PySpark include:
    - Avoiding excessive use of UDFs: Built-in functions are optimized for performance. Use UDFs only when necessary.
    - Ensuring UDFs are efficient: UDFs should be optimized for performance as they can affect the overall speed of Spark jobs.
    - Handling null values: Ensure your UDF can handle null values gracefully to avoid unexpected errors.

6

Given the following data.
Data = [("James","Sales","NY",90000,34,10000),
("Michael","Sales","NV",86000,56,20000),
("Robert","Sales","CA",81000,30,23000),
("Maria","Finance","CA",90000,24,23000),
("Raman","Finance","DE",99000,40,24000),
("Scott","Finance","NY",83000,36,19000),
("Jen","Finance","NY",79000,53,15000),
("Jeff","Marketing","NV",80000,25,18000),

("Kumar","Marketing","NJ",91000,50,21000)
], with the following schema schema =
["employee_name","department","state","salary","age ","bonus"]

Perform the following using the aforementioned data.
i. create an RDD from the above data using its schema
ii. create the PySparkdataframe from the RDD created.
iii. Using groupBy() function, display the salaries of the employees state-wise.
iv. Display the state-wise salaries that are greater than 1 lakh
v. Display the state-wise salaries in descending order.

Problem Statement:
To create an RDD from the given data using the provided schema.
To convert the RDD created in Task 1 into a PySpark DataFrame.
To use the groupBy() function to group the DataFrame by the state column and perform aggregations as needed.
To display the state-wise salaries where the salary is greater than 1 lakh.
To display the state-wise salaries in descending order and show the salaries of the employees state-wise.

Solution Overview:
To solve the problem, start by creating an RDD from the provided data using the given schema. Convert this RDD into a PySpark DataFrame. Next, use the groupBy() function to group the data by the "state" column. Then, filter the grouped data to show only those states where the total salary exceeds 1 lakh. Finally, sort and display the state-wise total salaries in descending order.
Intuition: To solve this problem, you'll first create a basic data structure (RDD) from the provided data using the schema. Convert this RDD into a DataFrame so that you can leverage PySpark's powerful DataFrame operations. By using groupBy() on the "state" column, you can aggregate the salaries by state. Then, filter the results to only show those states where the combined salaries exceed 1 lakh. Finally, sort these results in descending order to see which states have the highest total salaries.

Intuition:
To solve this problem, you'll first create a basic data structure (RDD) from the provided data using the schema. Convert this RDD into a DataFrame so that you can leverage PySpark's powerful DataFrame operations. By using groupBy() on the "state" column, you can aggregate the salaries by state. Then, filter the results to only show those states where the combined salaries exceed 1 lakh. Finally, sort these results in descending order to see which states have the highest total salaries.

Code Implementation (in C Programming Language):
Sample Data:
```
data = [("James","Sales","NY",90000,34,10000),
        ("Michael","Sales","NV",86000,56,20000),
        ("Robert","Sales","CA",81000,30,23000),
        ("Maria","Finance","CA",90000,24,23000),
        ("Raman","Finance","DE",99000,40,24000),
        ("Scott","Finance","NY",83000,36,19000),
        ("Jen","Finance","NY",79000,53,15000),
        ("Jeff","Marketing","NV",80000,25,18000),
        ("Kumar","Marketing","NJ",91000,50,21000)]
```

Sample Code:
```
from pyspark.sql import SparkSession
from pyspark.sql import Row
from pyspark.sql.types import StructType, StructField, StringType, IntegerType

# Initialize Spark session
spark = SparkSession.builder.appName("StatewiseSalary").getOrCreate()

# Define schema
schema = ["employee_name","department","state","salary","age","bonus"]
```

```
# Create RDD
rdd = spark.sparkContext.parallelize(data)

# Convert RDD to DataFrame
df = spark.createDataFrame(rdd, schema)

# Group by state and sum the salaries
result = df.groupBy("state").sum("salary")

# Filter states where the total salary is greater than 1 lakh
filtered_result = result.filter("sum(salary) > 100000")

# Sort by salary in descending order
final_result = filtered_result.orderBy("sum(salary)", ascending=False)

# Show the result
final_result.show()
```

Sample Output:

```
+-----+-----------+
|state|sum(salary)|
+-----+-----------+
|  NY |     252000|
|  CA |     171000|
|  NV |     166000|
|  NJ |      91000|
|  DE |      99000|
+-----+-----------+
```

Outcome of the Exercise:

The outcome reflects the state-wise aggregation of salaries, highlighting those states with the highest cumulative employee salaries, sorted in descending order.

- **NY** (New York) has the highest total salary of **252,000**.
- **CA** (California) follows with **171,000**.
- **NV** (Nevada) is next with **166,000**.
- **DE** (Delaware) and **NJ** (New Jersey) have total salaries just under 1 lakh but still show up in the order of their respective sums.

Interview Questions:

1. What is an RDD in PySpark, and how does it differ from a DataFrame?
   Answer: An RDD (Resilient Distributed Dataset) in PySpark is a fundamental data structure that represents an immutable, distributed collection of objects across a cluster. RDDs allow operations like map, filter, and reduce, and provide fault tolerance through lineage graphs. In contrast, a DataFrame is a higher-level abstraction built on top of RDDs, offering a tabular structure similar to a database table, with named columns. DataFrames are optimized for performance through Catalyst optimization and Tungsten execution engine, making them more efficient than RDDs for most operations. While RDDs provide more control over data processing, DataFrames are generally preferred for their ease of use and performance benefits.
2. How do you create an RDD from a list of tuples in PySpark?
   Answer: To create an RDD from a list of tuples in PySpark, you first need to initialize a SparkContext. Once the context is set up, you can use the parallelize() method of SparkContext, passing the list of tuples as the argument. This method distributes the data across the cluster and returns an RDD. For example, if sc is your SparkContext, you can create an RDD with rdd = sc.parallelize([(1, 2), (3, 4), (5, 6)]). The resulting RDD can then be used for various transformations and actions.

3. Why is it important to define a schema when working with PySpark DataFrames?
Answer: Defining a schema when working with PySpark DataFrames is important because it ensures data consistency and type safety across operations. By specifying the schema, you provide clear expectations for the structure and data types of each column, reducing the chances of runtime errors due to unexpected data types. It also improves performance, as Spark can skip the inferencing step, which can be costly, especially with large datasets. Additionally, defining a schema allows for better handling of complex and nested data structures. Overall, it enhances data processing efficiency and reliability in PySpark applications.
4. How can you convert an RDD into a DataFrame in PySpark?
Answer: To convert an RDD into a DataFrame in PySpark, you first need to have a SparkSession object. Once you have the SparkSession, you can use the createDataFrame() method, passing your RDD as the first argument. Optionally, you can also provide a schema for the DataFrame by passing a StructType object as the second argument. For example, if rdd is your RDD and schema is your defined schema, you would convert it with df = spark.createDataFrame(rdd, schema). This conversion allows you to leverage the optimized operations and SQL-like querying capabilities of DataFrames.
5. What is the purpose of the groupBy() function in PySpark, and how does it work?
Answer: The groupBy() function in PySpark is used to group data in a DataFrame or RDD based on one or more columns or keys. It works by returning a GroupedData object, which can then be used to perform aggregate functions such as sum(), count(), avg(), and more on the grouped data. For example, using groupBy("columnName").sum("otherColumn") will group the data by "columnName" and then calculate the sum of "otherColumn" for each group. This function is useful for analyzing patterns, summarizing data, and performing operations on subsets of data within a larger dataset. Overall, groupBy() enables efficient and scalable data aggregation in PySpark.
6. How would you filter DataFrame rows where the sum of salaries is greater than 1 lakh?
Answer: To filter DataFrame rows where the sum of salaries is greater than 1 lakh in PySpark, you can first group the data by the relevant column (if needed) using the groupBy() function. Then, apply the sum() function on the salary column to calculate the total salary for each group. After that, use the filter() function to keep only those groups where the sum of salaries exceeds 1 lakh. For example, if df is your DataFrame and "salary" is the column, you might write df.groupBy("someColumn").sum("salary").filter("sum(salary) > 100000"). This approach allows you to efficiently filter out rows based on the aggregated salary criteria.
7. Can you explain the steps to sort DataFrame results in descending order based on a specific column?
Answer: To sort DataFrame results in descending order based on a specific column in PySpark, you follow these steps:
    1. **Select the Column**: Identify the column you want to sort by within your DataFrame.
    2. **Use the orderBy() Function**: Apply the orderBy() function on the DataFrame, passing the column name as an argument.
    3. **Specify Descending Order**: To sort in descending order, you can either use the desc() function within orderBy() or pass the column name as a string and set the ascending parameter to False. For example, df.orderBy(df["columnName"].desc()) or df.orderBy("columnName", ascending=False).
    4. **Execute the Transformation**: This sorting operation returns a new DataFrame with the rows ordered as specified.
    5. **View or Process Results**: You can now use show(), collect(), or other actions to view or further process the sorted DataFrame.

    This sequence efficiently sorts the DataFrame based on the chosen column in descending order.

8. What is the significance of using agg() after groupBy() in PySpark?
Answer: The agg() function in PySpark is significant because it allows you to perform multiple aggregate operations simultaneously after using groupBy(). When you group data with groupBy(), the agg() function provides a flexible way to apply different aggregate functions, like sum(), avg(), max(), and min(), on different columns within the grouped data. For example, after grouping by a column, you can use agg() to calculate both the sum of one column and the average of another in a single step. This capability simplifies complex aggregation tasks and makes your code more efficient and readable. Essentially, agg() enhances the power of groupBy() by enabling customized and multi-faceted data summarization.
9. How would you handle a situation where the schema provided doesn't match the data accurately?
Answer: If the schema provided doesn't match the data accurately in PySpark, it can lead to errors or incorrect data processing. To handle this situation:

1. **Verify and Adjust the Schema**: Start by verifying the schema against the data and adjusting it to match the actual data types and structure. This might involve changing data types or adding/removing fields.
2. **Use Data Validation**: Implement data validation checks before applying the schema to identify and handle mismatches, such as unexpected data types or null values.
3. **Schema Inference**: If feasible, allow PySpark to infer the schema automatically by using inferSchema=True when reading the data, which can adapt the schema based on the actual data content.
4. **Error Handling**: Implement error handling strategies, such as using try-except blocks or handling corrupt records with options like mode="DROPMALFORMED" when reading data, to prevent the application from crashing.
5. **Data Cleansing**: Pre-process and cleanse the data to ensure it conforms to the expected schema, which may involve type casting, filling in missing values, or filtering out problematic records.

10. What are some common use cases where you would need to group and aggregate data in PySpark?

Answer: Grouping and aggregating data in PySpark is commonly used in several scenarios:

1. **Summarizing Sales Data**: Businesses often group sales data by regions, products, or time periods (e.g., months or quarters) to calculate total sales, average revenue, or other key metrics.
2. **Customer Analytics**: Companies group customer data by demographics, such as age, location, or customer segments, to analyze behaviors, preferences, and purchase patterns.
3. **Performance Monitoring**: Aggregating data by metrics like response times, error rates, or resource usage helps in monitoring and optimizing system or application performance.
4. **Financial Reporting**: Financial analysts group transaction data by account types, departments, or time periods to generate summaries like total expenditures, profits, or budget variances.
5. **Data Cleaning and Validation**: Aggregating data to check for consistency, such as counting occurrences of values in categorical columns or detecting duplicates, helps in data cleaning and ensuring data quality.

Additional Interview questions

1. Explain the difference between RDD and DataFrame.
2. When would you choose RDD over DataFrame and vice versa?
3. How do you convert an RDD to a DataFrame and vice versa?
4. What is a schema in the context of Spark?
5. How do you define a schema for an RDD?
6. Explain the groupBy operation in Spark.
7. What are common aggregation functions used with groupBy?
8. How does Spark optimize groupBy operations?
9. How would you handle missing values in the salary column?
10. What if the state column has invalid or inconsistent values?
11. How would you filter out employees with salaries less than 1 lakh before grouping?
12. How would you optimize the performance of the groupBy operation for large datasets?
13. What are the potential performance implications of using different data types for the salary column?
14. Write code snippets to perform specific tasks, such as creating the RDD, defining the schema, converting to DataFrame, grouping, and aggregating.

7 **Given the following data.**

First_name	Age
Sue	32
Li	3
Bob	75
Heo	13

a. Create a dataframe
b. Check the lifestage of each person into Adult, Child and Teenager
c. Write query to Display entries of teenager and adult only
d. Write query to average age

e. Write query to group by entries by life_stage
f. Insert a record “Frank,4, Child) into new data frame
g. Write a query to display teenage entries

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("demo").getOrCreate()
df = spark.createDataFrame(
    [
        ("sue", 32),
        ("li", 3),
        ("bob", 75),
        ("heo", 13),
    ],
    ["first_name", "age"],
)

df.show()
from pyspark.sql.functions import col, when

df1 = df.withColumn(
    "life_stage",
    when(col("age") < 13, "child")
    .when(col("age").between(13, 19), "teenager")
    .otherwise("adult"),
)

df1.show()
df1.where(col("life_stage").isin(["teenager", "adult"])).show()

from pyspark.sql.functions import avg

df1.select(avg("age")).show()
df1.groupBy("life_stage").avg().show()
spark.sql("select avg(age) from {df1}", df1=df1).show()
spark.sql("select life_stage, avg(age) from {df1} group by life_stage", df1=df1).show()
df1.write.saveAsTable("some_people")
spark.sql("select * from some_people").show()
spark.sql("INSERT INTO some_people VALUES ('frank', 4, 'child')")
spark.sql("select * from some_people").show()
spark.sql("select * from some_people where life_stage='teenager'").show()
```

sample output:

```
+----------+---+
|first_name|age|
+----------+---+
|      sue| 32|
|       li|  3|
|      bob| 75|
|      heo| 13|
+----------+---+
+----------+---+----------+
|first_name|age|life_stage|
+----------+---+----------+
|      sue| 32|     adult|
|       li|  3|     child|
|      bob| 75|     adult|
```

```
|      heo| 13|  teenager|
+----------+---+----------+
+----------+---+----------+
|first_name|age|life_stage|
+----------+---+----------+
|      sue| 32|     adult|
|      bob| 75|     adult|
|      heo| 13|  teenager|
+----------+---+----------+

+--------+
|avg(age)|
+--------+
|   30.75|
+--------+

+----------+--------+

|life_stage|avg(age)|

+----------+--------+

|     adult|    53.5|

|     child|     3.0|

|  teenager|    13.0|

+----------+--------+

+--------+

|avg(age)|

+--------+

|   30.75|

+--------+

+----------+--------+

|life_stage|avg(age)|

+----------+--------+

|     adult|    53.5|

|     child|     3.0|

|  teenager|    13.0|

+----------+--------+

+----------+---+----------+

|first_name|age|life_stage|

+----------+---+----------+

|      heo| 13|  teenager|

|      sue| 32|     adult|

|      bob| 75|     adult|
```

```
|       li|  3|     child|
+----------+---+----------+
+----------+---+----------+
|first_name|age|life_stage|
+----------+---+----------+
|      heo| 13|  teenager|
|    frank|  4|     child|
|      sue| 32|     adult|
|      bob| 75|     adult|
|       li|  3|     child|
+----------+---+----------+
+----------+---+----------+
|first_name|age|life_stage|
+----------+---+----------+
|      heo| 13|  teenager|
+----------+---+----------+
```

8

Write a Word Count Map Reduce program to understand Map Reduce Paradigm.

Problem Statement:
The task is to perform different operations for the mentioned data.

Solution Overview:
- **Mapper.py Creation: this is done to write the Mapping function to create key value pairs**
- **Reducer.pyCreation: reducer is written to take input from the Mapper.py and then to perform the reducing**
-

Intuition:
To solve this problem, you'll first create a Mapper.py file. this is done to write the Mapping function to create key value pairs and then Reducer.pyCreation: reducer is written to take input from the Mapper.py and then to perform the reducing. Then check with just the mapper the output and also see how the reducer helps in getting better output.

Mapper.py

```
#!/usr/bin/env python
"""mapper.py"""

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
```

```
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print ('%s\t%s' % (word, 1))

Reducer.py
#!/usr/bin/env python
"""reducer.py"""

from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()

    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)

    # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        continue

    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print ('%s\t%s' % (current_word, current_count))
        current_count = count
        current_word = word

# do not forget to output the last word if needed!
if current_word == word:
    print ('%s\t%s' % (current_word, current_count))


Sample Output:
echo "foo foo foo quux labs labs foo bar labs quux" |python mapper.py
foo     1
foo     1
foo     1
```

```
quux    1
labs    1
labs    1
foo     1
bar     1
labs    1
quux    1
```

echo "foo foo foo quux labs labs foo bar labs quux" |python mapper.py|python shuffleSort.py|python reducer.py

```
bar     1
foo     4
labs    3
quux    2
```

Outcome of the Exercise:
After executing the program, Students will able to

1. Understand about the concept of RDD and Creation of Text. Also, finds the count of unique words.
2. Gain practical experience in using PySpark for data processing tasks and develop a deeper understanding of RDD transformations and actions.

Interview Questions:

1. What is an RDD in PySpark?
   An RDD (Resilient Distributed Dataset) in PySpark is a fundamental data structure that represents an immutable, distributed collection of objects across a cluster. RDDs allow operations like map, filter, and reduce, and provide fault tolerance through lineage graphs.

2. How do you create an RDD from an external text file?
   - Set Up Spark Environment: First, you need to initialize a SparkSession or SparkContext.
   - Load Data: Use the textFile method to load data from an external file. This method reads the file and creates an RDD where each element corresponds to a line in the file.
   - Perform Actions or Transformations: Once the RDD is created, you can apply various transformations or actions as needed.

3. What transformations are used to perform word count in PySpark?
   flatMap: Splits each line into words and flattens the result.
   map: Cleans and normalizes each word.
   filter: Removes any empty strings.
   map: Creates key-value pairs for counting.
   reduceByKey: Aggregates counts for each unique word.

4. What is flatmap transformation?
   It splits each line into words and flattens the result

5. What is the difference between transformations and actions in PySpark?
   Transformations are operations that create a new RDD from an existing one. Actions are operations that trigger the execution of the transformations defined on RDDs and return a value to the driver program or write data to an external storage system.

6. What is the advantage of PySpark in dataprocessing.
   PySpark's significance in data processing stems from its speed, ease of use, ability to handle large datasets, support for distributed computing, and integration with a rich ecosystem of tools and libraries.

7. Explain the two types of operations in RDD

RDDs support two types of operations:

Transformations: Create a new RDD from an existing one, e.g. map(), filter(), reduceByKey().

Actions: Return a result to the driver program or write data to an external storage system, e.g. collect(), count(), saveAsTextFile().

8. What are the advantages of transformations and actions in efficient data processing

It is fundamental for efficient data processing in PySpark, as it allows developers to optimize their workflows and manage resources effectively.

9. What role does the reduceByKey transformation play in word counting

The reduceByKey transformation plays a crucial role in the word counting process in PySpark by aggregating the counts of each unique word efficiently.

10. How is RDD support partitioning

RDDs can be partitioned based on keys to enable efficient shuffling and reduce data movement during operations like groupByKey() and reduceByKey()

Additional Questions

1.What are the advantages of Data Partitioning
2.What is the use of spark.read.csv
3.What are the characteristics of Actions in RDD
4.How can you handle DataFrames
5.How can you Split Lines into Words
6.Explain the Interoperability of RDD with Other Libraries
7.What is the use of collect() action
8.How does PySpark's RDD handle data recovery in case of node failures
9.What is the importance of reduceByKey as associative and cumulative function?
10.Why is RDD considered as Immutable

PART – B

BIG DATA LAB MINI PROJECT REPORT FORMAT

11. Project Report Format:

Part-B

Implement and demonstrate any real life big data problem using any of the publicly available big data sets.

1. General Formatting Instructions

Font: Use Times New Roman for the entire document.

Font Size:

Main text: 12 pt

Headings: 14 pt (bold)

Subheadings: 12 pt (bold)

Captions: 10 pt

Line Spacing: Set the line spacing to 1.5 lines.

Alignment: Justify the text for a clean and professional appearance.

Margins: Set margins to 1 inch on all sides (Top, Bottom, Left, Right).

Page Numbering: Page numbers should be placed at the bottom center of the page and should start from the Introduction section.

2. Title Page

Title: Center-aligned, bold, and in 16 pt font.

Student Details: Below the title, include the student name(s) and roll number(s) in 14 pt font.

Course Title & Code: Place this below the student details in 14 pt font.

Instructor Name: Include this below the course title in 14 pt font.

Submission Date: Positioned at the bottom center of the title page, in 12 pt font.

3. Headings and Subheadings

Main Headings (e.g., Introduction, Methodology):

Font Size: 14 pt

Bold and numbered (e.g., 1. Introduction)

Left-aligned

Subheadings (e.g., 3.1 Data Collection):

Font Size: 12 pt

Bold and numbered in hierarchical order (e.g., 3.1, 3.2)

Left-aligned

4. Figures and Tables

Figures: Center-aligned on the page.

Caption should be placed below the figure in 10 pt font (e.g., Figure 1: Data Processing Workflow).

Number figures sequentially (e.g., Figure 1, Figure 2).

Tables: Center-aligned on the page.

Caption should be placed above the table in 10 pt font (e.g., Table 1: Summary of Dataset).

Number tables sequentially (e.g., Table 1, Table 2).

Text Referencing: Refer to figures and tables in the text (e.g., "As shown in Figure 2...").

5. Equations

Font: Times New Roman, 12 pt.

Alignment: Center-aligned.

Equation Numbering: Right-aligned in parentheses (e.g., Eq. 1).

Referencing: Mention equations in the text (e.g., "As seen in Eq. 1...").

6. Citations and References

Citation Style: Use a standard citation style, such as APA, IEEE, or Harvard.

In-text Citations:

APA/Harvard: Use author-date format (e.g., (Smith, 2020)).

IEEE: Use numerical superscript (e.g., [1]).

Reference List:

Title: “References,” centered, bold, and in 14 pt font.

Font Size: 12 pt.

Spacing: Single line, with a hanging indent of 0.5 inches.

Format:

List references in alphabetical order (APA/Harvard).

List references in the order of appearance (IEEE).

7. Appendices

Appendix Heading: Format as “Appendix A: Title” in 14 pt font, bold, and centered.

Content: Use the same formatting as the main text (Times New Roman, 12 pt, 1.5 line spacing).

Referencing: Refer to appendices in the main text (e.g., "See Appendix A for details...").

8. Document Structure

Section Breaks: Use page breaks between major sections (e.g., between Literature Review and Methodology).

Headers and Footers:

Header: Include the project title, left-aligned.

Footer: Include your name, right-aligned.

9. Binding and Submission

Binding: Use spiral or soft binding with a transparent front cover and a black back cover.

Submission: Submit one hard copy and one digital copy (PDF) as per the course requirements.

Front Sheet of Mini Project Report:

REVA UNIVERSITY
Bengaluru, India

School of Computer Science and Engineering

Course Code: M22TC0207	**Big Data And Analytics Lab Project Report**	**Academic Year: 2024-25** **Semester & Batch:**
Project Details:		
Project Title:		
Place of Project:	**REVA UNIVERSITY, BENGALURU**	
Student Details:		
Name:		**Sign:**
Mobile No:		
Email-ID:		
SRN:		
Guide and Lab Faculty Members Details		
Guide Name: (The Faculty Member Assigned)		**Sign:** **Date:**
Grade by Guide:		
Name of Lab Co-Faculty 1		**Sign:** **Date:**
Name of Lab Co-Faculty 2		**Sign:** **Date:**

Grade by Lab Faculty Members (combined)		
SEE Examiners		
Name of Examiner 1:		**Sign:** **Date:**
Name of Examiner 2:		**Sign:** **Date:**

Contents

1. Title Page Pg no
2. Abstract Pg no
3. Introduction Pg no
4. Literature Review Pg no
5. Methodology Pg no
6. Results and Discussion Pg no
7. Conclusions Pg no
8. References Pg no
9. Appendices Pg no

1. Title Page

Project Title

Student Name(s) and Roll Number(s)

Course Title and Code

Instructor Name

Submission Date

2. Abstract

A concise summary of the project, including the problem statement, methodology, key results, and conclusions.

3. Introduction

Background and significance of the problem.

Objectives of the project.

Overview of the report structure.

4. Literature Review

Summary of existing work related to the problem.

Discussion on the relevance of PySpark and big data analytics in solving the problem.

5. Methodology

Detailed description of the datasets used.

Data preprocessing steps.

Explanation of the PySpark implementation, including data processing, analysis, and machine learning algorithms applied.

Design and development of the real-time analytics dashboard.

6. Results and Discussion

Presentation of key findings from the data analysis and machine learning models.

Discussion on the performance of the PySpark application.

Interpretation of real-time analytics results.

Challenges faced and how they were addressed.

7. Conclusion

Summary of the project outcomes.

Reflections on the learning experience.

Suggestions for future work or improvements.

8. References

List all sources cited in the report, following the chosen citation style (e.g., APA, IEEE).

9. Appendices

Include additional material such as code snippets, detailed datasets, or supplementary charts and graphs.

12. Evaluation Rubrics

1. **Rubrics for Lab Cycle Programs' Performance in Internal and External Examination:**
   - **Internal Assessment Rubrics Finalized:** The total internal marks were divided into weekly performance, record writing, mini-project, and final internal assessment, with clear criteria for each.
   - **External Assessment Rubrics Finalized:** The total external marks were divided into lab cycle programs and mini-projects, with detailed breakdowns for write-up, execution, and viva.
2. **Rubrics for Mini-projects of Students in Internal and External Examination:**
   - **Internal Mini-project Evaluation:** Specific marks were allocated for the mini-project, including presentation, demonstration, and report.
   - **External Mini-project Evaluation:** The mini-project assessment in external exams was broken down into summary write-up, PPT and demonstration, and viva, ensuring a comprehensive evaluation.

These decisions were made to standardize the evaluation process, ensure consistency, and provide a clear framework for students to understand how their performance will be assessed.

RUBRICS:

Weekly Performance (in proportion with Attendance i.e., Continuous assessment)

PART-A

- Explanation of code line by line(3M)
- Modification incorporated in the code(2M)
- Documentation (Observation & Record) with Comments(3M)
- Viva Voce(2M)
- **Total(10M)**

PART-B

Evaluation Criteria	Description	Marks Allocation
1. Problem Understanding	Clarity in understanding the problem statement and objectives of the experiment/project.	1
2. Experimental Design	Appropriateness of the experimental setup, including selection of tools (e.g., PySpark, Hadoop).	1
3. Implementation	Accuracy and efficiency in executing the experiment/project using relevant Big Data tools.	2
4. Data Handling	Effectiveness in handling, processing, and analyzing large datasets.	1
5. Results & Analysis	Quality of results obtained, including correctness, relevance, and depth of analysis.	2
6. Documentation	Completeness and clarity of the report, including code, data, and explanation of results.	1

7. Teamwork & Collaboration	Ability to work collaboratively within a team, if applicable, and contribute effectively.	1
8. Innovation & Creativity	Application of innovative approaches or creative problem-solving in the experiment/project.	1
Total Marks		10

Rubrics for Internal Assessment

IA-1: PART-A

- Write-up(7M)
- Execution(8M)
- Continuous Assessment Marks(5M)
- Viva voce(5M)
- **Total(25M)**

IA-2: PART-B (Mini Project)

- Write-up (2M)
- Execution (5M)
- Viva-Voce (3M)
- Weekly Performance (in proportion with Attendance i.e., CA) (10M)
- Record Writing (in proportion with Attendance) (3M)
- Mini-project (with PPT, Demo, and Report) (2M)
- **Total(25M)**

The final IA mark is the average of both IA.

Rubrics for SEE

Program Number	Parameters considered	Marks Distribution
PART-A: **Programs 1 to 7**	Write-up(Step by Step)	**7**
	Execution and output	**8**
	Modification in the program	**5**
PART-B: **Mini Project**	Summary Write-up-Problem Statement, Modules, Conclusion	**7**
	Demonstration of Project (PPT)	**8**
	Submission of Project Report	**5**
Viva Voce (Part-A)		**5**
Viva Voce (Part-B)		**5**
TOTAL		**50**

Note:

The students are expected to implement the Mini-project. But, present and submit the project report based on the approval of the Guide (Faculty Member Assigned to a student).

13. Learning Resources:

Text Books:

1. Feng, Wenqiang. "Learning Apache Spark with Python." (2019): 231.
2. Mengle, Saket SR, and Maximo Gurmendez. Mastering machine learning on Aws: advanced machine learning in Python using SageMaker, Apache Spark, and TensorFlow. Packt Publishing Ltd, 2019.

Reference Book:

1. Michael Minelli, Michele chambers, AmbigaDhiraj,"Big data, big analytics", Wiley, 2013.

JOURNALS/MAGAZINES

1. IEEE, Introduction to the IEEE Transactions on Big Data.
2. Elsevier, Big data research journal Elsevier.
3. Springer, Journal on Big Data Springer.