**Experiment No.: 7**

* **Pyspark Definition (Apache Pyspark) and difference between Pyspark, Scala, pandas**
* **Pyspark files and class methods**
* **get (file name)**
* **get root directory ()**

**Definition of Apache PySpark**

**Apache PySpark** is the Python API for **Apache Spark**, which is an open-source, distributed computing system used for big data processing and analytics. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. PySpark allows Python developers to use the features of **Apache Spark** (like RDDs, DataFrames, and Machine Learning capabilities) for processing large-scale data across distributed computing environments.

PySpark integrates Python with the power of Spark, making it easier to work with distributed data processing. It provides a set of APIs that allow users to interact with the Spark core, SQL, machine learning (MLlib), and graph processing (GraphX) in Python.

PySpark is used for:

* **Distributed computing**: Processing large datasets across multiple machines in a cluster.
* **Big Data Analytics**: Processing and analyzing large amounts of data in real-time or batch mode.
* **Machine Learning**: Using the MLlib library for scalable machine learning algorithms.
* **Graph Processing**: Using GraphX for analyzing graph structures like social networks.

**PySpark vs Scala vs Pandas**

Let’s break down the differences between **PySpark**, **Scala**, and **Pandas** in terms of usage, performance, and features:

**1. PySpark (Python API for Apache Spark)**

* **Language**: Python
* **Data Processing**: PySpark is used for large-scale data processing and analytics on distributed clusters.
* **Performance**: PySpark runs on **Apache Spark**, which is optimized for distributed processing. PySpark is suitable for handling large datasets that don't fit in memory, as it can process data across a cluster of machines.
* **API**: PySpark offers APIs for working with **RDDs**, **DataFrames**, **Machine Learning** (MLlib), **SQL** (Spark SQL), and more.
* **Use Cases**: Big data analytics, machine learning at scale, distributed data processing, ETL tasks.
* **Data Model**: Works on **distributed datasets** (RDDs, DataFrames, and Datasets).

**2. Scala**

* **Language**: Scala is a JVM-based language that is native to Apache Spark.
* **Data Processing**: Scala is Spark's original language, and it is the most efficient in terms of performance, as Spark's core is written in Scala.
* **Performance**: Scala provides better performance than PySpark because it directly communicates with Spark’s core without the need for a Python-JVM bridge. It's highly efficient for Spark jobs and has low overhead.
* **API**: Scala provides access to **RDDs**, **DataFrames**, and **Datasets**, and the language has the full capability of interacting with Spark’s libraries.
* **Use Cases**: Suitable for low-latency, high-performance applications where distributed computing is needed at scale.
* **Data Model**: Similar to PySpark, Scala works with **distributed datasets**, but it provides more control and performance.

**3. Pandas**

* **Language**: Python
* **Data Processing**: Pandas is a Python library used for working with **in-memory datasets**. It is great for small to medium-sized datasets that fit into memory and allows for **data wrangling**, cleaning, and transformation in a local, single-node environment.
* **Performance**: Pandas operates on **single-machine** and is not distributed, which means it is suitable only for datasets that fit into memory. For larger datasets that don’t fit into memory, you would need tools like PySpark.
* **API**: Pandas provides a powerful API for working with **DataFrames**, similar to SQL, and supports operations like aggregation, filtering, reshaping, and visualization.
* **Use Cases**: Data analysis, data cleaning, and exploratory data analysis (EDA) for datasets that fit in memory.
* **Data Model**: Pandas operates with **single-node, in-memory data structures** (like DataFrames) and is optimized for non-distributed environments.

**Conclusion**

* **PySpark** is great for **big data processing** and distributed analytics across clusters, especially for Python developers. It is slower than **Scala** but much faster and more scalable than **Pandas** for large datasets.
* **Scala** is ideal for those seeking the best **performance** and full control over Spark's features, as it interacts directly with Spark’s core. It's best for high-performance, low-latency applications at scale.
* **Pandas** is best suited for **in-memory data processing** for small-to-medium datasets, where high performance and distribution are not necessary. It’s a favorite for **data manipulation**, **cleaning**, and **exploratory data analysis**.

Choose **PySpark** if you need to work with big data and **Spark**. Opt for **Scala** if you require **optimal performance** and control. Use **Pandas** if your dataset is manageable in memory and you need to perform in-depth analysis on a single machine.

1. **PySpark Script File Example**:

# Import necessary libraries

from pyspark.sql import SparkSession

# Initialize Spark Session

spark = SparkSession.builder.appName('PySpark Example').getOrCreate()

# Load data

df = spark.read.csv('path\_to\_your\_data.csv', header=True, inferSchema=True)

# Show the first few rows of the dataframe

df.show()

# Perform transformations and actions

df\_filtered = df.filter(df['age'] > 30)

# Collect the result

result = df\_filtered.collect()

print(result)

# Stop the Spark session

spark.stop()

This is a basic PySpark script where we:

* **Initialize a Spark session**: SparkSession is the entry point for working with PySpark.
* **Load data** into a DataFrame from a CSV file.
* **Perform transformations** (like filtering) and **actions** (like collect()).
* **Stop the Spark session** after completing the tasks.

PySpark provides a vast set of methods for handling large-scale data processing and machine learning workflows in a distributed manner. Understanding these classes and methods will allow you to effectively work with big data in PySpark.

**get (file name) in PySpark**

In PySpark, if you want to get the file name from the data being processed, you can typically use **input\_file\_name()** function, which is part of **pyspark.sql.functions**. This function returns the name of the file from which the current row was read. It’s often used with DataFrames to track the source of each row of data in a distributed environment.

### **Example: Using** input\_file\_name() **to Get the File Name**

Let's assume you have multiple CSV files and you want to know the file name from which each row has been read.

from pyspark.sql import SparkSession

from pyspark.sql.functions import input\_file\_name

# Initialize Spark Session

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

# Read data from multiple CSV files

df = spark.read.csv("path\_to\_directory/\*.csv", header=True, inferSchema=True)

# Add a new column that contains the file name of each row

df\_with\_file\_name = df.withColumn("file\_name", input\_file\_name())

# Show the results

df\_with\_file\_name.show(truncate=False)

# Stop the Spark session

spark.stop()

**Explanation:**

* **spark.read.csv()**: Reads all CSV files from the specified directory or file pattern (path\_to\_directory/\*.csv).
* **input\_file\_name()**: Adds a new column to the DataFrame, which contains the full path of the file for each row.
* **df.withColumn("file\_name", input\_file\_name())**: Creates a new column named "file\_name" and applies the input\_file\_name() function.

**Sample Output:**

Assuming you have two files data1.csv and data2.csv, and their contents are as follows:

* data1.csv:

name,age

Alice,30

Bob,25

data2.csv:

name,age

Charlie,35

David,40

The output of the show() method will look like this:

|  |  |  |
| --- | --- | --- |
| **name** | **age** | **file\_name** |
| Alice | 30 | file:/path/to/data1.csv |
| Bob | 25 | file:/path/to/data1.csv |
| Charlie | 35 | file:/path/to/data2.csv |
| David | 40 | file:/path/to/data2.csv |

This allows you to track the origin of each row in your DataFrame.

**Summary:**

* **input\_file\_name()** is a PySpark function that returns the file path of the file from which the data row was read.
* It’s helpful when you want to track or tag the source file of each row in a large-scale data processing task.

**PySpark get root directory ()**

In PySpark, there isn't a direct function to get the root directory of the Spark application or the filesystem. However, you can determine the working directory or root directory based on the context of your environment.

There are a couple of common approaches to getting the root directory in PySpark, depending on your specific use case:

### **1. Using Spark Context for the Working Directory**

If you're looking to get the working directory of your current Spark session, you can use the SparkContext or the system environment. For example:

from pyspark.sql import SparkSession

# Initialize SparkSession

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

# Get the SparkContext from the SparkSession

sc = spark.sparkContext

# Get the working directory (local machine's current directory or driver node's path)

print("Spark working directory: ", sc.\_jsc.hadoopConfiguration().get("spark.local.dir"))

# Stop the Spark session

spark.stop()

**Explanation:**

* **sc.\_jsc.hadoopConfiguration().get("spark.local.dir")**: This gives the directory where Spark stores temporary data. It could be the working directory on the local file system or a directory on the cluster's nodes.

Note: This gives you a path on the local node, not necessarily the root directory of your file system (HDFS or other distributed systems).