pyspark Practice Notes

content:-

Spark Session -

SparkSession is designed to be a singleton, which means that only one instance should be active in the application at any given time.

```
spark2 = SparkSession.newSession

print(spark2)

spark = SparkSession.builder \

    .appName("My PySpark Application") \

    .master("local[*]") \

    .config("spark.executor.memory", "4g") \

    .config("spark.executor.cores", "4") \

    .config("spark.driver.memory", "2g") \

    .getOrCreate()
```

What is SparkSession – PySpark Entry Point, Dive into SparkSession

Jagdeesh

Introduction

PySpark, the Python library for Apache Spark, has gained immense popularity among data engineers and data scientists due to its simplicity and power in handling big data tasks.

This blog post will provide a comprehensive understanding of the PySpark entry point, the SparkSession. We'll explore the concepts, features, and the use of SparkSession to set up a PySpark application effectively.

What is SparkSession?

SparkSession is the entry point for any PySpark application, introduced in Spark 2.0 as a unified API to replace the need for separate SparkContext, SQLContext, and HiveContext.

The SparkSession is responsible for coordinating various Spark functionalities and provides a simple way to interact with structured and semi-structured data, such as reading and writing data from various formats, executing SQL queries, and utilizing built-in functions for data manipulation.

SparkSession offers several benefits that make it an essential component of PySpark applications:

Simplified API: SparkSession unifies the APIs of SparkContext, SQLContext, and HiveContext, making it easier for developers to interact with Spark's core features without switching between multiple contexts.

Configuration management: You can easily configure a SparkSession by setting various options, such as the application name, the master URL, and other configurations.

Access to Spark ecosystem: SparkSession allows you to interact with the broader Spark ecosystem, such as DataFrames, Datasets, and MLlib, enabling you to build powerful data processing pipelines.

Improved code readability: By encapsulating multiple Spark contexts, SparkSession helps you write cleaner and more maintainable code.

Creating a SparkSession

To create a SparkSession, we first need to import the necessary PySpark modules and classes. Here's a simple example:

```
from pyspark.sql import SparkSession
```

```python
spark = SparkSession.builder \
    .appName("My PySpark Application") \
    .master("local[*]") \
    .getOrCreate()
```

In this example, we import the SparkSession class from the pyspark.sql module and use the builder method to configure the application name and master URL. The getOrCreate() method is then used to either get the existing SparkSession or create a new one if none exists.

The SparkSession.builder object provides various functions to configure the SparkSession before creating it. Some of the important functions are:

appName(name): Sets the application name, which will be displayed in the Spark web user interface.

master(masterURL): Sets the URL of the cluster manager (like YARN, Mesos, or standalone) that Spark will connect to. You can also set it to “local” or “local[N]” (where N is the number of cores) for running Spark locally.

config(key, value): Sets a configuration property with the specified key and value. You can use this method multiple times to set multiple configuration properties.

config(conf): Sets the Spark configuration object (SparkConf) to be used for building the SparkSession.

enableHiveSupport(): Enables Hive support, including connectivity to a persistent Hive metastore, support for Hive SerDes, and Hive user-defined functions (UDFs).

getOrCreate(): Retrieves an existing SparkSession or, if there is none, creates a new one based on the options set via the builder.

How many pyspark sessions can be created?

In PySpark, you can technically create multiple SparkSession instances, but it is not recommended. The standard practice is to use a single SparkSession per application.

SparkSession is designed to be a singleton, which means that only one instance should be active in the application at any given time

```
# Create new SparkSession

spark2 = SparkSession.newSession

print(spark2)
```

Configuring a SparkSession

You can configure a SparkSession with various settings, such as the number of executor cores, executor memory, driver memory, and more. Here's an example

```
spark = SparkSession.builder \
    .appName("My PySpark Application") \
    .master("local[*]") \
    .config("spark.executor.memory", "4g") \
    .config("spark.executor.cores", "4") \
    .config("spark.driver.memory", "2g") \
    .getOrCreate()
```

In this example, we've added three additional configurations for executor memory, executor cores, and driver memory using the config() method.

Accessing SparkSession Components

 # Access SparkContext

spark_context = spark.sparkContext

 # Access SQLContext

sql_context = spark._wrapped

 # Read a CSV file

data_frame = spark.read.csv("path/to/your/csv-file", header=True, inferSchema=True)

 # Write the data to a Parquet file

data_frame.write.parquet("path/to/output/parquet-file")

Executing SQL Queries with SparkSession

With SparkSession, you can also execute SQL queries directly on your data. Here's an example:

Register a DataFrame as a temporary table

```
data_frame.createOrReplaceTempView("my_table")


# Execute an SQL query on the temporary table
result = spark.sql("SELECT * FROM my_table WHERE age > 30")


# Display the result
result.show()
```

Example: Word Count with SparkSession

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import split, explode, col
# creating a spark session
spark = SparkSession.builder \
        .appName("Counting ")
        .master("local[*]") \
        .getOrCreate()
# read the input file
data = spark.read.txt("txt file")


# Split the line into words
words = data.select(explode(split(col("Value", " ")).alias("Word")
```

perform the word count

word_count = words.groupBy("Word").count()

Display the results

word_count.show()

spark.stop()

Note -

explode - EXPLODE is a PySpark function used to works over columns in PySpark.

EXPLODE is used for the analysis of nested column data.

PySpark EXPLODE converts the Array of Array Columns to row.

EXPLODE can be flattened up post analysis using the flatten method.

EXPLODE returns type is generally a new row for each element given.

explode() will return each and every individual value from an array. If the array is empty or null, it will ignore and go to the next array in an array type column in PySpark DataFrame. This is possible using the select() method. Inside this method, we can use the array_min() function and return the result.

<u>Read and Write files using PySpark – Multiple ways to Read and Write data using PySpark</u>

pip install findspark

pip install pyspark

import findspark

findspark.init()

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \

            .appName("My name is ") \

            .master("Local[*]") \

            .getOrCreate()
```

Creating a Pyspark dataFrame

```
data = [("Alice", 34), ("Bob" , 45), ("Cathy", 29)]

columns = ["Name" "Roll_Number"]

df = spark.createDataFrame(data, columns)

df.show()
```

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

| Name|Age|

+-----+---+

|Alice| 34|

|  Bob| 45|

|Cathy| 29|
```

```
csv_file = "path/to/your/csv/file.csv"
```

df_csv = spark.read.csv(csv_file, header=True, inferSchema=True)

output_path = "path/to/output/csv/file.csv"

df_csv.write.csv(output_path, header=True, mode="overwrite")

json_file = "path/to/your/json/file.json"

df_json = spark.read.json(json_file)

output_path = "path/to/output/json/file.json"

df_json.write.json(output_path, mode="overwrite")

Reading and Writing Parquet Files

To read a Parquet file using PySpark, you can use the read.parquet() method:

parquet_file = "path/to/your/parquet/file.parquet"

df_parquet = spark.read.parquet(parquet_file)

To write the data back to a Parquet file, use the write.parquet() method:

output_path = "path/to/output/parquet/file.parquet"

df_parquet.write.parquet(output_path, mode="overwrite")

```
data [
{"Name":"Alice", "Age":30, "City":"New York"},
{"Name":"Bob", "Age":25, "City":"San Francisco"},
{"Name":"Charlie", "Age":35, "City":"Los Angeles"}]
df = spark.createDataFrame(data)
df.createOrReplaceTempView("People")

query = "select * from People where Age > 30"
result_df = spark.sql(query)
result_df.show()
```

Convert Pandas Dataframe to PySpark Dataframe

```
import pandas as pd

data [
{"Name":"Alice", "Age":30, "City":"New York"},
{"Name":"Bob", "Age":25, "City":"San Francisco"},
{"Name":"Charlie", "Age":35, "City":"Los Angeles"}]

pandasData = pd.dataFrame(data, columns = ["Name", "Age", "City"]
```

```
# creating the pyspark dataframe from pandas

sparkdf = spark.createDataFrame(pandasData)

sparkdf.printSchema()

sparkdf.show()
```

```
root

 |-- name: string (nullable = true)

 |-- age: long (nullable = true)

 |-- city: string (nullable = true)


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

|   name|age|         city|

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

|  Alice| 30|     New York|

|    Bob| 25|San Francisco|

|Charlie| 35|  Los Angeles|
```


A DataFrame is a distributed collection of data organized into named columns.

It is conceptually equivalent to a table in a relational database or a data frame in Python, but optimized for large-scale processing. DataFrame as a spreadsheet with rows and columns.

The show() function is a method available for DataFrames in PySpark. It is used to display the contents of a DataFrame in a tabular format, making it easier to visualize and understand the data.

Syntax

DataFrame.show(n=20, truncate=True, vertical=False)

Parameters:

n: The number of rows to display. The default value is 20.

truncate: If set to True, the column content will be truncated if it is too long. The default value is True.

vertical: If set to True, the output will be displayed vertically. The default value is False.

Example 1-

```
import findaspark

findspark.init()

from pyspark.sql import SparkSession

spark = SparkSession.builder \

        .appName \

        .getOrCreate()

data = [("Alice":34 ), ("Bob":45), ("Charlie":29), ("David": 31)]

columns = ["Name", "Age"]


df = spark.createDataFrame(data, columns)

df.show()
```

```
+-------+---+
|   Name|Age|
+-------+---+
|  Alice| 34|
|    Bob| 45|
|Charlie| 29|
|  David| 31|
```

Display the Specific number of row. like 2

```
df.show(2)
+-----+---+
| Name|Age|
+-----+---+
|Alice| 34|
|  Bob| 45|
+-----+---+
only showing top 2 rows
```

Display Contents Without Truncation

```
df.show(truncate = False)
Name   |Age|
+-------+---+
|Alice  |34 |
```

|Bob |45 |

|Charlie|29 |

|David |31 |

Display Contents Vertically

df.show(vertical=True)

-RECORD 0-------

Name | Alice

Age | 34

-RECORD 1-------

Name | Bob

Age | 45

-RECORD 2-------

Name | Charlie

Age | 29

-RECORD 3-------

Name | David

Age | 31

Run SQL Queries with PySpark

OrderIDProductID Quantity Price OrderDate

1	101	3	100	2023-01-01
2	102	1	200	2023-01-02
3	101	2	100	2023-01-03
4	103	5	50	2023-01-04

name of file = sales data

```
csv_file = sales_data.csv
df = spark.read.option("header":"true").option("inferSchema":"true").csv(csv_files)

# creatting a temporary View

df.createOrReplaceTempView("Saless_data)
```

Calculating the total Revenue of each product

```
query = """
select ProductID,
SUM (Quantity * Price) AS totalRevenue
from saless_data
group by ProductID
"""
result = spark.sql(query)
result.show()
```

```
+---------+------------+
|ProductID|TotalRevenue|
+---------+------------+
| 101| 500|
| 102| 200|
| 103| 250|
+---------+--
```

to find the top 2 products with the highest revenue

```
query = """
select
ProductID,
SUM (Quantity * Price) AS totalRevenue
from saless_data
group by ProductID
order by totalRevenue
limit 2; """
resultDF = spark.sql(query)
resultDF.show()
```

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

```
|ProductID|TotalRevenue|
+---------+------------+
|      101|         500|
|      102|         200|
```

PySpark Pandas API

```
import pandas as pd
import numpy as np
from pyspark.sql import SparkSession
import databricks.koalas as ks

# creatting the sparkSession
spark = SparkSession.builder \
        .appName \
        .getOrCreate()
```

reading the file

```
sales_data = ks.read_csv("sales_data.csv")
```

calculate the average revenue per unit sold and add it as a new column

```
sales_data['Avg_Revenue_per_Unit] =

sales_data['Revenue'] / sales_data['Unit_Sold']
```

Computing summary statistics

```
summary_stats = sales_data.groupby(['Store_ID', 'Product_ID']).agg(

            {'Revenue': 'sum', 'Units_Sold': 'sum'}).reset_index()
```

Sorting the results

```
sorted_summary_stats = summary_stats.sort_values(

   by=['Store_ID', 'Revenue'], ascending=[True, False])
```

Exporting the results

```
sorted_summary_stats.to_csv("summary_stats.csv", index=False)


spark.stop()      # for stoping that current session.
```

Select columns in PySpark dataframe

```
import findspark

findspark.init()

spark = SparkSession.builder \

          .appName \

          .master \

          .getOrCreate()
```

```
data = [("Alice",  34, "Female"), ("Bob":45, "Male"),  ("Charlie":28, "Male"),
("Diana", 39, "Female")]
columns = ["Name", "Age", "Gender"]
df = spark.createDateFrame(data, columns)
df.show()
  Name|Age|Gender|
+-------+---+------+
|  Alice| 34|Female|
|    Bob| 45|  Male|
|Charlie| 28|  Male|
|  Diana| 39|Female|
+-------+---+------+
# Select columns using column names
selectdf1 = df.select("Name", "Age")
selectdf1.show()
|   Name|Age|
+-------+---+
|  Alice| 34|
|    Bob| 45|
|Charlie| 28|
|  Diana| 39|
# select column using the col functions
selectdf2 = df.select (col("name"), col("Age"))
```

```
selectdf2.show()

 Name|Age|
+-------+---+
|  Alice| 34|
|    Bob| 45|
|Charlie| 28|
|  Diana| 39
```

Selecting Columns using the '[ ]' Operator

```
# Select a single column using the '[]' operator
name_df = df["Name"]


# Select multiple columns using the '[]' operator
selected_df3 = df.select(df["Name"], df["Age"])
selected_df3.show()

 Name|Age|
+-------+---+
|  Alice| 34|
|    Bob| 45|
|Charlie| 28|
|  Diana| 39|
```

Select Columns using index

```
# Define the column indices you want to select
column_indices = [0, 2]

# Extract column names based on indices
selected_columns = [df.columns[i] for i in column_indices]

# Select columns using extracted column names
selected_df4 = df.select(selected_columns)
# Show the result DataFrame
selected_df4.show()
```

```
 Name|Gender|
+--------+------+
|  Alice|Female|
|    Bob|  Male|
|Charlie|  Male|
|  Diana|Female|
```

Selecting Columns using the ‘withColumn’ and ‘drop’ Functions.

select specific columns while adding or removing columns, you can use the 'withColumn' function to add a new column and the 'drop' function to remove a column.

Add a new column 'IsAdult' and remove the 'Gender' column

```
selectDF = df.withColumn("IsAdult", col("Age") > 18).drop("Gender")
selectDF.show()
```

```
 Name|Age|IsAdult|
+-------+---+-------+
|  Alice| 34|   true|
|    Bob| 45|   true|
|Charlie| 28|   true|
|  Diana| 39|   true|
```

Selecting Columns using SQL Expressions

select columns using the 'selectExpr' function. This is useful when you want to perform operations on columns while selecting them.

```
selectDF = df.selectExpr("Name", "Age", "Age >= 18  AS IsAdult")
selectDF.show()
```

```
 Name|Age|IsAdult|
```

```
+-------+---+-------+
|  Alice| 34|   true|
|    Bob| 45|   true|
|Charlie| 28|   true|
|  Diana| 39|   true|
```

PySpark withColumn

The “withColumn” function in PySpark allows to add, replace, or update columns in a DataFrame. It is a DataFrame transformation operation, meaning it returns a new DataFrame with the specified changes, without altering the original DataFrame.

The “withColumn” function is particularly useful when we eed to perform column-based operations like renaming, changing the data type, or applying a function to the values in a column.

Syntax

```
DataFrame.withColumn(colName, col)
```

where:

DataFrame: The original PySpark DataFrame you want to manipulate.

colName: The name of the new or existing column you want to add, replace, or update.

col: The new expression or value for the specified column.

```
import findspark

findspark.init()

from pyspark.sql import SparkSession

from pyspark.sql.functions import col

# initialization the sparksession


spark = SparkSession.builder \

            .appName \

            .getOrCreate()

data = [(1, "Alice", 25),

(2, "Bob", 30),

(3, "Charlie", 35)

]

columns = ["id", "name", "age"]

# create a Dataframe

df = spark.createDataFrame(data, columns)

df.show()
```

```
id|   name|age|

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

|  1|  Alice| 25|

|  2|    Bob| 30|

|  3|Charlie| 35|
```

1. Remaining the column.

"withColumns" to rename the "age" column to "years"

rename the age column to years

```
df = df.withColumn("Years", col("age"))
```

drop the column age

```
df = df.drop("Age")
df.show()
```

```
 id|   name|years|
+---+-------+-----+
|  1|  Alice|   25|
|  2|    Bob|   30|
|  3|Charlie|   35|
+---+-------+-----
```

2. Applying a function to a column

Converting the age from years to months

```
from pyspark.sql.functions import expr

df = df.withClumn("months", expr("years * 12")

df.show()
```

```
id|   name|years|months|

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

| 1|  Alice|   25|   300|

| 2|    Bob|   30|   360|

| 3|Charlie|   35|   420|
```

3. Change the column data type

```
from pyspark.sql.types import StringType

# change the id to string

df = df.withColumn("id", col("id").cast(StringType()))

df.show()
```

```
id|   name|years|months|

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

| 1|  Alice|   25|   300|

| 2|    Bob|   30|   360|

| 3|Charlie|   35|   420
```

4. Conditional column update with "withColumn".

```
from pyspark.sql.functions import when

data = [(1, "Alice", 25, 45000), (2, "Bob", 30, 55000), (3, "Charlie", 35, 60000)

]

columns = ["id", "name", "age", "salary"]

df = spark.createDataFrame(data, columns)

df.show()

# add the "tax" column based on the "salary".

df = df.withColumn("tax", when(col("salary")>=50000, col("salary")*0.1).otherwise(col("salary")
* 0.05


df.show()

id|   name|age|salary|

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

|  1|  Alice| 25| 45000|

|  2|    Bob| 30| 55000|

|  3|Charlie| 35| 60000|

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


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

| id|   name|age|salary|   tax|

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

|  1|  Alice| 25| 45000|2250.0|

|  2|    Bob| 30| 55000|5500.0|

|  3|Charlie| 35| 60000|6000.0|
```

5. Using the USer Defined Function with withColumn.

```
from pyspark.sql.functions import udf

from pyspark.sql.types import StringType


def age_group(age):

    if age < 30:

        return "Young"

    elif age < 45:

        return "MiddleAged"

    else:

        return "Old"


# Register the UDF

age_group_udf = udf(age_group, Stringtype())

# add the new columnn age_group based on the age

df = df.withColumn("age_group", age_group_udf(col("age")))

df.show()
```

```
id|   name|age|salary|   tax|  age_group|

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

|  1|  Alice| 25| 45000|2250.0|      Young|

|  2|    Bob| 30| 55000|5500.0|Middle-aged|

|  3|Charlie| 35| 60000|6000.0|Middle-aged|
```

creating the new column net_sales column based on the subtracting the tax column from salary

df = df.withColumn("net_sales", round(col("salary")- col("tax"), 2))

df.show()

```
id|   name|age|salary|   tax|  age_group|net_salary|
+---+-------+---+------+------+-----------+----------+
|  1|  Alice| 25| 45000|2250.0|      Young|   42750.0|
|  2|    Bob| 30| 55000|5500.0|Middle-aged|   49500.0|
|  3|Charlie| 35| 60000|6000.0|Middle-aged|   54000.0|
```

combining the multiple column into one column.

name and age_group column = name_age_group

We will use the "concat_ws" function, which allows us to concatenate multiple columns with a specified delimiter.

from pyspark.sql.functions import concat_ws

combining the column

df = df.withColumn("name_age_group", concat_ws("_", col("name"), col("age_group")))

df.show()

```
id|   name|age|salary|   tax|  age_group|net_salary|      name_age_group|
+---+-------+---+------+------+-----------+----------+--------------------+
|  1|  Alice| 25| 45000|2250.0|      Young|   42750.0|       Alice - Young|
|  2|    Bob| 30| 55000|5500.0|Middle-aged|   49500.0|   Bob - Middle-aged|
|  3|Charlie| 35| 60000|6000.0|Middle-aged|   54000.0|Charlie - Middle-...|
```

PySpark Drop Columns

Drop() function to remove columns from a DataFrame

Creating the dataframe

```
import findspark
findspark.init()
from pyspark.sql import SparkSession
spark = SparkSession.builder \
            .appName \
            .getOrCreate()

data = [("Alice", 30, "New York", "F"),
("Bob", 28, "San Francisco", "M"),
("Cathy", 29, "Los Angeles", "F"),
("David", 32 "Chicago", "M")
]
```

```
columns= ["name", "age", "city", "gender"]

df = df.createDataFrame(data, columns)

df.show()
```

```
name|age|         city|gender|
+-----+---+-------------+------+
|Alice| 30|     New York|     F|
|  Bob| 28|San Francisco|     M|
|Cathy| 29|  Los Angeles|     F|
|David| 32|      Chicago|     M|
```

Different ways to drop columns in PySpark DataFrame

Dropping a Single Column

The Drop() function can be used to remove a single column from a DataFrame.

Syntax:

```
df = df.drop("gender")

df.show()

name|         city|
+-----+-------------+
|Alice|     New York|
|  Bob|San Francisco|
|Cathy|  Los Angeles|
|David|      Chicago|
```

Dropping Multiple Columns

Drop() function to remove multiple columns from a DataFrame. Simply pass a list of column names to the function.

```
df = df.drop("age", "gender")

df.show()
```

or list of column name

```
dropping_column_name = ["age" , "gender"]

df = df.drop(* dropping_column_name)

df.show()
```

```
 name|         city|
+-----+-------------+
|Alice|     New York|
|  Bob|San Francisco|
|Cathy|  Los Angeles|
|David|      Chicago
```

Dropping Columns Conditionally

```
if "gender" in df.columns:
        df = df.drop("gender")
df.show()
name|age|         city|
+-----+---+-------------+
```

```
|Alice| 30|     New York|
|  Bob| 28|San Francisco|
|Cathy| 29|  Los Angeles|
|David| 32|      Chicago|
```

Dropping Columns Using Regex Pattern

"drop()" function in combination with a regular expression (regex) pattern to drop multiple columns matching the pattern.

```
from pyspark.sql.functions import col
import re
regex_pattern = "gender|age"
df = df.select([col(c) for c in df.columns if not re.match(regex_pattern, c)])
df.show()
```

```
+-----+-------------+
| name|         city|
+-----+-------------+
|Alice|     New York|
|  Bob|San Francisco|
|Cathy|  Los Angeles|
|David|      Chicago|
```

PySpark Rename Columns

```
import findspark

findspark.init()

from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("PySpark Rename Columns").getOrCreate()

from pyspark.sql import Row

data = [Row(name="Alice", age=25, city="New York"),

    Row(name="Bob", age=30, city="San Francisco"),

    Row(name="Cathy", age=35, city="Los Angeles")]

sample_df = spark.createDataFrame(data)

sample_df.show()
```

```
 name|age|         city|
+-----+---+-------------+
|Alice| 25|     New York|
|  Bob| 30|San Francisco|
|Cathy| 35|  Los Angeles|
+-----+---+-------------+
```

Different ways to rename columns in a PySpark DataFrame

Renaming Columns Using 'withColumnRenamed'

The 'withColumnRenamed' method is a simple way to rename a single column in a DataFrame

```
renamed_df = sample_df.withColumnRenamed("age", "user_age")
```

```
renamed_df.show()
```

```
name|user_age|        city|
+-----+--------+-------------+
|Alice|      25|     New York|
|  Bob|      30|San Francisco|
|Cathy|      35|  Los Angeles
```

Renaming Columns Using ‘select’ and ‘alias’

```
from pyspark.sql.functions import col
renamed_df = sample_df.select(col("name"), col("age").alias("user_age"), col("city"))
renamed_df.show()
name|user_age|        city|
+-----+--------+-------------+
|Alice|      25|     New York|
|  Bob|      30|San Francisco|
|Cathy|      35|  Los Angeles
```

Renaming Columns Using ‘toDF’

```
renamed_df = sample_df.toDF("user_name", "user_age", "user_city")
renamed_df.show()
```

user_name|user_age| user_city|

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

| Alice| 25| New York|

| Bob| 30|San Francisco|

| Cathy| 35| Los Angeles

Renaming Multiple Columns

```
renamed_df = sample_df.withColumnRenamed("name", "user_name") \
                 .withColumnRenamed("age", "user_age") \
                 .withColumnRenamed("city", "user_city")
renamed_df.show()
```

user_name|user_age| user_city|

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

| Alice| 25| New York|

| Bob| 30|San Francisco|

| Cathy| 35| Los Angeles

PySpark Filter vs Where

```
import findspark
findspark.init()
from pyspark.sql import SparkSession
```

```
spark = SparkSession.builder \
   .appName("Filtering Rows in PySpark DataFrames") \
   .getOrCreate()

from pyspark.sql import Row
data = [
   Row(id=1, name="Alice", age=30),
   Row(id=2, name="Bob", age=25),
   Row(id=3, name="Charlie", age=35),
   Row(id=4, name="David", age=28)]
columns = ["id", "name", "age"]
df = spark.createDataFrame(data, columns)
df.show()
```

```
id|   name|age|
+---+-------+---+
|  1|  Alice| 30|
|  2|    Bob| 25|
|  3|Charlie| 35|
|  4|  David| 28
```

Different ways to filter rows in PySpark DataFrames

1. Filtering Rows Using ‘filter’ Function

It takes a boolean expression as an argument and returns a new DataFrame containing only the rows that satisfy the condition.

Example: Filter rows with age greater than 30

```
filtered_df = df.filter(df.age > 29)

filtered_df.show()


id|   name|age|

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

|  1|  Alice| 30|

|  3|Charlie| 35|
```

2. Filtering Rows Using ‘where’ Function

The where function is an alias for the ‘filter’ function and can be used interchangeably. It also takes a boolean expression as an argument and returns a new DataFrame containing only the rows that satisfy the condition.

PySpark Filter vs Where – Comprehensive Guide Filter Rows from PySpark DataFrame

Jagdeesh

Apache PySpark is a popular open-source distributed data processing engine built on top of the Apache Spark framework. It provides a high-level API for handling large-scale data processing tasks in Python, Scala, and Java.

One of the most common tasks when working with PySpark DataFrames is filtering rows based on certain conditions. In this blog post, we’ll discuss different ways to filter rows in PySpark DataFrames, along with code examples for each method.

Different ways to filter rows in PySpark DataFrames

1. Filtering Rows Using 'filter' Function

2. Filtering Rows Using 'where' Function

3. Filtering Rows Using SQL Queries

4. Combining Multiple Filter Conditions

Before we dive into filtering rows, let's quickly review some basics of PySpark DataFrames. To work with PySpark DataFrames, we first need to import the necessary modules and create a SparkSession

```
import findspark

findspark.init()


from pyspark.sql import SparkSession


spark = SparkSession.builder \

   .appName("Filtering Rows in PySpark DataFrames") \

   .getOrCreate()
```

Next, let's create a simple DataFrame to use in our examples

```
from pyspark.sql import Row


data = [

   Row(id=1, name="Alice", age=30),

   Row(id=2, name="Bob", age=25),

   Row(id=3, name="Charlie", age=35),

   Row(id=4, name="David", age=28)
```

```
]

columns = ["id", "name", "age"]

df = spark.createDataFrame(data, columns)
df.show()
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  1|  Alice| 30|
|  2|    Bob| 25|
|  3|Charlie| 35|
|  4|  David| 28|
+---+-------+---+
```

1. Filtering Rows Using 'filter' Function

The filter function is one of the most straightforward ways to filter rows in a PySpark DataFrame. It takes a boolean expression as an argument and returns a new DataFrame containing only the rows that satisfy the condition.

Example: Filter rows with age greater than 30

```
filtered_df = df.filter(df.age > 29)

filtered_df.show()
+---+-------+---+
```

```
| id|   name|age|
+---+-------+---+
|  1|  Alice| 30|
|  3|Charlie| 35|
+---+-------+---+
```

2. Filtering Rows Using ‘where’ Function

The where function is an alias for the ‘filter’ function and can be used interchangeably. It also takes a boolean expression as an argument and returns a new DataFrame containing only the rows that satisfy the condition.

Example: Filter rows with name equal to “Alice”:

```
filtered_df = df.where(df.name.isin(["Alice", "Charlie"]))

filtered_df.show()
```

```
 id|   name|age|
+---+-------+---+
|  1|  Alice| 30|
|  3|Charlie| 35|
```

3. Filtering Rows Using SQL Queries

Example: Filter rows with age less than or equal to 25

```
df.createOrReplaceTempView("people")

filtered_df = spark.sql("SELECT * FROM people WHERE age <= 25")

filtered_df.show()
```

id|name|age|

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

| 2| Bob| 25|

4. Combining Multiple Filter Conditions

combine multiple filter conditions using the ‘&’ (and), ‘|’ (or), and ‘~’ (not) operators. Make sure to use parentheses to separate different conditions, as it helps maintain the correct order of operations.

```
filtered_df = df.filter((df.age > 25) & (df.name != "David"))

filtered_df.show()
```

```
id|   name|age|

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

| 1|  Alice| 30|

| 3|Charlie| 35
```

PySpark orderBy() and sort()

```
import findspark

findspark.init()

from pyspark.sql import SparkSession

# Create a SparkSession

spark = SparkSession.builder \
```

```
    .appName("PySpark orderBy() and sort() Example") \
    .getOrCreate()
# Sample data
data = [
    ("Alice", 30, "New York"),
    ("Bob", 28, "San Francisco"),
    ("Charlie", 34, "Los Angeles"),
    ("Diana", 29, "Chicago")
]
# Create a DataFrame
columns = ["Name", "Age", "City"]
df = spark.createDataFrame(data, columns)
df.show()
```

```
 Name|Age|         City|
+-------+---+-------------+
|  Alice| 30|     New York|
|    Bob| 28|San Francisco|
|Charlie| 34|  Los Angeles|
|  Diana| 29|      Chicago
```

orderBy() function

The orderBy() function in PySpark is used to sort a DataFrame based on one or more columns. It takes one or more columns as arguments and returns a new DataFrame sorted by the specified columns.

Syntax:

DataFrame.orderBy(*cols, ascending=True)

The sort() Function

The sort() function is an alias of orderBy() and has the same functionality. The syntax and parameters are identical to orderBy().

Syntax:

DataFrame.sort(*cols, ascending=True)

Difference between orderBy() and sort()

There is no functional difference between orderBy() and sort() in PySpark. The sort() function is simply an alias for orderBy().'

```
sorted_by_age = df.orderBy("Age")
sorted_by_age.show()
 Name|Age|         City|
+-------+---+-------------+
|    Bob| 28|San Francisco|
|  Diana| 29|      Chicago|
|  Alice| 30|     New York|
|Charlie| 34|  Los Angeles|
```

Sort by multiple columns using orderBy()

```
sorted_by_age_and_city = df.orderBy(["Age", "City"], ascending=[True, False])
sorted_by_age_and_city.show()
```

```
Name|Age|         City|
+-------+---+-------------+
|    Bob| 28|San Francisco|
|  Diana| 29|      Chicago|
|  Alice| 30|     New York|
|Charlie| 34|  Los Angeles|
```

Sort the DataFrame using sort()

```
sorted_by_age = df.sort("Age")
sorted_by_age.show()
```

```
Name|Age|         City|
+-------+---+-------------+
|    Bob| 28|San Francisco|
|  Diana| 29|      Chicago|
|  Alice| 30|     New York|
|Charlie| 34|  Los Angeles
```

Sort by multiple columns using sort()

sorted_by_age_and_city = df.sort(["Age", "City"], ascending=[True, False])

sorted_by_age_and_city.show()

```
Name|Age|        City|
+-------+---+-------------+
|    Bob| 28|San Francisco|
|  Diana| 29|      Chicago|
|  Alice| 30|     New York|
|Charlie| 34|  Los Angeles|
```

Sorting a DataFrame using column expressions

from pyspark.sql.functions import desc, asc

Sort the DataFrame by age in descending order using column expressions

sorted_by_age_desc_expr = df.orderBy(desc("Age"))

sorted_by_age_desc_expr.show()

```
Name|Age|        City|
+-------+---+-------------+
|Charlie| 34|  Los Angeles|
|  Alice| 30|     New York|
|  Diana| 29|      Chicago|
```

| Bob| 28|San Francisco

Sort the DataFrame by city in ascending order using column expressions

sorted_by_city_asc_expr = df.sort(asc("City"))

sorted_by_city_asc_expr.show()

Name|Age| City|

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

| Diana| 29| Chicago|

|Charlie| 34| Los Angeles|

| Alice| 30| New York|

| Bob| 28|San Francisco|

Sorting a DataFrame with NULL values

data_with_nulls = [

("Alice", None, "New York"),

("Bob", 28, None),

("Charlie", 34, "Los Angeles"),

("Diana", 29, "Chicago")

]

Create a DataFrame with NULL values

df_with_nulls = spark.createDataFrame(data_with_nulls, columns)

Sort the DataFrame with NULL values in Age column (NULLs appear last)

sorted_with_nulls = df_with_nulls.orderBy("Age", ascending=True, nulls_last=True) sorted_with_nulls.show()

Name| Age| City|

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

| Alice|null| New York|

| Bob| 28| null|

| Diana| 29| Chicago|

|Charlie| 34|Los Angeles

Sort the DataFrame with NULL values in City column (NULLs appear first)

sorted_with_nulls_alt = df_with_nulls.sort("City", ascending=True, nulls_first=True)

sorted_with_nulls_alt.show()

Name| Age| City|

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

| Bob| 28| null|

| Diana| 29| Chicago|

|Charlie| 34|Los Angeles|

| Alice|null| New York

Sorting a DataFrame using a custom sorting order

from pyspark.sql.functions import col, when

Define a custom sorting order for cities

city_order = ["New York", "Los Angeles", "Chicago", "San Francisco"]

```
# Create a custom sorting column
custom_sort_col = when(col("City") == city_order[0], 0) \
    .when(col("City") == city_order[1], 1) \
    .when(col("City") == city_order[2], 2) \
    .when(col("City") == city_order[3], 3) \
    .otherwise(4)
# Sort the DataFrame using the custom sorting order
sorted_by_custom_order = df.orderBy(custom_sort_col)
sorted_by_custom_order.show()
```

```
Name|Age|        City|
+-------+---+-------------+
|  Alice| 30|     New York|
|Charlie| 34|  Los Angeles|
|  Diana| 29|      Chicago|
|    Bob| 28|San Francisco
```

PySpark GroupBy()

PySpark GroupBy is a powerful operation that allows you to perform aggregations on your data. It groups the rows of a DataFrame based on one or more columns and then applies an aggregation function to each group. Common aggregation functions include sum, count, mean, min, and max.

Here's a general structure of a GroupBy operation:

Syntax :

```
dataFrame.groupBy("column_name").agg(aggregation_function)
```

aggregation functions

count() – return the number of rows for each group

max() – returns the maximum of values for each group

min() – returns the minimum of values for each group

sum() – returns the total for values for each group

avg() – returns the average for values for each group

```
import findspark
findspark.init()
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
# Create a Spark session
spark = SparkSession.builder \
    .appName("PySpark GroupBy Example") \
    .getOrCreate()
# Sample data
data = [("1001", "Laptop", "Electronics", 1, 1000, "2023-01-01"),
        ("1002", "Mouse", "Electronics", 2, 50, "2023-01-02"),
        ("1003", "Laptop", "Electronics", 1, 1200, "2023-01-03"),
        ("1004", "Mouse", "Electronics", 3, 30, "2023-01-04"),
        ("1005", "Smartphone", "Electronics", 1, 700, "2023-01-05")]
# Create DataFrame
columns = ["OrderID", "Product", "Category", "Quantity", "Price", "Date"]
df = spark.createDataFrame(data, columns)
```

df.show()

```
OrderID|   Product|   Category|Quantity|Price|      Date|
+-------+----------+-----------+--------+-----+----------+
|   1001|    Laptop|Electronics|       1| 1000|2023-01-01|
|   1002|     Mouse|Electronics|       2|   50|2023-01-02|
|   1003|    Laptop|Electronics|       1| 1200|2023-01-03|
|   1004|     Mouse|Electronics|       3|   30|2023-01-04|
|   1005|Smartphone|Electronics|       1|  700|2023-01-05
```

GroupBy operation on single column

```
# GroupBy and aggregate
result = df.groupBy("Product").agg(sum("Price").alias("Total_Sales"))
# Show results
result.show()
```

```
 Product|Total_Sales|
+----------+-----------+
|    Laptop|       2200|
|     Mouse|         80|
|Smartphone|        700
```

GroupBy operation on Multiple Columns

```
# GroupBy and aggregate
result = df.groupBy(["Product", "Category"]) \
   .agg(sum("Price").alias("Total_Sales"))
# Show results
result.show()
```

```
Product|   Category|Total_Sales|
+----------+-----------+-----------+
|    Laptop|Electronics|       2200|
|     Mouse|Electronics|         80|
|Smartphone|Electronics|        700
```

GroupBy operation on Multiple Aggregations

```
# GroupBy and aggregate
result = df.groupBy("Product") \
   .agg(sum("Price").alias("Total_Sales"),
       sum("Quantity").alias("Total_Quantity"))
# Show results
result.show()
```

```
 Product|Total_Sales|Total_Quantity|
+----------+-----------+--------------+
|    Laptop|       2200|             2|
```

```
|    Mouse|       80|          5|
|Smartphone|      700|          1|
```

er Aggregated data using where condition

can use a combination of where() (which is equivalent to the SQL WHERE clause) and groupBy() to perform a groupBy operation with a specific condition.

```
# GroupBy and aggregate using where condition
result = df.groupBy("Product") \
   .agg(avg("Price").alias("Total_Sales"),
       sum("Quantity").alias("Total_Quantity")) \
   .where(col("Total_Quantity") >= 2)
# Show results
result.show()
```

```
Product|Total_Sales|Total_Quantity|
+-------+-----------+--------------+
| Laptop|    1100.0|           2|
|  Mouse|      40.0|          5
```

Custom Aggregation Functions

```
import pandas as pd
from pyspark.sql.types import FloatType
```

```
from pyspark.sql.functions import pandas_udf
@pandas_udf(FloatType())
def median(column: pd.Series) -> float:
    return float(column.median())
```

```
Category|Median_Price|
+-----------+------------+
|Electronics|      500.0
```

PySpark Joins

Type of Joins

Inner Join

Outer (Full) Join

Left Join

Right Join

Left Semi Join

Left Anti Join

Cross Join

```
import findspark
findspark.init()
from pyspark.sql import SparkSession
```

```
# Initialize Spark Session

spark = SparkSession.builder.master("local").appName("PySpark Join Types").getOrCreate()

# Create sample dataframes

df1 = spark.createDataFrame([(1, "A"), (2, "B"), (3, "C")], ["id", "value1"])

df2 = spark.createDataFrame([(1, "X"), (2, "Y"), (4, "Z")], ["id", "value2"])


# Perform inner join

result = df1.join(df2, on="id", how="inner")

# Show result

result.show()


id|value1|value2|

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

|  1|     A|     X|

|  2|     B|     Y|


# Perform outer join

result = df1.join(df2, on="id", how="outer")


# Show result

result.show()


 id|value1|value2|

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

```
| 1|    A|    X|
| 2|    B|    Y|
| 3|    C|  null|
| 4|  null|    Z
```

```
# Perform left join
result = df1.join(df2, on="id", how="left")
# Show result
result.show()
```

```
id|value1|value2|
+---+------+------+
| 1|    A|    X|
| 3|    C|  null|
| 2|    B|    Y|
```

```
# Perform right join
result = df1.join(df2, on="id", how="right")

# Show result
result.show()
```

```
+---+------+------+
| id|value1|value2|
```

```
+---+------+------+
| 1|     A|     X|
| 2|     B|     Y|
| 4|  null|     Z
```

Left Semi Join

A left semi join returns only the columns from the left dataframe for the rows with matching keys in both dataframes. It is similar to an inner join but only returns the columns from the left dataframe.

```
# Perform left semi join
result = df1.join(df2, on="id", how="left_semi")

# Show result
result.show()
+---+------+
| id|value1|
+---+------+
| 1|     A|
| 2|     B|
```

Left Anti Join

A left anti join returns the rows from the left dataframe that do not have matching keys in the right dataframe. It is the opposite of a left semi join.

Perform left anti join

result = df1.join(df2, on="id", how="left_anti")

Show result

result.show()

```
+---+------+
| id|value1|
+---+------+
|  3|     C|
```

Cross Join

A cross join, also known as a cartesian join, returns the cartesian product of both dataframes. It combines each row from the left dataframe with each row from the right dataframe.

Perform cross join

result = df1.crossJoin(df2)

Show result

result.show()

```
+---+------+---+------+
| id|value1| id|value2|
+---+------+---+------+
|  1|     A|  1|     X|
|  1|     A|  2|     Y|
|  1|     A|  4|     Z|
```

```
| 2|   B|  1|   X|
| 2|   B|  2|   Y|
| 2|   B|  4|   Z|
| 3|   C|  1|   X|
| 3|   C|  2|   Y|
| 3|   C|  4|   Z|
```

PySpark Union?

PySpark Union is an operation that allows you to combine two or more DataFrames with the same schema, creating a single DataFrame containing all rows from the input DataFrames.

It's important to note that the Union operation doesn't eliminate duplicate rows, so you may need to use the distinct() function afterward if you want to remove duplicates.

```
import findspark
findspark.init()
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, IntegerType
# Create a Spark session
spark = SparkSession.builder.appName("PySpark Union Example").getOrCreate()
# Define the schema
schema = StructType([
```

```
    StructField("product", StringType(), True),
    StructField("price", IntegerType(), True),
    StructField("quantity", IntegerType(), True)
])
# Create DataFrame for region A
data_A = [("apple", 3, 5), ("banana", 1, 10), ("orange", 2, 8)]
df_A = spark.createDataFrame(data_A, schema=schema)
# Create DataFrame for region B
data_B = [("apple", 3, 5), ("banana", 1, 15), ("grape", 4, 6)]
df_B = spark.createDataFrame(data_B, schema=schema)
# Create DataFrame for region C
data_C = [("apple", 3, 10), ("banana", 1, 20), ("grape", 4, 10), ("orange", 2, 7)]
df_C = spark.createDataFrame(data_C, schema=schema)

# Perform the Union operation on two DataFrames
df_union = df_A.union(df_B)

# Show the results
df_union.show()
+-------+-----+--------+
|product|price|quantity|
+-------+-----+--------+
|  apple|    3|       5|
| banana|    1|      10|
```

| orange| 2| 8|

| apple| 3| 5|

| banana| 1| 15|

| grape| 4| 6|

Union without Duplicates

It's important to note that the Union operation doesn't eliminate duplicate rows, so you may need to use the distinct() function afterward if you want to remove duplicates.

```
# Perform the Union operation on two DataFrames
df_union_dist = df_A.union(df_B).distinct()
# Show the results
df_union_dist.show()
```

product|price|quantity|

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

| apple| 3| 5|

| banana| 1| 10|

| orange| 2| 8|

| banana| 1| 15|

| grape| 4| 6|

Union Multiple DataFrames

Perform the Union operation on multiple DataFrames

```
df_union_all = df_A.union(df_B).union(df_C)

# Show the results

df_union_all.show()
```

```
product|price|quantity|

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

|  apple|   3|      5|

| banana|   1|     10|

| orange|   2|      8|

|  apple|   3|      5|

| banana|   1|     15|

|  grape|   4|      6|

|  apple|   3|     10|

| banana|   1|     20|

|  grape|   4|     10|

| orange|   2|      7
```