Getting started with PySpark

DHANUSHKUMAR PALANI
@dhanushpalani1120@gmail.com

Learning the Fundamentals of Pyspark using Dataframe

Import Pyspark

In [51]:

```
from pyspark.sql import SparkSession
import getpass
username=getpass.getuser()
```

Create Spark Session

In [52]:

```
spark=SparkSession. \
    builder. \
    config('spark.ui.port','0'). \
    config("spark.sql.warehouse.dir", f"/user/{username}/warehouse"). \
    config('spark.shuffle.useOldFetchProtocol', 'true'). \
    enableHiveSupport(). \
    master('yarn'). \
    getOrCreate()
```

Load the Data

Using Spark Reader API

In [ ]:

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

using Spark range function

In [98]:

```
dfr = spark.range(1, 10, 2)
```

In [99]:

```
dfr.show()
```

```
+---+
| id|
+---+
|  1|
|  3|
|  5|
|  7|
|  9|
+---+
```

Using Local List

In [101]:

```
data = [("Alice", 25, "Female",25000),
        ("Bob", 30, "Male",45000),
        ("Charlie", 22, "Male",28000),
        ("David", 28, "Male",35000),
        ("Eva", 35, "Female",50000),
       ()]

columns = ["Name", "Age", "Gender","Salary"]
df = spark.createDataFrame(data, columns)
```

In [ ]:

```
```

Display dataframe

In [81]:

```
df.show()
```

```
+-------+---+------+------+
|   Name|Age|Gender|Salary|
+-------+---+------+------+
|  Alice| 25|Female| 25000|
|    Bob| 30|  Male| 45000|
|Charlie| 22|  Male| 28000|
|  David| 28|  Male| 35000|
|    Eva| 35|Female| 50000|
+-------+---+------+------+
```

In [102]:

```
print(df)
```

```
+-------+---+------+------+
|   Name|Age|Gender|Salary|
+-------+---+------+------+
|  Alice| 25|Female| 25000|
|    Bob| 30|  Male| 45000|
|Charlie| 22|  Male| 28000|
|  David| 28|  Male| 35000|
|    Eva| 35|Female| 50000|
+-------+---+------+------+
```

Schema:The structure of a DataFrame, specifying the names and data types of its columns

Inferred Schema: While loading the data,may not be precise for complex datasets

In [ ]:

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

Explicit Schema:For better control and accuracy, you can explicitly define the schema when creating a DataFrame.

In [18]:

```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType
schema = StructType([
    StructField("Name", StringType(), True),
    StructField("Age", IntegerType(), True),
    StructField("City", StringType(), True)
])

data = [("A", 25, "New York"), ("B", 30, "San Francisco")]
df = spark.createDataFrame(data,schema)
```

In [19]:

```
df.show()
```

```
+----+---+-------------+
|Name|Age|         City|
+----+---+-------------+
|   A| 25|     New York|
|   B| 30|San Francisco|
+----+---+-------------+
```

check the schema of your DataFrame.

In [22]:

```
df.printSchema()
```

```
root
 |-- Name: string (nullable = true)
 |-- Age: long (nullable = true)
 |-- City: string (nullable = true)
```

Select Columns

In [83]:

```
Name_df=df.select("Name")
```

In [84]:

```
Name_df.show()
```

```
+-------+
|   Name|
+-------+
|  Alice|
|    Bob|
|Charlie|
|  David|
|    Eva|
+-------+
```

Filter Columns with condition

In [85]:

```
Filtered_df=df.filter(df["age"]> 25)
```

In [86]:

```
Filtered_df.show()
```

```
+-----+---+------+------+
| Name|Age|Gender|Salary|
+-----+---+------+------+
|  Bob| 30|  Male| 45000|
|David| 28|  Male| 35000|
|  Eva| 35|Female| 50000|
+-----+---+------+------+
```

In [29]:

```
from pyspark.sql.functions import when
data = [("Alice", 25), ("Bob", 30), ("Charlie", 22)]
columns = ["Name", "Age"]
Df = spark.createDataFrame(data, columns)
```

Conditional functions

In [32]:

```
df_result = Df.withColumn("Category", when(Df["Age"] >= 25, "Old").when(Df["Age"]<25,"Young").otherwise("Unknown"
))
```

In [33]:

```
df_result.show()
```

```
+-------+---+--------+
|   Name|Age|Category|
+-------+---+--------+
|  Alice| 25|     Old|
|    Bob| 30|     Old|
|Charlie| 22|   Young|
+-------+---+--------+
```

String functions in Pyspark

PySpark provides a variety of functions for string manipulation, such as concat, substring, length, lower, upper, trim, etc.

In [34]:

```
from pyspark.sql.functions import concat, substring, lower ,upper ,length ,trim
data = [("John", "Doe"),
        (" Jane", "Smith"),
        ("Bob ", "Johnson")]

schema = ["FirstName", "LastName"]
df = spark.createDataFrame(data,schema)

df_trimmed=df.withColumn("FirstName",trim(df["FirstName"]))

df_trimmed.show()
```

```
+---------+--------+
|FirstName|LastName|
+---------+--------+
|     John|     Doe|
|     Jane|   Smith|
|      Bob| Johnson|
+---------+--------+
```

In [35]:

```
df_transformed= df_trimmed.withColumn("FullName", concat("FirstName", "LastName"))\
                  .withColumn("ShortName", substring("FirstName", 1, 2))\
                  .withColumn("LowerFirstName", lower("FirstName"))\
                  .withColumn("UpperLastName",upper("LastName"))\
                  .withColumn("Length",length("FullName"))\
```

In [36]:

```
df_transformed.show()
```

```
+---------+--------+----------+---------+--------------+-------------+------+
|FirstName|LastName|  FullName|ShortName|LowerFirstName|UpperLastName|Length|
+---------+--------+----------+---------+--------------+-------------+------+
|     John|     Doe|   JohnDoe|       Jo|          john|          DOE|     7|
|     Jane|   Smith| JaneSmith|       Ja|          jane|        SMITH|     9|
|      Bob| Johnson|BobJohnson|       Bo|           bob|      JOHNSON|    10|
+---------+--------+----------+---------+--------------+-------------+------+
```

Date functions

PySpark provides functions for working with date and timestamp data, such as date_add, date_sub, datediff, year, month, day, etc.

In [45]:

```
from pyspark.sql.functions import current_date, date_add, year ,month,dayofmonth
data = [("John", "2020-01-15"),
        ("Jane", "2021-03-22"),
        ("Bob", "2019-11-05")]

schema = ["Name", "BirthDate"]
df = spark.createDataFrame(data,schema)
```

In [46]:

```
df_age = df.withColumn("CurrentDate", current_date())\
               .withColumn("Age", year("CurrentDate") - year("BirthDate"))\
               .withColumn("BirthMonth",month("BirthDate"))\
               .withColumn("Birthday",dayofmonth("BirthDate"))\
```

In [47]:

```
df_age.show()
```

```
+----+----------+-----------+---+----------+--------+
|Name| BirthDate|CurrentDate|Age|BirthMonth|Birthday|
+----+----------+-----------+---+----------+--------+
|John|2020-01-15| 2024-01-18|  4|         1|      15|
|Jane|2021-03-22| 2024-01-18|  3|         3|      22|
| Bob|2019-11-05| 2024-01-18|  5|        11|       5|
+----+----------+-----------+---+----------+--------+
```

Group based on a column

Use agg functions like max,min,avg,sum with groupBy

In [87]:

```
import pyspark.sql.functions as f

Grouped_df=df.groupBy("Gender").agg(f.max("Salary").alias("max"))
```

In [88]:

```
Grouped_df.show()
```

```
+------+-----+
|Gender|  max|
+------+-----+
|Female|50000|
|  Male|45000|
+------+-----+
```

Sort the columns

using orderBy

In [114]:

```
ordered_df = df.orderBy("Salary")
```

In [115]:

```
ordered_df.show()
```

```
+-------+---+------+------+
|   Name|Age|Gender|Salary|
+-------+---+------+------+
|  Alice| 25|Female| 25000|
|Charlie| 22|  Male| 28000|
|  David| 28|  Male| 35000|
|    Bob| 30|  Male| 45000|
|    Eva| 35|Female| 50000|
+-------+---+------+------+
```

using sort

In [118]:

```
sorted_df = df.sort("Salary")
```

In [119]:

```
sorted_df.show()
```

```
+-------+---+------+------+
|   Name|Age|Gender|Salary|
+-------+---+------+------+
|  Alice| 25|Female| 25000|
|Charlie| 22|  Male| 28000|
|  David| 28|  Male| 35000|
|    Bob| 30|  Male| 45000|
|    Eva| 35|Female| 50000|
+-------+---+------+------+
```

order the df in descending

In [116]:

```
ordered_df = df.orderBy("Salary",ascending=False)
```

In [117]:

```
ordered_df.show()
```

```
+-------+---+------+------+
|   Name|Age|Gender|Salary|
+-------+---+------+------+
|    Eva| 35|Female| 50000|
|    Bob| 30|  Male| 45000|
|  David| 28|  Male| 35000|
|Charlie| 22|  Male| 28000|
|  Alice| 25|Female| 25000|
+-------+---+------+------+
```

Add a column

In [110]:

```
ctc_df=df.withColumn("Annual_salary",(df["Salary"]*12))
```

In [109]:

```
ctc_df.show()
```

```
+-------+---+------+------+----------+
|   Name|Age|Gender|Salary|Annual_CTC|
+-------+---+------+------+----------+
|  Alice| 25|Female| 25000|    300000|
|    Bob| 30|  Male| 45000|    540000|
|Charlie| 22|  Male| 28000|    336000|
|  David| 28|  Male| 35000|    420000|
|    Eva| 35|Female| 50000|    600000|
+-------+---+------+------+----------+
```

Rename a Column

In [112]:

```
updated_df=df.withColumnRenamed("Salary","Monthly_income")
```

In [113]:

```
updated_df.show()
```

```
+-------+---+------+--------------+
|   Name|Age|Gender|Monthly_income|
+-------+---+------+--------------+
|  Alice| 25|Female|         25000|
|    Bob| 30|  Male|         45000|
|Charlie| 22|  Male|         28000|
|  David| 28|  Male|         35000|
|    Eva| 35|Female|         50000|
+-------+---+------+--------------+
```

Handle Missing Data

Drops rows with any null values

In [17]:

```
data = [("Alice", 25, "Female",25000),
        ("Bob", 30, "Male",45000),
        ("Charlie", 22, "Male",None),
        ("David", 28, "Male",35000),
        (None, 35, "Female",50000),
       ]

columns = ["Name", "Age", "Gender","Salary"]
df1 = spark.createDataFrame(data, columns)
```

In [12]:

```
dropped_df=df1.na.drop()
```

In [13]:

```
dropped_df.show()
```

```
+-----+---+------+------+
| Name|Age|Gender|Salary|
+-----+---+------+------+
|Alice| 25|Female| 25000|
|  Bob| 30|  Male| 45000|
|David| 28|  Male| 35000|
|  Eva| 35|Female| 50000|
+-----+---+------+------+
```

Fills null values with 0

In [23]:

```
salary_filled=df1.na.fill(0)
```

In [24]:

```
salary_filled.show()
```

```
+-------+---+------+------+
|   Name|Age|Gender|Salary|
+-------+---+------+------+
|  Alice| 25|Female| 25000|
|    Bob| 30|  Male| 45000|
|Charlie| 22|  Male|     0|
|  David| 28|  Male| 35000|
|   null| 35|Female| 50000|
+-------+---+------+------+
```

Fills null values on conditions

In [21]:

```
Name_filled=df1.na.fill({"Name":"unknown"},{"salary":0})
```

In [22]:

```
Name_filled.show()
```

```
+-------+---+------+------+
|   Name|Age|Gender|Salary|
+-------+---+------+------+
|  Alice| 25|Female| 25000|
|    Bob| 30|  Male| 45000|
|Charlie| 22|  Male|  null|
|  David| 28|  Male| 35000|
|unknown| 35|Female| 50000|
+-------+---+------+------+
```

Handling duplicates

In [3]:

```
data = [("John", 28),
        ("Jane", 25),
        ("John", 28),
        ("Bob", 30),
        ("Jane", 25)]
schema = ["Name", "Age"]
df = spark.createDataFrame(data,schema)
```

dropDuplicates method eliminates rows with identical values in all columns

In [4]:

```
df_nodup= df.dropDuplicates()
```

In [5]:

```
df_nodup.show()
```

```
+----+---+
|Name|Age|
+----+---+
|John| 28|
|Jane| 25|
| Bob| 30|
+----+---+
```

dropDuplicates([column]) method eliminates rows with identical values in specific columns

In [6]:

```
df_no_duplicates_name = df.dropDuplicates(["Name"])
```

In [7]:

```
df_no_duplicates_name.show()
```

```
+----+---+
|Name|Age|
+----+---+
| Bob| 30|
|John| 28|
|Jane| 25|
+----+---+
```

Joins Explained using Pyspark

In [9]:

```
data1 = [(1, "Alice", 1), (2, "Bob", 1), (3, "Charlie", 2),(4,"David",None)]
data2 = [(1, "Sales"), (2, "Marketing")]

columns1 = ["id", "name", "deptId"]
columns2 = ["deptId", "deptName"]


df3 = spark.createDataFrame(data1, columns1)
df4 = spark.createDataFrame(data2, columns2)
```

Inner Join-Returns only the rows where there is a match in both DataFrames based on the specified column(s).

In [10]:

```
inner_joined_df = df3.join(df4, "deptId", "inner")
inner_joined_df.show()
```

```
+------+---+-------+---------+
|deptId| id|   name| deptName|
+------+---+-------+---------+
|     1|  1|  Alice|    Sales|
|     1|  2|    Bob|    Sales|
|     2|  3|Charlie|Marketing|
+------+---+-------+---------+
```

Left Join-Returns all rows from the left DataFrame and the matching rows from the right DataFrame. If there is no match, null values are filled for the columns from the right DataFrame.

In [11]:

```
left_joined_df = df3.join(df4, "deptId", "left").orderBy("id")
left_joined_df.show()
```

```
+------+---+-------+---------+
|deptId| id|   name| deptName|
+------+---+-------+---------+
|     1|  1|  Alice|    Sales|
|     1|  2|    Bob|    Sales|
|     2|  3|Charlie|Marketing|
|  null|  4|  David|     null|
+------+---+-------+---------+
```

Left Semi Join-Returns all unique rows from the left DataFrame where there is a match in the right DataFrame. It only includes the rows that have a corresponding match in the right DataFrame.

In [12]:

```
left_semi_joined_df = df3.join(df4, "deptId", "left_semi")
left_semi_joined_df.show()
```

```
+------+---+-------+
|deptId| id|   name|
+------+---+-------+
|     1|  1|  Alice|
|     1|  2|    Bob|
|     2|  3|Charlie|
+------+---+-------+
```

Left Anti Join-Returns all rows from the left DataFrame where there is no match in the right DataFrame. It only includes the rows that do not have a corresponding match in the right DataFrame.

In [13]:

```
left_anti_joined_df = df3.join(df4, "deptId", "left_anti")








left_anti_joined_df.show()
```

```
+------+---+-----+
|deptId| id| name|
+------+---+-----+
|  null|  4|David|
+------+---+-----+
```

Right Join-Returns all rows from the right DataFrame and the matching rows from the left DataFrame. If there is no match, null values are filled for the columns from the left DataFrame.

In [14]:

```
right_joined_df = df3.join(df4, "deptId", "right")
right_joined_df.show()
```

```
+------+---+-------+---------+
|deptId| id|   name| deptName|
+------+---+-------+---------+
|     1|  1|  Alice|    Sales|
|     1|  2|    Bob|    Sales|
|     2|  3|Charlie|Marketing|
+------+---+-------+---------+
```

Full Outer Join-Returns all rows from both DataFrames, filling null values for columns where there is no match.

In [15]:

```
outer_joined_df = df3.join(df4, "deptId", "outer")
outer_joined_df.show()
```

```
+------+---+-------+---------+
|deptId| id|   name| deptName|
+------+---+-------+---------+
|  null|  4|  David|     null|
|     1|  1|  Alice|    Sales|
|     1|  2|    Bob|    Sales|
|     2|  3|Charlie|Marketing|
+------+---+-------+---------+
```

Window Functions

In [5]:

```
data5 = [(1,"Alice","HR",25000),
        (2,"Bob","IT",45000),
        (3,"Charlie","HR",55000),
        (4,"David","IT",35000),
        (5,"Frank","Finance",50000),
        (6,"Eva","Finance",50000),
         (7,"Danny","Finance",48000)
       ]

column5 = ["Employee_id","Name","Department","Salary"]
df5 = spark.createDataFrame(data5,column5)
```

Importing window functions

In [6]:

```
from pyspark.sql.window import Window
from pyspark.sql.functions import  rank, dense_rank, sum ,row_number
```

In [7]:

```
window_spec = Window.partitionBy("department").orderBy(df5["salary"].desc())
```

rank(): Assigns a rank to each row within its partition based on the specified order. The rank is not skipped in case of ties.

In [8]:

```
df_rank = df5.withColumn("rank", rank().over(window_spec))
```

In [9]:

```
df_rank.show()
```

```
+-----------+-------+----------+------+----+
|Employee_id|   Name|Department|Salary|rank|
+-----------+-------+----------+------+----+
|          3|Charlie|        HR| 55000|   1|
|          1|  Alice|        HR| 25000|   2|
|          5|  Frank|   Finance| 50000|   1|
|          6|    Eva|   Finance| 50000|   1|
|          7|  Danny|   Finance| 48000|   3|
|          2|    Bob|        IT| 45000|   1|
|          4|  David|        IT| 35000|   2|
+-----------+-------+----------+------+----+
```

dense_rank(): Similar to rank(), but skips the rank in case of ties. It assigns consecutive ranks.

In [17]:

```
df_dense_rank = df5.withColumn("dense_rank", dense_rank().over(window_spec))
```

In [18]:

```
df_dense_rank.show()
```

```
+-----------+-------+----------+------+----------+
|Employee_id|   Name|Department|Salary|dense_rank|
+-----------+-------+----------+------+----------+
|          3|Charlie|        HR| 55000|         1|
|          1|  Alice|        HR| 25000|         2|
|          5|  Frank|   Finance| 50000|         1|
|          6|    Eva|   Finance| 50000|         1|
|          7|  Danny|   Finance| 48000|         2|
|          2|    Bob|        IT| 45000|         1|
|          4|  David|        IT| 35000|         2|
+-----------+-------+----------+------+----------+
```

sum("salary").over(window_spec): Calculates the total salary for each department based on the specified window specification.

In [19]:

```
df_total_salary = df5.withColumn("total_salary", sum("salary").over(window_spec))
```

In [20]:

```
df_total_salary.show()
```

```
+-----------+-------+----------+------+------------+
|Employee_id|   Name|Department|Salary|total_salary|
+-----------+-------+----------+------+------------+
|          3|Charlie|        HR| 55000|       55000|
|          1|  Alice|        HR| 25000|       80000|
|          5|  Frank|   Finance| 50000|      100000|
|          6|    Eva|   Finance| 50000|      100000|
|          7|  Danny|   Finance| 48000|      148000|
|          2|    Bob|        IT| 45000|       45000|
|          4|  David|        IT| 35000|       80000|
+-----------+-------+----------+------+------------+
```

Create a table by saving a DataFrame to a specific storage format and location, such as Parquet, Avro, or ORC.

As a temporary table

In [ ]:

```
data = [("John", 25),
        ("Jane", 30),
        ("Bob", 22)]

schema = ["Name", "Age"]
```

In [48]:

```
df = spark.createDataFrame(data,schema)

path = "pathtotable"

df.write.mode("overwrite").parquet(path)

df.createOrReplaceTempView("table_name")

result = spark.sql("SELECT * FROM table_name")

result.show()
```

```
+----+---+
|Name|Age|
+----+---+
|John| 25|
|Jane| 30|
| Bob| 22|
+----+---+
```

As permanent table (ensure creating Spark session with Hive support using enableHiveSupport)

In [ ]:

```
data = [("John", 25),
        ("Jackob", 18),
        ("Bob", 22)]

schema = ["Name", "Age"]

df = spark.createDataFrame(data,schema)

df.write.mode("overwrite").saveAsTable("table_name")
result = spark.sql("SELECT * FROM table_name")
result.show()
```

Reading modes

Permissive (Default)

It sets the parsing mode to permissive, meaning that it tries to parse corrupt records and sets corrupt values to null.

In [ ]:

```
df = spark.read.option("mode", "permissive").csv("pathtodata")
```

dropMalformed:

In this mode, Spark drops the rows that contain corrupt records during the reading process.

In [ ]:

```
df = spark.read.option("mode", "dropMalformed").csv("pathtodata")
```

failFast:

This mode causes Spark to fail immediately if it encounters any corrupt records during the reading process.

In [ ]:

```
df = spark.read.option("mode", "failFast").csv("pathtodata")
```

In [ ]:

```

```

In [ ]:

```

```

Writing Modes

overwrite:

This mode overwrites the existing data at the specified location. If the data does not exist, it creates a new one.

In [ ]:

```
df.write.mode("overwrite").parquet("pathtooutput")
```

append:

This mode appends the data to the existing data at the specified location. It's useful when you want to add new data to an existing dataset.

In [ ]:

```
df.write.mode("append").parquet("pathtooutput")
```

ignore:

This mode ignores the operation if the data or file already exists at the specified location. It doesn't perform any action and doesn't raise an error.

In [ ]:

```
df.write.mode("ignore").parquet("pathtooutput")
```

error (default):

This mode raises an error if the data or file already exists at the specified location. It prevents accidental overwrites.

In [ ]:

```
df.write.mode("error").parquet("pathtooutput")
```

Closing the Spark Session:

In [ ]:

```
spark.stop()
```

FOLLOW FOR MORE DATA CONTENT

Do Like and Repost

By DhanushKumar Palani