

Master PySpark Zero to Big Data Hero

Day 1 - Day 10: Core PySpark DataFrame Operations

- **Day 1:** Creating DataFrames in PySpark
 - **Day 2:** Loading CSV Files into DataFrames
 - **Day 3:** Defining Schema in PySpark
-
- **Day 4:** Column Selection Tips
 - **Day 5:** Column Manipulation – Add, Rename & Drop Columns
 - **Day 6:** DataFrame Operations: Distinct and Filter
 - **Day 7:** Sorting & String Functions
 - **Day 8:** String Functions & Concatenation
 - **Day 9:** Split, Explode & Array Functions
 - **Day 10:** Trimming & Padding Strings
-

Advanced PySpark Topics

- **Day 11-12:** Date Functions Part 1 & 2
 - **Day 13:** Handling Null Values Like a Pro
 - **Day 14-15:** Aggregation Functions – Parts 1 & 2
 - **Day 16-19:** Joins Series – From Introduction to Output Manipulation in 4 Parts
 - **Day 20:** Mastering when & otherwise Statements for Conditional Logic
-

Advanced PySpark Functions

- **Day 21:** cast() and printSchema()
 - **Day 22:** Union vs UnionAll in PySpark
 - **Day 23:** Union vs. UnionByName in PySpark
 - **Day 24-27:** Mastering PySpark Window Functions: Part 1 to Part 4
 - **Day 28:** explode() vs explode_outer
-
- **Day 29:** Pivot in PySpark
 - **Day 30:** Unpivot in PySpark

20+ Spark Differences to Clear Interview Questions

Bonus Section: Spark Interview Preparation

Master PySpark: From Zero to Big Data Hero!!

PySpark: Spark SQL and DataFrames

Spark SQL is a module in Spark for working with structured data. It allows you to query structured data inside Spark using SQL and integrates seamlessly with DataFrames.

A **DataFrame** in PySpark is a distributed collection of data organized into named columns. It's conceptually similar to a table in a relational database or a DataFrame in R/Python.

How to Create DataFrames in PySpark

Here are some different ways to create DataFrames in PySpark:

1. Creating DataFrame Manually with Hardcoded Values:

This is one of the most straightforward ways to create a DataFrame using Python lists of tuples.

```
# Sample Data
data = [(1, "Alice"), (2, "Bob"), (3, "Charlie"), (4, "David"), (5, "Eve")]
columns = ["ID", "Name"]

# Create DataFrame
df = spark.createDataFrame(data, columns)

# Show DataFrame
df.show()
```

```
+---+-----+
| ID|   Name|
+---+-----+
|  1|  Alice|
|  2|   Bob|
|  3|Charlie|
|  4|  David|
|  5|   Eve|
+---+-----+
```

2. Creating DataFrame from Pandas:

```
import pandas as pd

# Sample Pandas DataFrame
pandas_df = pd.DataFrame(data, columns=columns)

# Convert to PySpark DataFrame
df_from_pandas = spark.createDataFrame(pandas_df)
df_from_pandas.show()
```

```
+---+-----+
| ID|   Name|
+---+-----+
|  1|  Alice|
|  2|   Bob|
|  3|Charlie|
|  4|  David|
|  5|   Eve|
+---+-----+
```

3. Create DataFrame from Dictionary:

```
data_dict = [{"ID": 1, "Name": "Alice"}, {"ID": 2, "Name": "Bob"}]
df_from_dict = spark.createDataFrame(data_dict)
df_from_dict.show()
```

```
+---+-----+
| ID|  Name|
+---+-----+
|  1|Alice|
|  2|  Bob|
+---+-----+
```

4. Create Empty DataFrame:

You can create an empty DataFrame with just schema definitions.

```

from pyspark.sql.types import StructType, StructField, StringType, IntegerType

# Define Schema
schema = StructType([
    StructField("ID", IntegerType(), True),
    StructField("Name", StringType(), True)
])

# Create Empty DataFrame
empty_df = spark.createDataFrame([], schema)
empty_df.show()

```

▶ (3) Spark Jobs

▶  empty_df: pyspark.sql.dataframe.DataFrame = [ID: integer, Name: string]

```

+---+-----+
| ID|Name|
+---+-----+
+---+-----+

```

5. Creating DataFrame from Structured Data (CSV, JSON, Parquet)

Reading CSV file into DataFrame

```

df_csv = spark.read.csv("/path/to/file.csv", header=True,
inferSchema=True)
df_csv.show()

```

Reading JSON file into DataFrame

```

df_json = spark.read.json("/path/to/file.json")
df_json.show()

```

Reading Parquet file into DataFrame

```

df_parquet = spark.read.parquet("/path/to/file.parquet")
df_parquet.show()

```

show() Function in PySpark DataFrames

The show() function in PySpark displays the contents of a DataFrame in a tabular format. It has several useful parameters for customization:

1. n: Number of rows to display (default is 20)
2. truncate: If set to True, it truncates column values longer than 20 characters (default is True).
3. vertical: If set to True, prints rows in a vertical format.

```
#Show the first 3 rows, truncate columns to 25 characters, and display vertically:
```

```
df.show(n=3, truncate=25, vertical=True)
```

```
#Show entire DataFrame (default settings):
```

```
df.show()
```

```
#Show the first 10 rows:
```

```
df.show(10)
```

```
#Show DataFrame without truncating any columns:
```

```
df.show(truncate=False)
```

Master PySpark: From Zero to Big Data Hero!!

Loading Data from CSV File into a DataFrame

Loading data into DataFrames is a fundamental step in any data processing workflow in PySpark. This document outlines how to load data from CSV files into a DataFrame, including using a custom schema and the implications of using the inferSchema option.

Step-by-Step Guide

1. Import Required Libraries

Before loading the data, ensure you import the necessary modules:

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, IntegerType,
StringType, DoubleType
```

2. Define the Schema

You can define a custom schema for your CSV file. This allows you to explicitly set the data types for each column.

```
# Define the schema for the CSV file
custom_schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True),
    StructField("salary", DoubleType(), True)
])
```

3. Read the CSV File

Load the CSV file into a DataFrame using the read.csv() method. Here, header=True treats the first row as headers, and inferSchema=True allows Spark to automatically assign data types to columns.

```
# Read the CSV file with the custom schema
df = spark.read.csv("your_file.csv", schema=custom_schema,
header=True)
```

4. Load Multiple CSV Files

To read multiple CSV files into a single DataFrame, you can pass a list of file paths. Ensure that the schema is consistent across all files.

```
# List of file paths
file_paths = ["file1.csv", "file2.csv", "file3.csv"]
# Read multiple CSV files into a single DataFrame
df = spark.read.csv(file_paths, header=True, inferSchema=True)
```

5. Load a CSV from FileStore

Here is an example of loading a CSV file from Databricks FileStore:

```
df = spark.read.csv("/FileStore/tables/Order.csv", header=True,
inferSchema=True, sep=',')
```

6. Display the DataFrame

Use the following commands to check the schema and display the DataFrame:

```
# Print the schema of the DataFrame
df.printSchema()

# Show the first 20 rows of the DataFrame
df.show() # Displays only the first 20 rows

# Display the DataFrame in a tabular format
display(df) # For Databricks notebooks
```

Interview Question: How Does inferSchema Work?

- **Behind the Scenes:** When you use inferSchema, Spark runs a job that scans the CSV file from top to bottom to identify the best-suited data type for each column based on the values it encounters.

Does It Make Sense to Use inferSchema?

- **Pros:**
 - Useful when the schema of the file keeps changing, as it allows Spark to automatically detect the data types.
- **Cons:**
 - **Performance Impact:** Spark must scan the entire file, which can take extra time, especially for large files.
 - **Loss of Control:** You lose the ability to explicitly define the schema, which may lead to incorrect data types if the data is inconsistent.

Conclusion

Loading data from CSV files into a DataFrame is straightforward in PySpark. Understanding how to define a schema and the implications of using inferSchema is crucial for optimizing your data processing workflows.

This document provides a comprehensive overview of how to load CSV data into DataFrames in PySpark, along with considerations for using schema inference. Let me know if you need any more details or adjustments!

Master PySpark: From Zero to Big Data Hero!!

PySpark DataFrame Schema Definition

1. Defining Schema Programmatically with StructType

```
from pyspark.sql.types import *

# Define the schema using StructType
employeeSchema = StructType([
    StructField("ID", IntegerType(), True),
    StructField("Name", StringType(), True),
    StructField("Age", IntegerType(), True),
    StructField("Salary", DoubleType(), True),
    StructField("Joining_Date", StringType(), True), # Keeping as
String for date issues
    StructField("Department", StringType(), True),
    StructField("Performance_Rating", IntegerType(), True),
    StructField("Email", StringType(), True),
    StructField("Address", StringType(), True),
    StructField("Phone", StringType(), True)
])

# Load the DataFrame with the defined schema
df = spark.read.load("/FileStore/tables/employees.csv",
format="csv", header=True, schema=employeeSchema)

# Print the schema of the DataFrame
df.printSchema()

# Optionally display the DataFrame
# display(df)
```

```
root
|-- ID: integer (nullable = true)
|-- Name: string (nullable = true)
|-- Age: integer (nullable = true)
|-- Salary: double (nullable = true)
|-- Joining_Date: string (nullable = true)
|-- Department: string (nullable = true)
|-- Performance_Rating: integer (nullable = true)
|-- Email: string (nullable = true)
|-- Address: string (nullable = true)
|-- Phone: string (nullable = true)
```

2. Defining Schema as a String

```
# Define the schema as a string
```

```
employeeSchemaString = '''
```

```
ID Integer,
```

```
Name String,
```

```
Age Integer,
```

```
Salary Double,
```

```
Joining_Date String,
```

```
Department String,
```

```
Performance_Rating Integer,
```

```
Email String,
```

```
Address String,
```

```
Phone String
```

```
'''
```

```
# Load the DataFrame with the defined schema
```

```
df =
```

```
spark.read.load("dbfs:/FileStore/shared_uploads/imsvk11@gmail.com/e  
mployee_data.csv", format="csv", header=True,  
schema=employeeSchemaString)
```

```
# Print the schema of the DataFrame
```

```
df.printSchema()
```

```
# Optionally display the DataFrame
```

```
# display(df)
```

```
root
|-- ID: integer (nullable = true)
|-- Name: string (nullable = true)
|-- Age: integer (nullable = true)
|-- Salary: double (nullable = true)
|-- Joining_Date: string (nullable = true)
|-- Department: string (nullable = true)
|-- Performance_Rating: integer (nullable = true)
|-- Email: string (nullable = true)
|-- Address: string (nullable = true)
|-- Phone: string (nullable = true)
```

Explanation

- **Schema Definition:** Both methods define a schema for the DataFrame, accommodating the dataset's requirements, including handling null values where applicable.
- **Data Types:** The Joining_Date column is defined as StringType to accommodate potential date format issues or missing values.
- **Loading the DataFrame:** The spark.read.load method is used to load the CSV file into a DataFrame using the specified schema.
- **Printing the Schema:** The df.printSchema() function allows you to verify that the DataFrame is structured as intended.

Master PySpark: From Zero to Big Data Hero!!

PySpark Column Selection & Manipulation: Key Techniques

1. Different Methods to Select Columns

In PySpark, you can select specific columns in multiple ways:

- Using `col()` function/ `column()` / string way:

```
#Using col() function
df.select(col("Name")).show()

#Using column() function
df.select(column("Age")).show()

#Directly using string name
df.select("Salary").show()
```

2. Selecting Multiple Columns Together

You can combine different methods to select multiple columns:

```
#multiple column

df2 = df.select("ID", "Name", col("Salary"), column("Department"),
df.Phone)
df2.show()
```

3. Listing All Columns in a DataFrame

To get a list of all the column names:

```
#get all column name
df.columns
```

4. Renaming Columns with alias()

You can rename columns using the alias() method:

```
df.select(
    col("Name").alias('EmployeeName'), # Rename "Name" to "EmployeeName"
    col("Salary").alias('EmployeeSalary'), # Rename "Salary" to
    "EmployeeSalary"
    column("Department"), # Select "Department"
    df.Joining_Date # Select "Joining_Date"
).show()
```

5. Using selectExpr() for Concise Column Selection

selectExpr() allows you to use SQL expressions directly and rename columns concisely:

```
df.selectExpr("Name as EmployeeName", "Salary as EmployeeSalary",
    "Department").show()
```

Summary

- Use col(), column(), or string names to select columns.
- Use expr() and selectExpr() for SQL-like expressions and renaming.
- Use alias() to rename columns.
- Get the list of columns using df.columns.

Master PySpark: From Zero to Big Data Hero!!

PySpark DataFrame Manipulation part 2: Adding, Renaming, and Dropping Columns

1. Adding New Columns with withColumn()

In PySpark, the `withColumn()` function is widely used to add new columns to a DataFrame. You can either assign a constant value using `lit()` or perform transformations using existing columns.

- Add a constant value column:

```
newdf = df.withColumn("NewColumn", lit(1))
```

- Add a column based on an expression:

```
newdf = df.withColumn("withinCountry", expr("Country == 'India'"))
```

This function allows adding multiple columns, including calculated ones:

- Example:
 - Assign a constant value with `lit()`.
 - Perform calculations using existing columns like multiplying values.

2. Renaming Columns with withColumnRenamed()

PySpark provides the `withColumnRenamed()` method to rename columns. This is especially useful when you want to change the names for clarity or to follow naming conventions:

- Renaming a column:

```
new_df = df.withColumnRenamed("oldColumnName", "newColumnName")
```

- Handling column names with special characters or spaces: If a column has special characters or spaces, you need to use backticks (```) to escape it:

```
newdf.select("`New Column Name`").show()
```

3. Dropping Columns with drop()

To remove unwanted columns, you can use the `drop()` method:

- Drop a single column:

```
df2 = df.drop("Country")
```

- Drop multiple columns:

```
df2 = df.drop("Country", "Region")
```

Dropping columns creates a new DataFrame, and the original DataFrame remains unchanged.

4. Immutability of DataFrames

In Spark, DataFrames are immutable by nature. This means that after creating a DataFrame, its contents cannot be changed. All transformations like adding, renaming, or dropping columns result in a new DataFrame, keeping the original one intact.

- For instance, dropping columns creates a new DataFrame without altering the original:

```
newdf = df.drop("ItemType", "SalesChannel")
```

This immutability ensures data consistency and supports Spark's parallel processing, as transformations do not affect the source data.

Key Points

- Use `withColumn()` for adding columns, with `lit()` for constant values and expressions for computed values.
- Use `withColumnRenamed()` to rename columns and backticks for special characters or spaces.
- Use `drop()` to remove one or more columns.
- DataFrames are immutable in Spark—transformations result in new DataFrames, leaving the original unchanged.

Master PySpark: From Zero to Big Data Hero!!

Here's a structured set of notes with code to cover changing data types, filtering data, and handling unique/distinct values in PySpark using the employee data:

1. Changing Data Types (Schema Transformation)

In PySpark, you can change the data type of a column using the `cast()` method. This is helpful when you need to convert data types for columns like Salary or Phone.

```
from pyspark.sql.functions import col

# Change the 'Salary' column from integer to double
df = df.withColumn("Salary", col("Salary").cast("double"))

# Convert 'Phone' column to string
df = df.withColumn("Phone", col("Phone").cast("string"))

df.printSchema()
```

2. Filtering Data

You can filter rows based on specific conditions. For instance, to filter employees with a salary greater than 50,000:

```
# Filter rows where Salary is greater than 50,000
filtered_df = df.filter(col("Salary") > 50000)
filtered_df.show()

# Filtering rows where Age is not null
filtered_df = df.filter(df["Age"].isNotNull())
filtered_df.show()
```

3. Multiple Filters (Chaining Conditions)

You can also apply multiple conditions using `&` or `|` (AND/OR) to filter data. For example, finding employees over 30 years old and in the IT department:


```
# Filter rows where Age > 30 and Department is 'IT'
filtered_df = df.filter((df["Age"] > 30) & (df["Department"] ==
"IT"))
filtered_df.show()
```

4. Filtering on Null or Non-Null Values

Filtering based on whether a column has NULL values or not is crucial for data cleaning:

```
# Filter rows where 'Address' is NULL
filtered_df = df.filter(df["Address"].isNull())
filtered_df.show()

# Filter rows where 'Email' is NOT NULL
filtered_df = df.filter(df["Email"].isNotNull())
filtered_df.show()
```

5. Handling Unique or Distinct Data

To get distinct rows or unique values from your dataset:

```
# Get distinct rows from the entire DataFrame
unique_df = df.distinct()
unique_df.show()

# Get distinct values from the 'Department' column
unique_departments_df = df.select("Department").distinct()
unique_departments_df.show()
```

To remove duplicates based on specific columns, such as Email or Phone, use `dropDuplicates()`:

```
# Remove duplicates based on 'Email' column
unique_df = df.dropDuplicates(["Email"])
unique_df.show()

# Remove duplicates based on both 'Phone' and 'Email'
unique_df = df.dropDuplicates(["Phone", "Email"])
unique_df.show()
```

6. Counting Distinct Values

You can count distinct values in a particular column, or combinations of columns:

```
# Count distinct values in the 'Department' column
distinct_count_department =
df.select("Department").distinct().count()
print("Distinct Department Count:", distinct_count_department)

# Count distinct combinations of 'Department' and
'Performance_Rating'
distinct_combinations_count = df.select("Department",
"Performance_Rating").distinct().count()
print("Distinct Department and Performance Rating Combinations:",
distinct_combinations_count)
```

This set of operations will help you efficiently manage and transform your data in PySpark, ensuring data integrity and accuracy for your analysis!

Mastering PySpark DataFrame Operations

1. **Changing Data Types:** Easily modify column types using `.cast()`. E.g., change 'Salary' to double or 'Phone' to string for better data handling.
2. **Filtering Data:** Use `.filter()` or `.where()` to extract specific rows. For example, filter employees with a salary over 50,000 or non-null Age.
3. **Multiple Conditions:** Chain filters with `&` and `|` to apply complex conditions, such as finding employees over 30 in the IT department.
4. **Handling NULLs:** Use `.isNull()` and `.isNotNull()` to filter rows with missing or available values, such as missing addresses or valid emails.
5. **Unique/Distinct Values:** Use `.distinct()` to get unique rows or distinct values in a column. Remove duplicates based on specific fields like Email or Phone using `.dropDuplicates()`.
6. **Count Distinct Values:** Count distinct values in one or multiple columns to analyze data diversity, such as counting unique departments or combinations of Department and Performance_Rating.

Master PySpark: From Zero to Big Data Hero!!

Here's an example of a PySpark DataFrame with data and corresponding notes that explain the various transformations, sorting, and string functions:

Sample Data Creation

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, desc, asc, concat,
concat_ws, initcap, lower, upper, instr, length, lit

# Create a Spark session
spark =
SparkSession.builder.appName("SortingAndStringFunctions").getOrCreate()

# Sample data
data = [
    ("USA", "North America", 100, 50.5),
    ("India", "Asia", 300, 20.0),
    ("Germany", "Europe", 200, 30.5),
    ("Australia", "Oceania", 150, 60.0),
    ("Japan", "Asia", 120, 45.0),
    ("Brazil", "South America", 180, 25.0)
]

# Define the schema
columns = ["Country", "Region", "UnitsSold", "UnitPrice"]

# Create DataFrame
df = spark.createDataFrame(data, columns)

# Display the original DataFrame
df.show()
```

Notes with Examples

Sorting the DataFrame

1. Sort by a single column (ascending order):

```
df.orderBy("Country").show(5)
```

► (1) Spark Jobs

```
+-----+-----+-----+-----+
| Country|      Region|UnitsSold|UnitPrice|
+-----+-----+-----+-----+
|Australia|    Oceania|      150|     60.0|
|  Brazil|South America|      180|     25.0|
|  Germany|    Europe|      200|     30.5|
|   India|     Asia|      300|     20.0|
|   Japan|     Asia|      120|     45.0|
+-----+-----+-----+-----+
only showing top 5 rows
```

Note: By default, the sorting is in ascending order. This shows the top 5 countries in alphabetical order.

2. Sort by multiple columns:

```
df.orderBy("Country", "UnitsSold").show(5)
```

► (1) Spark Jobs

```
+-----+-----+-----+-----+
| Country|      Region|UnitsSold|UnitPrice|
+-----+-----+-----+-----+
|Australia|    Oceania|      150|     60.0|
|  Brazil|South America|      180|     25.0|
|  Germany|    Europe|      200|     30.5|
|   India|     Asia|      300|     20.0|
|   Japan|     Asia|      120|     45.0|
+-----+-----+-----+-----+
only showing top 5 rows
```

Note: Here, the DataFrame is sorted first by Country (ascending), and within the same country, it is sorted by UnitsSold in ascending order.

3. Sort by a column in descending order and limit:

```
sorted_df = df.orderBy(desc("Country")).limit(3)
sorted_df.show()
```

▶ (1) Spark Jobs

▶ sorted_df: pyspark.sql.dataframe.DataFrame = [Country: string, Region: string ... 2 more fields]

Country	Region	UnitsSold	UnitPrice
USA	North America	100	50.5
Japan	Asia	120	45.0
India	Asia	300	20.0

Note: This sorts the DataFrame by Country in descending order and limits the output to the top 3 rows.

4. Sorting with null values last:

```
sorted_df = df.orderBy(col("Country").desc(), nulls_last=True).show(5)
```

▶ (1) Spark Jobs

Country	Region	UnitsSold	UnitPrice
USA	North America	100	50.5
Japan	Asia	120	45.0
India	Asia	300	20.0
Germany	Europe	200	30.5
Brazil	South America	180	25.0

only showing top 5 rows

Note: This ensures that null values (if present) are placed at the end when sorting by Country.

Summary of Key Functions:

- Sorting: You can sort a DataFrame by one or more columns using `.orderBy()` or `.sort()`. By default, sorting is ascending, but you can change it using `asc()` or `desc()`.

These functions and transformations are common in PySpark for manipulating and querying data effectively!

Master PySpark: From Zero to Big Data Hero!!

String Functions

1. Convert the first letter of each word to uppercase (initcap):

```
df.select(initcap(col("Country"))).show()
```

► (3) Spark Jobs

```
+-----+
|initcap(Country)|
+-----+
|          Usa|
|         India|
|       Germany|
|     Australia|
|         Japan|
|         Brazil|
+-----+
```

Note: This transforms the first letter of each word in the Country column to uppercase.

2. Convert all text to lowercase (lower):

```
df.select(lower(col("Country"))).show()
```

► (3) Spark Jobs

```
+-----+
|lower(Country)|
+-----+
|         usa|
|        india|
|      germany|
|    australia|
|        japan|
|        brazil|
+-----+
```

Note: Converts all letters in the Country column to lowercase.

3. Convert all text to uppercase (upper):

```
df.select(upper(col("Country"))).show()
```

► (3) Spark Jobs

```
+-----+
|upper(Country)|
+-----+
|          USA|
|         INDIA|
|        GERMANY|
|    AUSTRALIA|
|         JAPAN|
|        BRAZIL|
+-----+
```

Note: Converts all letters in the Country column to uppercase.

Concatenation Functions

1. Concatenate two columns:

```
df.select(concat(col("Region"), col("Country"))).show()
```

► (3) Spark Jobs

```
+-----+
|concat(Region, Country)|
+-----+
|North AmericaUSA|
|      AsiaIndia|
|    EuropeGermany|
|OceaniaAustralia|
|      AsiaJapan|
|South AmericaBrazil|
+-----+
```

Note: Concatenates the values of Region and Country without any separator.

2. Concatenate with a separator:

```
df.select(concat_ws(' | ', col("Region"), col("Country"))).show()
```

Note: Concatenates the values of Region and Country with | as a separator.

3. Create a new concatenated column:

```
concatenated_df = df.withColumn("concatenated", concat(df["Region"], lit(" "), df["Country"]))
concatenated_df.show()
```

▶ (3) Spark Jobs

▶ concatenated_df: pyspark.sql.dataframe.DataFrame = [Country: string, Region: string ... 3 more fields]

Country	Region	UnitsSold	UnitPrice	concatenated
USA	North America	100	50.5	North America USA
India	Asia	300	20.0	Asia India
Germany	Europe	200	30.5	Europe Germany
Australia	Oceania	150	60.0	Oceania Australia
Japan	Asia	120	45.0	Asia Japan
Brazil	South America	180	25.0	South America Brazil

Note: This creates a new column concatenated by combining Region and Country with a space between them.

Summary of Key Functions:

- **String Manipulation:** You can convert strings to lowercase, uppercase, or capitalize the first letter of each word. Use `initcap()`, `lower()`, and `upper()` for these transformations.
- **Concatenation:** Use `concat()` to join two columns or `concat_ws()` to join with a separator.

These functions and transformations are common in PySpark for manipulating and querying data effectively!

Master PySpark: From Zero to Big Data Hero!!

Split Function In Dataframe

Let's create a PySpark DataFrame for employee data, which will include columns such as EmployeeID, Name, Department, and Skills.

I'll demonstrate the usage of the split, explode, and other relevant PySpark functions with the employee data, along with notes for each operation.

Sample Data Creation for Employee Data

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import split, explode, size,
array_contains, col
# Sample employee data
data = [
    (1, "Alice", "HR", "Communication Management"),
    (2, "Bob", "IT", "Programming Networking"),
    (3, "Charlie", "Finance", "Accounting Analysis"),
    (4, "David", "HR", "Recruiting Communication"),
    (5, "Eve", "IT", "Cloud DevOps")
]

# Define the schema
columns = ["EmployeeID", "Name", "Department", "Skills"]

# Create DataFrame
df = spark.createDataFrame(data, columns)

# Display the original DataFrame
df.show(truncate=False)
```

EmployeeID	Name	Department	Skills
1	Alice	HR	Communication Management
2	Bob	IT	Programming Networking
3	Charlie	Finance	Accounting Analysis
4	David	HR	Recruiting Communication
5	Eve	IT	Cloud DevOps

Notes with Examples

1. Split the "Skills" column:

We will split the Skills column into an array, where each skill is separated by a space.

python

```
# Split the "Skills" column and alias it as "Skills_Array"
df2 = df.select(col("EmployeeID"), col("Name"), split(col("Skills"), " ").alias("Skills_Array"))
df2.show(truncate=False)
```

▶ (3) Spark Jobs

▶ df2: pyspark.sql.dataframe.DataFrame = [EmployeeID: long, Name: string ... 1 more field]

EmployeeID	Name	Skills_Array
1	Alice	[Communication, Management]
2	Bob	[Programming, Networking]
3	Charlie	[Accounting, Analysis]
4	David	[Recruiting, Communication]
5	Eve	[Cloud, DevOps]

Note: This splits the Skills column into an array of skills based on the space separator. The alias("Skills_Array") gives the resulting array a meaningful name.

2. Select the first skill from the "Skills_Array":

You can select specific elements from an array using index notation. In this case, we'll select the first skill from the Skills_Array.

```
# Select the first element from the "Skills_Array" (index 0)
df2.select(col("EmployeeID"), col("Name"), col("Skills_Array")[0].alias("First_Skill")).show(truncate=False)
```

▶ (3) Spark Jobs

EmployeeID	Name	First_Skill
1	Alice	Communication
2	Bob	Programming
3	Charlie	Accounting
4	David	Recruiting
5	Eve	Cloud

Note: The array index starts from 0, so Skills_Array[0] gives the first skill for each employee.

3. Calculate the size of the "Skills_Array":

We can calculate how many skills each employee has by using the `size()` function.

```
# Calculate the size of the "Skills_Array"
df2.select(col("EmployeeID"), col("Name"), size(col("Skills_Array")).alias("Number_of_Skills")).show(truncate=False)
```

▶ (3) Spark Jobs

EmployeeID	Name	Number_of_Skills
1	Alice	2
2	Bob	2
3	Charlie	2
4	David	2
5	Eve	2

Note: The `size()` function returns the number of elements (skills) in the `Skills_Array`.

4. Check if the array contains a specific skill:

We can check if a particular skill (e.g., "Cloud") is present in the employee's skillset using the `array_contains()` function.

```
# Check if the "Skills_Array" contains the skill "Cloud"
df.select(col("EmployeeID"), col("Name"), array_contains(split(col("Skills"), " "), "Cloud").alias("Has_Cloud_Skill")).show(truncate=False)
```

▶ (3) Spark Jobs

EmployeeID	Name	Has_Cloud_Skill
1	Alice	false
2	Bob	false
3	Charlie	false
4	David	false
5	Eve	true

Note: This returns a boolean indicating whether the array contains the specified skill, "Cloud", for each employee.

5. Use the explode function to transform array elements into individual rows:

The explode() function can be used to flatten the array into individual rows, where each skill becomes a separate row for the employee.

```
# Explode the "Skills_Array" into separate rows
df3 = df2.withColumn("Skill", explode(col("Skills_Array")))
df3.select("EmployeeID", "Name", "Skill").show(truncate=False)
```

▶ (3) Spark Jobs

▶ df3: pyspark.sql.dataframe.DataFrame = [EmployeeID: long, Name: string ... 2 more field]

EmployeeID	Name	Skill
1	Alice	Communication
1	Alice	Management
2	Bob	Programming
2	Bob	Networking
3	Charlie	Accounting
3	Charlie	Analysis
4	David	Recruiting
4	David	Communication
5	Eve	Cloud
5	Eve	DevOps

Note: The explode() function takes an array column and creates a new row for each element of the array. Here, each employee will have multiple rows, one for each skill.

Summary of Key Functions:

- **split():** This splits a column's string value into an array based on a specified delimiter (in this case, a space).
- **explode():** Converts an array column into multiple rows, one for each element in the array.
- **size():** Returns the number of elements in an array.
- **array_contains():** Checks if a specific value exists in the array.
- **selectExpr():** Allows you to use SQL expressions (like array[0]) to select array elements.

Master PySpark: From Zero to Big Data Hero!!

Trim Function in Dataframe

Let's create a new sample dataset for employees and demonstrate the usage of string trimming and padding functions (ltrim, rtrim, trim, lpad, and rpad) in PySpark.

Steps:

1. Create sample employee data.
2. Demonstrate the usage of ltrim(), rtrim(), trim(), lpad(), and rpad() on string columns.

Sample Data Creation for Employees

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import lit, ltrim, rtrim, rpad, lpad,
trim, col
```

```
# Sample employee data with leading and trailing spaces in the
'Name' column
```

```
data = [
    (1, " Alice  ", "HR"),
    (2, "  Bob", "IT"),
    (3, "Charlie  ", "Finance"),
    (4, "  David ", "HR"),
    (5, "Eve  ", "IT")
]
```

```
# Define the schema for the DataFrame
```

```
columns = ["EmployeeID", "Name", "Department"]
```

```
# Create DataFrame
```

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

```
# Show the original DataFrame
```

```
df.show(truncate=False)
```

```
+-----+-----+-----+
|EmployeeID|Name      |Department|
+-----+-----+-----+
|1          | Alice    |HR         |
|2          |  Bob     |IT         |
|3          |Charlie   |Finance    |
|4          |  David   |HR         |
|5          |Eve       |IT         |
+-----+-----+-----+
```

Applying Trimming and Padding Functions

1. ltrim(), rtrim(), and trim():

- **ltrim()**: Removes leading spaces.
- **rtrim()**: Removes trailing spaces.
- **trim()**: Removes both leading and trailing spaces.

2. lpad() and rpad():

- **lpad()**: Pads the left side of a string with a specified character up to a certain length.
- **rpadd()**: Pads the right side of a string with a specified character up to a certain length.

Example:

```
# Apply trimming and padding functions
result_df = df.select(
    col("EmployeeID"),
    col("Department"),
    ltrim(col("Name")).alias("ltrim_Name"), # Remove leading spaces
    rtrim(col("Name")).alias("rtrim_Name"), # Remove trailing spaces
    trim(col("Name")).alias("trim_Name"),   # Remove both leading and trailing spaces
    lpad(col("Name"), 10, "X").alias("lpad_Name"), # Left pad with "X" to make the
string length 10
    rpad(col("Name"), 10, "Y").alias("rpad_Name") # Right pad with "Y" to make the
string length 10
)

# Show the resulting DataFrame
result_df.show(truncate=False)
```

	EmployeeID	Department	ltrim_Name	rtrim_Name	trim_Name	lpad_Name	rpadd_Name
1	HR	Alice	Alice	Alice	Alice	X Alice	Alice Y
2	IT	Bob	Bob	Bob	Bob	XXXXX Bob	BobYYYYY
3	Finance	Charlie	Charlie	Charlie	Charlie	XCharlie	Charlie Y
4	HR	David	David	David	David	XX David	David YY
5	IT	Eve	Eve	Eve	Eve	XXXXXEve	Eve YYYYY

Output Explanation:

- **ltrim_Name**: The leading spaces from the Name column are removed.
- **rtrim_Name**: The trailing spaces from the Name column are removed.
- **trim_Name**: Both leading and trailing spaces are removed from the Name column.
- **lpad_Name**: The Name column is padded on the left with "X" until the string length becomes 10.
- **rpadd_Name**: The Name column is padded on the right with "Y" until the string length becomes 10.

Master PySpark: From Zero to Big Data Hero!!

Date Function in Dataframe – Part 1

In PySpark, you can use various date functions to manipulate and analyze date and timestamp columns. Below, I'll provide a sample dataset and demonstrate key date functions like `current_date`, `current_timestamp`, `date_add`, `date_sub`, `datediff`, and `months_between`.

Code Explanation with Notes

1. Creating a Spark Session:

- We begin by creating a Spark session to run the PySpark operations.

2. Generating a DataFrame:

- Using `spark.range(10)` creates a DataFrame with 10 rows and a single column (`id`) with numbers ranging from 0 to 9.
- Two additional columns are added:
 - **today**: Contains the current date using `current_date()`.
 - **now**: Contains the current timestamp using `current_timestamp()`.

3. Date Manipulation Functions:

- **date_add**: Adds a specified number of days to the date.
- **date_sub**: Subtracts a specified number of days from the date.
- **datediff**: Returns the difference in days between two dates.
- **months_between**: Returns the number of months between two dates.

Code Example

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import current_date, current_timestamp,
date_add, date_sub, col, datediff, months_between, to_date, lit

# Generate a DataFrame with 10 rows, adding "today" and "now"
columns
dateDF = spark.range(10).withColumn("today",
current_date()).withColumn("now", current_timestamp())

# Show the DataFrame with today and now columns
dateDF.show(truncate=False)
```

```

+---+-----+-----+
|id |today      |now              |
+---+-----+-----+
|0  |2024-10-13|2024-10-13 15:27:37.466|
|1  |2024-10-13|2024-10-13 15:27:37.466|
|2  |2024-10-13|2024-10-13 15:27:37.466|
|3  |2024-10-13|2024-10-13 15:27:37.466|
|4  |2024-10-13|2024-10-13 15:27:37.466|
|5  |2024-10-13|2024-10-13 15:27:37.466|
|6  |2024-10-13|2024-10-13 15:27:37.466|
|7  |2024-10-13|2024-10-13 15:27:37.466|
|8  |2024-10-13|2024-10-13 15:27:37.466|
|9  |2024-10-13|2024-10-13 15:27:37.466|
+---+-----+-----+

```

Explanation of Code and Output

1. `current_date` and `current_timestamp`:

- `current_date()` gives the current date (e.g., 2024-10-12).
- `current_timestamp()` provides the current timestamp, which includes both date and time (e.g., 2024-10-12 12:34:56).
- These are used to create columns `today` and `now` in the DataFrame.

2. `date_add` and `date_sub`:

- `date_sub(col("today"), 5)`: Subtracts 5 days from the current date, so if today is 2024-10-12, it returns 2024-10-07.
- `date_add(col("today"), 5)`: Adds 5 days to the current date, returning 2024-10-17.

```

# Add 5 days and subtract 5 days from "today"
dateDF.select(
  date_sub(col("today"), 5).alias("date_sub_5_days"),
  date_add(col("today"), 5).alias("date_add_5_days")
).show(1)

```

► (2) Spark Jobs

```

+-----+-----+
|date_sub_5_days|date_add_5_days|
+-----+-----+
|      2024-10-08|      2024-10-18|
+-----+-----+
only showing top 1 row

```


3. datediff:

- **datediff(col("week_ago"), col("today"))**: Calculates the difference in days between the current date and 7 days ago (i.e., -7).

```
# Calculate the days difference between "today" and "week_ago" (7 days ago)
dateDF.withColumn("week_ago", date_sub(col("today"), 7))\
  .select(datediff(col("week_ago"), col("today")).alias("days_difference")).show(1)
```

► (2) Spark Jobs

```
+-----+
|days_difference|
+-----+
|              -7|
+-----+
only showing top 1 row
```

4. months_between:

- **months_between(to_date(lit("2016-01-01")), to_date(lit("2017-01-01")))**: Calculates the number of months between January 1, 2016, and January 1, 2017, which is -12 months because start_date is earlier than end_date.

```
# Calculate the number of months between two specific dates
dateDF.select(
  to_date(lit("2016-01-01")).alias("start_date"),
  to_date(lit("2017-01-01")).alias("end_date")
).select(months_between(col("start_date"), col("end_date")).alias("months_between")).show(1)
```

► (2) Spark Jobs

```
+-----+
|months_between|
+-----+
|          -12.0|
+-----+
only showing top 1 row
```

Master PySpark: From Zero to Big Data Hero!!

Date Function in Dataframe – Part 2

In PySpark, handling dates with the correct format and extracting date/time components such as year, month, day, etc., can be done with functions like `to_date`, `to_timestamp`, `year`, `month`, `dayofmonth`, `hour`, `minute`, and `second`. Below is a detailed explanation of how to work with date formats and extract date components.

Code Explanation with Notes

1. Default Date Parsing (`to_date`):

- When using `to_date()`, the default date format is `yyyy-MM-dd`.
- If the format of the string does not match this, PySpark returns null for invalid date parsing.

```
# Example: "2016-20-12" is invalid (20 is not a valid month), returns null
dateDF.select(
  to_date(lit("2016-20-12")).alias("incorrect_date"),
  to_date(lit("2017-12-11")).alias("correct_date")
).show(1)
```

► (2) Spark Jobs

```
+-----+-----+
|incorrect_date|correct_date|
+-----+-----+
|          null|  2017-12-11|
+-----+-----+
only showing top 1 row
```

2. Handling Custom Date Formats:

- You can specify a custom date format using the `to_date` function by providing a format string, such as `yyyy-dd-MM`.
- This allows PySpark to correctly parse the dates that deviate from the default format.

```
from pyspark.sql.functions import to_date

dateFormat = "yyyy-dd-MM"
cleanDateDF = spark.range(1).select(
    to_date(lit("2017-12-11"), dateFormat).alias("correct_format_date"),
    to_date(lit("2017-20-12"), dateFormat).alias("incorrect_format_date")
)
cleanDateDF.show()
```

► (1) Spark Jobs

```
cleanDateDF: pyspark.sql.dataframe.DataFrame
  correct_format_date: date
  incorrect_format_date: date
```

```
+-----+-----+
|correct_format_date|incorrect_format_date|
+-----+-----+
|      2017-11-12|      2017-12-20|
+-----+-----+
```

- Here, "2017-12-11" will be parsed correctly since it fits yyyy-dd-MM, but "2017-20-12" will return null since the day (20) is out of the valid range for December (month 12).

3. Handling Timestamps:

- You can use `to_timestamp` to convert strings with both date and time into a timestamp format. This is useful when working with datetime values.
- After casting to a timestamp, you can extract various date/time components such as the year, month, day, hour, minute, and second.

```
from pyspark.sql.functions import to_timestamp, year, month, dayofmonth, hour, minute, second, col

# Select all components (year, month, day, hour, minute, second) in a single show
cleanDateDF.select(
    to_timestamp(col("correct_format_date"), dateFormat).alias("timestamp"),
    year(to_timestamp(col("correct_format_date"), dateFormat)).alias("year"),
    month(to_timestamp(col("correct_format_date"), dateFormat)).alias("month"),
    dayofmonth(to_timestamp(col("correct_format_date"), dateFormat)).alias("day"),
    hour(to_timestamp(col("correct_format_date"), dateFormat)).alias("hour"),
    minute(to_timestamp(col("correct_format_date"), dateFormat)).alias("minute"),
    second(to_timestamp(col("correct_format_date"), dateFormat)).alias("second")
).show()
```

► (1) Spark Jobs

```
+-----+-----+-----+-----+-----+-----+
|timestamp|year|month|day|hour|minute|second|
+-----+-----+-----+-----+-----+-----+
|2017-11-12 00:00:00|2017| 11| 12|  0|    0|    0|
+-----+-----+-----+-----+-----+-----+
```

Detailed Explanation of Each Function

1. `to_date`:

- Converts a string column to a date column based on the given format. If the format does not match, null is returned.

2. `to_timestamp`:

- Converts a string column with date and time information into a timestamp, which includes both date and time.

3. Extracting Date Components:

- **year**: Extracts the year from a date or timestamp.
- **month**: Extracts the month from a date or timestamp.
- **dayofmonth**: Extracts the day of the month from a date or timestamp.
- **hour**: Extracts the hour from a timestamp.
- **minute**: Extracts the minute from a timestamp.
- **second**: Extracts the second from a timestamp.

Sample Output

For the input "2017-12-11" (with the format yyyy-dd-MM), you can expect the following results:

- **Year**: 2017
- **Month**: 12
- **Day**: 11
- **Hour**: 0 (since no time is provided)
- **Minute**: 0
- **Second**: 0

For invalid date strings (like "2017-20-12"), you will get null in the resulting DataFrame.

Master PySpark: From Zero to Big Data Hero!!

Null Handling in Dataframe

Here's an example of how you can use PySpark functions for null handling with sales data. The code includes null detection, dropping rows with nulls, filling null values, and using `coalesce()` to handle nulls in aggregations. I will provide the notes alongside the code.

Sample Sales Data with Null Values

```
# Sample data: sales data with nulls
data = [
    ("John", "North", 100, None),
    ("Doe", "East", None, 50),
    (None, "West", 150, 30),
    ("Alice", None, 200, 40),
    ("Bob", "South", None, None),
    (None, None, None, None)
]
columns = ["Name", "Region", "UnitsSold", "Revenue"]
# Create DataFrame
df = spark.createDataFrame(data, columns)
df.show()
```

```
+-----+-----+-----+-----+
| Name|Region|UnitsSold|Revenue|
+-----+-----+-----+-----+
| John| North|      100|  null|
|  Doe|  East|     null|    50|
| null|  West|     150|    30|
| Alice| null|     200|    40|
|  Bob| South|     null|   null|
| null|  null|     null|   null|
+-----+-----+-----+-----+
```

Notes:

1. Detecting Null Values:

The `isNull()` function identifies rows where a specified column has null values. The output shows a boolean flag for each row to indicate whether the value in the column is null.

```
from pyspark.sql.functions import *
# Detecting Null Values in the "Region" Column
df.select("Name", "Region", isnull("Region").alias("is_Region_Null")).show()
```

▶ (3) Spark Jobs

```
+-----+-----+-----+
| Name|Region|is_Region_Null|
+-----+-----+-----+
| John| North|         false|
|  Doe|  East|         false|
| null|  West|         false|
| Alice| null|          true|
|  Bob| South|         false|
| null| null|          true|
+-----+-----+-----+
```

2. Dropping Rows with Null Values:

- **dropna()** removes rows that contain null values in any column when the default mode is used.
- Specifying "all" ensures rows are only removed if *all* columns contain null values.
- You can also apply null handling only on specific columns by providing a list of column names to the subset parameter.

```
# Dropping Rows with Null Values (if any value in the row is null)
df2 = df.dropna()
df2.show()
```

▶ (3) Spark Jobs

▶ df2: pyspark.sql.dataframe.DataFrame = [Name: string, Region: string ... 2 more fields]

```
+-----+-----+-----+
| Name|Region|UnitsSold|Revenue|
+-----+-----+-----+
+-----+-----+-----+
```

```
# Dropping Rows where all values are Null
df3 = df.na.drop("all")
df3.show()
```


▶ (3) Spark Jobs

▶ df3: pyspark.sql.dataframe.DataFrame = [Name: string, Region: string ... 2 more fields]

```
+-----+-----+-----+
| Name|Region|UnitsSold|Revenue|
+-----+-----+-----+
| John| North|      100|    null|
|  Doe|  East|     null|     50|
| null|  West|     150|     30|
| Alice| null|     200|     40|
|  Bob| South|     null|    null|
+-----+-----+-----+
```

```
# Dropping Rows if null values exist in "Name" or "Region" columns
df4 = df.na.drop("all", subset=["Name", "Region"])
df4.show()
```

▶ (3) Spark Jobs

▶  df4: pyspark.sql.dataframe.DataFrame = [Name: string, Region: string ... 2 more fields]

```
+-----+-----+-----+-----+
| Name|Region|UnitsSold|Revenue|
+-----+-----+-----+-----+
| John| North|    100|   null|
|  Doe|  East|    null|    50|
| null| West|    150|    30|
| Alice| null|    200|    40|
|  Bob| South|    null|   null|
+-----+-----+-----+-----+
```

3. Filling Null Values:

- **fillna()** allows replacing null values with specified replacements, either for all columns or selectively.
- In the example, nulls in Region are replaced with "Unknown", while UnitsSold and Revenue nulls are filled with 0.


✓ 09:28 PM (1s)

6

```
# Filling Null Values with Specific Values
```

```
df5 = df.fillna({"Region": "Unknown", "UnitsSold": 0, "Revenue": 0})
df5.show()
```

▶ (3) Spark Jobs

▶  df5: pyspark.sql.dataframe.DataFrame = [Name: string, Region: string ... 2 more fields]

```
+-----+-----+-----+-----+
| Name| Region|UnitsSold|Revenue|
+-----+-----+-----+-----+
| John|  North|    100|    0|
|  Doe|   East|     0|   50|
| null|  West|    150|   30|
| Alice|Unknown|    200|   40|
|  Bob|  South|     0|    0|
| null|Unknown|     0|    0|
+-----+-----+-----+-----+
```

```
# Filling all Null values in "Region" and "Name" columns
df6 = df.na.fill("N/A", subset=["Name", "Region"])
df6.show()
```

▶ (3) Spark Jobs

▶ df6: pyspark.sql.dataframe.DataFrame = [Name: string, Region: string ... 3 more fields]

```
+-----+-----+-----+-----+
| Name|Region|UnitsSold|Revenue|
+-----+-----+-----+
| John| North|      100|  null|
|  Doe|  East|     null|    50|
|  N/A|  West|     150|    30|
|Alice|  N/A|     200|    40|
|  Bob| South|     null|   null|
|  N/A|  N/A|     null|   null|
+-----+-----+-----+-----+
```

4. Coalesce Function:

The **coalesce()** function returns the first non-null value in a list of columns. It's useful when you need to handle missing data by providing alternative values from other columns.

```
# Using coalesce() to handle nulls by taking the first non-null value
df7 = df.withColumn("Adjusted_UnitsSold", coalesce("UnitsSold", "Revenue"))
df7.show()
```

▶ (3) Spark Jobs

▶ df7: pyspark.sql.dataframe.DataFrame = [Name: string, Region: string ... 3 more fields]

```
+-----+-----+-----+-----+-----+
| Name|Region|UnitsSold|Revenue|Adjusted_UnitsSold|
+-----+-----+-----+-----+-----+
| John| North|      100|  null|             100|
|  Doe|  East|     null|    50|              50|
| null|  West|     150|    30|             150|
|Alice| null|     200|    40|             200|
|  Bob| South|     null|   null|             null|
| null| null|     null|   null|             null|
+-----+-----+-----+-----+-----+
```


Handling Nulls in Aggregations:

Null values can distort aggregate functions like `mean()`. Using `coalesce()` in an aggregation ensures that any null values are replaced with a default (e.g., 0.0) to avoid skewing the results.

```
# Aggregating while handling null values using coalesce
df8 = df.groupBy("Region").agg(coalesce(mean("UnitsSold"), lit(0)).alias("Avg_UnitsSold"))
df8.show()
```

▶ (2) Spark Jobs

▶ df8: pyspark.sql.dataframe.DataFrame = [Region: string, Avg_UnitsSold: double]

Region	Avg_UnitsSold
North	100.0
East	0.0
West	150.0
null	200.0
South	0.0

Null Handling in DataFrames - Summary

- 1. Detecting Nulls:** Use `isNull()` to identify null values in specific columns.
- 2. Dropping Nulls:** `dropna()` removes rows with null values, either in any or all columns. You can target specific columns using the `subset` parameter.
- 3. Filling Nulls:** `fillna()` replaces nulls with specified default values, either for all or selected columns.
- 4. Coalesce Function:** `coalesce()` returns the first non-null value from multiple columns, providing a fallback when some columns contain nulls.
- 5. Aggregations:** Use `coalesce()` during aggregations like `mean()` to handle nulls by substituting them with defaults (e.g., 0), ensuring accurate results.

Master PySpark: From Zero to Big Data Hero!!

Aggregate function in Dataframe – Part 1

Let's create a sample DataFrame using PySpark that includes various numerical values. This dataset will be useful for demonstrating the aggregate functions.

```
# Create sample data
```

```
data = [  
    Row(id=1, value=10),  
    Row(id=2, value=20),  
    Row(id=3, value=30),  
    Row(id=4, value=None),  
    Row(id=5, value=40),  
    Row(id=6, value=20)  
]
```

```
# Create DataFrame
```

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

```
# Show the DataFrame
```

```
df.show()
```

Sample Output

```
+---+-----+  
| id|value|  
+---+-----+  
|  1|   10|  
|  2|   20|  
|  3|   30|  
|  4|  null|  
|  5|   40|  
|  6|   20|  
+---+-----+
```

Aggregate Functions in PySpark

1. **Summation (sum):** Sums up the values in a specified column.

```
from pyspark.sql import functions as F  
  
# Summation  
total_sum = df.select(F.sum("value")).show()
```

► (2) Spark Jobs

```
+-----+  
|sum(value)|  
+-----+  
|       120|  
+-----+
```

2. average of the values in a specified column.

```
# Average
average_value = df.select(F.avg("value")).show()
```

► (2) Spark Jobs

```
+-----+
|avg(value)|
+-----+
|      24.0|
+-----+
```

3. Count (count): Counts the number of non-null values in a specified column.

```
# Count
non_null_count = df.select(F.count("value")).show()
```

► (2) Spark Jobs

```
+-----+
|count(value)|
+-----+
|           5|
+-----+
```

4. Maximum (max) and Minimum (min): Finds the maximum and minimum values in a specified column.

```
# Maximum and Minimum
max_min_values = df.select(F.max("value"), F.min("value")).show()
```

► (2) Spark Jobs

```
+-----+-----+
|max(value)|min(value)|
+-----+-----+
|        40|        10|
+-----+-----+
```

5. Distinct Values Count (countDistinct): Counts the number of distinct values in a specified column.

```
# Distinct Values Count
distinct_count = df.select(F.countDistinct("value")).show()
```

▶ (3) Spark Jobs

```
+-----+
|count(DISTINCT value)|
+-----+
|                      4|
+-----+
```

Notes

- **Handling Nulls:** The count function will count only non-null values, while sum, avg, max, and min will ignore null values in their calculations.
- **Performance:** Aggregate functions can be resource-intensive, especially on large datasets. Using the appropriate partitioning can improve performance.
- **Use Cases:**
 - **Summation:** Useful for calculating total sales, total revenue, etc.
 - **Average:** Helpful for finding average metrics like average sales per day.
 - **Count:** Useful for counting occurrences, such as the number of transactions.
 - **Max/Min:** Helps to determine the highest and lowest values, such as maximum sales on a specific day.
 - **Distinct Count:** Useful for finding unique items, like unique customers or products.

This should give you a solid understanding of aggregate functions in PySpark! If you have any specific questions or need further assistance, feel free to ask!

Master PySpark: From Zero to Big Data Hero!!

Aggregate function in Dataframe – Part 2

Let's create some sample data to demonstrate each of these PySpark DataFrame operations and give notes explaining the functions. Here's how you can create a PySpark DataFrame and apply these operations.

Sample Data

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.types import StructType, StructField, StringType, IntegerType

# Create Spark session
spark =
SparkSession.builder.appName("AggregationExamples").getOrCreate()

# Sample data
data = [
    ("HR", 10000, 500, "John"),
    ("Finance", 20000, 1500, "Doe"),
    ("HR", 15000, 1000, "Alice"),
    ("Finance", 25000, 2000, "Eve"),
    ("HR", 20000, 1500, "Mark")
]

# Define schema
schema = StructType([
    StructField("department", StringType(), True),
    StructField("salary", IntegerType(), True),
    StructField("bonus", IntegerType(), True),
    StructField("employee_name", StringType(), True)
])

# Create DataFrame
df = spark.createDataFrame(data, schema)
df.show()
```

Sample Data Output:

```
+-----+-----+-----+-----+
|department|salary|bonus|employee_name|
+-----+-----+-----+-----+
|      HR| 10000|  500|        John|
| Finance| 20000| 1500|         Doe|
|      HR| 15000| 1000|        Alice|
| Finance| 25000| 2000|         Eve|
|      HR| 20000| 1500|        Mark|
+-----+-----+-----+-----+
```

1. Grouped Aggregation

Perform aggregation within groups based on a grouping column.

```
df.groupBy("department").agg(
    F.sum("salary"),
    F.avg("salary"),
    F.max("salary"),
    F.min("salary")
).show()
```

► (2) Spark Jobs

```
+-----+-----+-----+-----+-----+
|department|sum(salary)|avg(salary)|max(salary)|min(salary)|
+-----+-----+-----+-----+-----+
|      HR|      45000|    15000.0|     20000|     10000|
| Finance|      45000|    22500.0|     25000|     20000|
+-----+-----+-----+-----+-----+
```

Explanation:

- **sum:** Adds the values in the group for column1.
- **avg:** Calculates the average value of column1 in each group.
- **max:** Finds the maximum value.
- **min:** Finds the minimum value.

2. Multiple Aggregations

Perform multiple aggregations in a single step.

```
df.groupBy("department").agg(
  F.count("salary"),
  F.avg("bonus"),
  F.max("salary")
).show()
```

► (2) Spark Jobs

```
+-----+-----+-----+
|department|count(salary)|avg(bonus)|max(salary)|
+-----+-----+-----+
|      HR   |          3   |  1000.0   |    20000   |
| Finance   |          2   |  1750.0   |    25000   |
+-----+-----+-----+
```

Explanation:

- count: Counts the number of rows in each group.
- avg: Computes the average of column2.
- max: Finds the maximum value in column1.

3. Concatenate Strings

Concatenate strings within a column.

```
df.agg(F.concat_ws(", ", F.collect_list("employee_name")).alias("concatenated_names")).show(truncate=False)
```

► (2) Spark Jobs

```
+-----+
|concatenated_names|
+-----+
|John, Doe, Alice, Eve, Mark|
+-----+
```

```
df.groupBy("department").agg(
  F.concat_ws(", ", F.collect_list("employee_name")).alias("concatenated_names")
).show(truncate=False)
```

► (2) Spark Jobs

```
+-----+-----+
|department|concatenated_names|
+-----+-----+
|HR        |John, Alice, Mark |
|Finance   |Doe, Eve          |
+-----+-----+
```

Explanation:

- concat_ws: Concatenates string values within the column, separating them by the specified delimiter (,).

4. First and Last

Find the first and last values in a column (within each group).

```
df.groupBy("department").agg(F.first("employee_name"), F.last("employee_name")).show()
```

► (2) Spark Jobs

```
+-----+-----+-----+
|department|first(employee_name)|last(employee_name)|
+-----+-----+-----+
|   Finance|          Doe|          Eve|
|      HR|          John|          Mark|
+-----+-----+-----+
```

Explanation:

- first: Retrieves the first value of the name column within each group.
- last: Retrieves the last value of the name column within each group.

5. Standard Deviation and Variance

Calculate the standard deviation and variance of values in a column.

```
df.select(F.stddev("salary"), F.variance("salary")).show()
```

► (2) Spark Jobs

```
+-----+-----+
|stddev_samp(salary)|var_samp(salary)|
+-----+-----+
|  5700.87712549569|          3.25E7|
+-----+-----+
```

Explanation:

- stddev: Calculates the standard deviation of column.
- variance: Calculates the variance of column.

6. Aggregation with Alias

Provide custom column names for the aggregated results.


```
df.groupBy("department").agg(
    F.sum("salary").alias("total_salary"),
    F.avg("salary").alias("average_salary")
).show()
```

► (2) Spark Jobs

```
+-----+-----+-----+
|department|total_salary|average_salary|
+-----+-----+-----+
|      HR|      45000|      15000.0|
|  Finance|      45000|      22500.0|
+-----+-----+-----+
```

Explanation:

- `.alias()`: Used to rename the resulting columns from the aggregation.

7. Sum of Distinct Values

Calculate the sum of distinct values in a column.

```
df.select(F.sumDistinct("salary")).show()
```

► (3) Spark Jobs

```
+-----+
|sum(DISTINCT salary)|
+-----+
|              70000|
+-----+
```

Explanation:

- `sumDistinct`: Sums only the distinct values in column. This avoids counting duplicates.

These examples showcase various aggregation operations in PySpark, useful in data summarization and analysis. The grouped aggregation functions like `sum()`, `avg()`, and `max()` are frequently used in big data pipelines to compute metrics for different segments or categories.

Master PySpark: From Zero to Big Data Hero!!

Joins in Dataframe – Part 1

Joins in PySpark

Joins are used to combine two DataFrames based on a common column or condition. PySpark supports several types of joins, similar to SQL. Below are explanations and examples for each type of join.

1. Inner Join

Code:

```
inner_join = df1.join(df2, on="common_column", how="inner")
```

Explanation:

- **Purpose:** Returns rows where there is a match in both DataFrames (df1 and df2) based on the common_column.
 - **Behavior:** Rows with no matching value in either DataFrame are excluded.
 - **Use Case:** When you only need records that exist in both DataFrames.
-

2. Left Join (Left Outer Join)

Code:

```
left_join = df1.join(df2, on="common_column", how="left")
```

Explanation:

- **Purpose:** Returns all rows from df1 and the matching rows from df2. If no match exists in df2, the result will contain NULL for columns from df2.
 - **Behavior:** All rows from the left DataFrame (df1) are preserved, even if there's no match in the right DataFrame (df2).
 - **Use Case:** When you want to retain all rows from df1, even if there's no match in df2.
-

3. Right Join (Right Outer Join)

Code:

```
right_join = df1.join(df2, on="common_column", how="right")
```

Explanation:

- **Purpose:** Returns all rows from df2 and the matching rows from df1. If no match exists in df1, the result will contain NULL for columns from df1.
 - **Behavior:** All rows from the right DataFrame (df2) are preserved, even if there's no match in the left DataFrame (df1).
 - **Use Case:** When you want to retain all rows from df2, even if there's no match in df1.
-

4. Full Join (Outer Join)

Code:

```
full_join = df1.join(df2, on="common_column", how="outer")
```

Explanation:

- **Purpose:** Returns all rows when there is a match in either df1 or df2. Non-matching rows will have NULL values in the columns from the other DataFrame.
 - **Behavior:** Retains all rows from both DataFrames, filling in NULL where there is no match.
 - **Use Case:** When you want to retain all rows from both DataFrames, regardless of whether there's a match.
-

5. Left Semi Join

Code:

```
left_semi_join = df1.join(df2, on="common_column", how="left_semi")
```

Explanation:

- **Purpose:** Returns only the rows from df1 where there is a match in df2. It behaves like an inner join but only keeps columns from df1.
 - **Behavior:** Filters df1 to only keep rows that have a match in df2.
 - **Use Case:** When you want to filter df1 to keep rows with matching keys in df2, but you don't need columns from df2.
-

6. Left Anti Join

Code:

```
left_anti_join = df1.join(df2, on="common_column", how="left_anti")
```

Explanation:

- **Purpose:** Returns only the rows from df1 that do **not** have a match in df2.
- **Behavior:** Filters out rows from df1 that have a match in df2.
- **Use Case:** When you want to filter df1 to keep rows with no matching keys in df2.

7. Cross Join

Code:

```
cross_join = df1.crossJoin(df2)
```

Explanation:

- **Purpose:** Returns the Cartesian product of df1 and df2, meaning every row of df1 is paired with every row of df2.
- **Behavior:** The number of rows in the result will be the product of the row count of df1 and df2.
- **Use Case:** Typically used in edge cases or for generating combinations of rows, but be cautious as it can result in a very large DataFrame.

8. Join with Explicit Conditions

Code:

```
inner_join = df1.join(df2, (df1["columnA"] == df2["columnB"]), "inner")
```

Explanation:

- **Purpose:** This is an example of an inner join where the common columns have different names in df1 and df2.
- **Behavior:** Joins df1 and df2 based on a condition where columnA from df1 matches columnB from df2.
- **Use Case:** When the join condition involves columns with different names or more complex conditions.

Conclusion:

- **Inner Join:** Matches rows from both DataFrames.
- **Left/Right Join:** Keeps all rows from the left or right DataFrame and matches where possible.
- **Full Join:** Keeps all rows from both DataFrames.
- **Left Semi:** Filters df1 to rows that match df2 without including columns from df2.
- **Left Anti:** Filters df1 to rows that do not match df2.
- **Cross Join:** Returns the Cartesian product, combining all rows of both DataFrames.
- **Explicit Condition Join:** Allows complex join conditions, including columns with different names.

These joins are highly useful for various types of data integration and analysis tasks in PySpark.

Master PySpark: From Zero to Big Data Hero!!

Joins Part 2

```
from pyspark.sql import SparkSession
from pyspark.sql import Row
from pyspark.sql.functions import broadcast

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

# Sample DataFrames
data1 = [Row(id=0), Row(id=1), Row(id=1), Row(id=None),
Row(id=None)]
data2 = [Row(id=1), Row(id=0), Row(id=None)]
df1 = spark.createDataFrame(data1)
df2 = spark.createDataFrame(data2)

# Inner Join
inner_join = df1.join(df2, on="id", how="inner")
print("Inner Join:")
inner_join.show()
```

```
Inner Join:
```

```
+---+
| id|
+---+
|  0|
|  1|
|  1|
+---+
```

```
# Left Join
left_join = df1.join(df2, on="id", how="left")
print("Left Join:")
left_join.show()
```

Left Join

```
+-----+
|  id|
+-----+
|   0|
|   1|
|   1|
|null|
|null|
+-----+
```

Right Join

```
right_join = df1.join(df2, on="id", how="right")
print("Right Join:")
right_join.show()
```

Right Join:

```
+-----+
|  id|
+-----+
|   1|
|   1|
|   0|
|null|
+-----+
```

Full (Outer) Join

```
full_join = df1.join(df2, on="id", how="outer")
print("Full (Outer) Join:")
full_join.show()
```

Full (Outer) Join:

```
+-----+
|  id|
+-----+
|null|
|null|
|null|
|   0|
|   1|
|   1|
+-----+
```

Left Anti Join

```
left_anti_join = df1.join(df2, on="id", how="left_anti")  
print("Left Anti Join:")  
left_anti_join.show()
```

Left Anti Join:

```
+----+  
|  id|  
+----+  
|null|  
|null|  
+----+
```

Right Anti Join (Equivalent to swapping DataFrames and performing Left Anti Join)

```
right_anti_join = df2.join(df1, on="id", how="left_anti")  
print("Right Anti Join:")  
right_anti_join.show()
```

Right Anti Join:

```
+----+  
|  id|  
+----+  
|null|  
+----+
```

Broadcast Join (Optimizing a join with a smaller DataFrame)

```
broadcast_join = df1.join(broadcast(df2), on="id", how="inner")  
print("Broadcast Join:")  
broadcast_join.show()
```

Broadcast Join:

```
+----+  
|  id|  
+----+  
|  0|  
|  1|  
|  1|  
+----+
```

Comparison: Left Join vs. Left Anti Join

- **Left Join:**
 - Keeps all rows from the left DataFrame and includes matching rows from the right, filling in null for unmatched rows.
- **Left Anti Join:**
 - Keeps only rows from the left DataFrame that do not have a match in the right DataFrame.
- **Summary:** Choose a left join to combine data and keep all rows from the left DataFrame. Use a left anti join to identify entries unique to the left DataFrame.

Broadcast Joins in PySpark

- **Definition:** A broadcast join optimizes joins when one DataFrame is small enough to fit into memory by broadcasting it to all nodes. This eliminates the need to shuffle data across the cluster, significantly improving performance for large datasets.
- **Usage:** Recommended for joining a large DataFrame with a small DataFrame (that can fit into memory). You can force a broadcast join using `broadcast(df_small)`.
- **Advantages:**
 - Avoids data shuffling, which can speed up processing for suitable cases.
 - Reduces memory consumption and network I/O for specific types of joins.

In summary:





- **Left Anti Join** is useful for identifying non-matching rows from one DataFrame against another.
- **Broadcast Join** is a performance optimization technique ideal for joining a large DataFrame with a small one efficiently, reducing shuffle costs.

These concepts help you manage and optimize your data processing tasks efficiently in PySpark.

Master PySpark: From Zero to Big Data Hero!!

Joins Part 3

Coding Question:

-  Write a PySpark query to find employees whose location matches the location of their department. Display emp_id, emp_name, emp_location, dept_name, and dept_location for matching records.
-  Modify the code to find departments that have no employees assigned to them. Display dept_id, dept_name, and dept_head.
-  Write a PySpark query to get the average salary of employees in each department, displaying dept_name and the calculated average_salary.
-  List the employees who earn more than the average salary of their department. Display emp_id, emp_name, emp_salary, dept_name, and dept_location.

Example → for joins with emp and dept data

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

Sample DataFrames

```
emp_data = [
    Row(emp_id=1, emp_name="Alice", emp_salary=50000,
emp_dept_id=101, emp_location="New York"),
    Row(emp_id=2, emp_name="Bob", emp_salary=60000,
emp_dept_id=102, emp_location="Los Angeles"),
    Row(emp_id=3, emp_name="Charlie", emp_salary=55000,
emp_dept_id=101, emp_location="Chicago"),
    Row(emp_id=4, emp_name="David", emp_salary=70000,
emp_dept_id=103, emp_location="San Francisco"),
    Row(emp_id=5, emp_name="Eve", emp_salary=48000,
emp_dept_id=102, emp_location="Houston")
]
```

```
dept_data = [
    Row(dept_id=101, dept_name="Engineering", dept_head="John",
dept_location="New York"),
    Row(dept_id=102, dept_name="Marketing", dept_head="Mary",
dept_location="Los Angeles"),
```

```
Row(dept_id=103, dept_name="Finance", dept_head="Frank",
dept_location="Chicago")
]
```

```
emp_columns = ["emp_id", "emp_name", "emp_salary", "emp_dept_id",
"emp_location"]
dept_columns = ["dept_id", "dept_name", "dept_head",
"dept_location"]
```

```
emp_df = spark.createDataFrame(emp_data, emp_columns)
dept_df = spark.createDataFrame(dept_data, dept_columns)
```

```
# Display emp data
```

```
print("emp_data:")
```

```
emp_df.show()
```

```
# Display dept data
```

```
print("dept_data:")
```

```
dept_df.show()
```

emp_data:

emp_id	emp_name	emp_salary	emp_dept_id	emp_location
1	Alice	50000	101	New York
2	Bob	60000	102	Los Angeles
3	Charlie	55000	101	Chicago
4	David	70000	103	San Francisco
5	Eve	48000	102	Houston

dept_data:

dept_id	dept_name	dept_head	dept_location
101	Engineering	John	New York
102	Marketing	Mary	Los Angeles
103	Finance	Frank	Chicago

```
# Inner Join on emp_dept_id and dept_id
```

```
inner_join = emp_df.join(dept_df, emp_df["emp_dept_id"] ==
dept_df["dept_id"], "inner")
```

```
# Display the result
print("Inner Join Result:")
inner_join.show()

# Inner Join with Filtering Columns and WHERE Condition
inner_join = emp_df.join(dept_df, emp_df["emp_dept_id"] ==
dept_df["dept_id"], "inner")\
    .select("emp_id", "emp_name", "emp_salary", "dept_name",
"dept_location")\
    .filter("emp_salary > 55000") # Add a WHERE condition

# Display the result
print("Inner Join with Filter and WHERE Condition:")
inner_join.show()
```

Inner Join Result:

emp_id	emp_name	emp_salary	emp_dept_id	emp_location	dept_id	dept_name	dept_head	dept_location
1	Alice	50000	101	New York	101	Engineering	John	New York
3	Charlie	55000	101	Chicago	101	Engineering	John	New York
2	Bob	60000	102	Los Angeles	102	Marketing	Mary	Los Angeles
5	Eve	48000	102	Houston	102	Marketing	Mary	Los Angeles
4	David	70000	103	San Francisco	103	Finance	Frank	Chicago

Inner Join with Filter and WHERE Condition:

emp_id	emp_name	emp_salary	dept_name	dept_location
2	Bob	60000	Marketing	Los Angeles
4	David	70000	Finance	Chicago

```
# Left Join with Filtering Columns and WHERE Condition
left_join_filtered = emp_df.join(dept_df, emp_df["emp_dept_id"] ==
dept_df["dept_id"], "left")\
    .select("emp_id", "emp_name", "dept_name", "dept_location")\
    .filter("emp_salary > 55000") # Add a WHERE condition

# Display the result
print("Left Join with Filter and WHERE Condition:")
left_join_filtered.show()
```

Left Anti Join

```
left_anti_join = emp_df.join(dept_df, emp_df["emp_dept_id"] ==  
dept_df["dept_id"], "left_anti")
```

Display the result

```
print("Left Anti Join Result:")  
left_anti_join.show()
```

Left Join with Filter and WHERE Condition:

```
+-----+-----+-----+-----+  
|emp_id|emp_name|dept_name|dept_location|  
+-----+-----+-----+-----+  
|      2|      Bob|Marketing|  Los Angeles|  
|      4|     David|  Finance|    Chicago|  
+-----+-----+-----+-----+
```

Left Anti Join Result:

```
+-----+-----+-----+-----+-----+  
|emp_id|emp_name|emp_salary|emp_dept_id|emp_location|  
+-----+-----+-----+-----+-----+  
+-----+-----+-----+-----+-----+
```

Left Join Result without filter:

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
|emp_id|emp_name|emp_salary|emp_dept_id| emp_location|dept_id| dept_name|dept_head|dept_location|  
+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
|      1|  Alice|    50000|      101|    New York|    101|Engineering|   John|    New York|  
|      2|   Bob|    60000|      102| Los Angeles|    102| Marketing|   Mary| Los Angeles|  
|      3| Charlie|    55000|      101|    Chicago|    101|Engineering|   John|    New York|  
|      4|  David|    70000|      103|San Francisco|    103|  Finance|  Frank|    Chicago|  
|      5|   Eve|    48000|      102|    Houston|    102| Marketing|   Mary| Los Angeles|  
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

Left Anti Join Result without filter:

```
+-----+-----+-----+-----+-----+  
|emp_id|emp_name|emp_salary|emp_dept_id|emp_location|  
+-----+-----+-----+-----+-----+  
+-----+-----+-----+-----+-----+
```

Master PySpark: From Zero to Big Data Hero!!

Joins Part 4

Coding Question:

1. Write a PySpark query to create a DataFrame that lists each employee along with their manager's name. Display columns employee and manager.
2. Modify the code to find and display only the employee(s) who do not have a manager (CEO-level employees). Display columns employee and manager.
3. Extend the code to find all employees who directly report to "Manager A." Display columns empid, ename, and mrgid.
4. Write a query to determine the hierarchy level of each employee, where the CEO is level 1, direct reports to the CEO are level 2, and so on. Display columns empid, ename, mrgid, and level.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, expr

# Create a Spark session
spark =
SparkSession.builder.appName("EmployeeHierarchy").getOrCreate()

# Sample data
data = [
    (1, None, "CEO"),
    (2, 1, "Manager A"),
    (3, 1, "Manager B"),
    (4, 2, "Employee X"),
    (5, 3, "Employee Y"),
]
columns = ["empid", "mrgid", "ename"]
employee_df = spark.createDataFrame(data, columns)
# Display the result
print("emp_data:")
employee_df.show()
```

```

# Self-join to find the manager and CEO
manager_df = employee_df.alias("e") \
    .join(employee_df.alias("m"), col("e.mrgid") == col("m.empid"),
"left") \
    .select(
        col("e.ename").alias("employee"),
        col("m.ename").alias("manager")
    )

# Display the result
print("mgr:")
manager_df.show()

# filter for employees without a manager (CEO)
manager_df2 = employee_df.alias("e1") \
    .join(employee_df.alias("m1"), col("e1.mrgid") ==
col("m1.empid"), "left") \
    .select(
        col("e1.ename").alias("employee"),
        col("m1.ename").alias("manager")
    ) \
    .filter(col("manager").isNull())

# Display the result
manager_df2.show()

```

emp_data:

empid	mrgid	ename
1	null	CEO
2	1	Manager A
3	1	Manager B
4	2	Employee X
5	3	Employee Y

mgr:

employee	manager
CEO	null
Manager A	CEO
Manager B	CEO
Employee X	Manager A
Employee Y	Manager B

Master PySpark: From Zero to Big Data Hero!!

Key Notes on when and otherwise

The when and otherwise functions in PySpark provide a way to create conditional expressions within a DataFrame, allowing you to specify different values for new or existing columns based on specific conditions.

when: The when function in PySpark is used to define a condition. If the condition is met, it returns the specified value. You can chain multiple when conditions to handle various cases.

otherwise: The otherwise function specifies a default value to return if none of the conditions in the when statements are met.

```
from pyspark.sql.functions import when

# Syntax to add a new column based on a condition
df = df.withColumn("new_column_name", when(condition1,
value1).when(condition2, value2).otherwise(default_value))
```

Example

Let's create a dataset and apply when and otherwise conditions.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import when
from pyspark.sql.types import StructType, StructField, IntegerType,
StringType

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

# Define the schema for the dataset
schema = StructType([
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True),
    StructField("salary", IntegerType(), True)
])
```

```
# Create a sample dataset
```

```
data = [  
    ("Alice", 25, 3000),  
    ("Bob", 35, 4000),  
    ("Charlie", 40, 5000),  
    ("David", 28, 4500),  
    ("Eve", 32, 3500)  
]
```

```
# Create DataFrame
```

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

```
df.show()
```

```
+-----+---+-----+  
|  name|age|salary|  
+-----+---+-----+  
|  Alice| 25|  3000|  
|   Bob| 35|  4000|  
|Charlie| 40|  5000|  
|  David| 28|  4500|  
|   Eve| 32|  3500|  
+-----+---+-----+
```

```
# Apply 'when' and 'otherwise' to add new columns based on  
conditions
```

```
df = (  
    df.withColumn("status", when(df.age < 30,  
"Young").otherwise("Adult"))  
    .withColumn("income_bracket", when(df.salary < 4000, "Low")  
    .when((df.salary >= 4000) &  
(df.salary <= 4500), "Medium")  
    .otherwise("High"))  
)
```

```
# Show the result
```

```
df.show()
```


name	age	salary	status	income_bracket
Alice	25	3000	Young	Low
Bob	35	4000	Adult	Medium
Charlie	40	5000	Adult	High
David	28	4500	Young	Medium
Eve	32	3500	Adult	Low

Explanation

1. **"status" column:** Assigns "Young" if age < 30, otherwise "Adult".
2. **"income_bracket" column:**
 - Assigns "Low" if salary < 4000.
 - Assigns "Medium" if salary is between 4000 and 4500.
 - Assigns "High" for any other salary values.

This approach allows for flexible handling of multiple conditions in PySpark DataFrames using when and otherwise.

Master PySpark: From Zero to Big Data Hero!!

Key Notes on cast() and printSchema()

In PySpark, the cast() function is used to change the data type of a column within a DataFrame. This is helpful when you need to standardize column data types for data processing, schema consistency, or compatibility with other operations.

- **Purpose:** The cast() function allows you to change the data type of a column, useful in situations like standardizing formats (e.g., converting strings to dates or integers).
- **Syntax:** The cast() function is applied on individual columns and requires specifying the target data type in quotes.
- **Multiple Columns:** You can cast multiple columns at once by using a list of cast expressions and passing them to select().
- **Supported Data Types:** PySpark supports various data types for casting, including:
 - StringType
 - IntegerType (or "int")
 - DoubleType (or "double")
 - DateType
 - TimestampType
 - BooleanType
 - Others, based on the data types available in PySpark.

Basic Syntax for cast()

```
from pyspark.sql.functions import col

# Single column cast
df = df.withColumn("column_name",
col("column_name").cast("target_data_type"))

# Multiple columns cast with select
cast_expr = [
    col("column1_name").cast("target_data_type1"),
    col("column2_name").cast("target_data_type2"),
    # More columns and data types as needed
]
df = df.select(*cast_expr)
```

Example

Let's create a dataset and apply `cast()` to change the data types of multiple columns

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
from pyspark.sql.types import StructType, StructField, StringType,
IntegerType, FloatType

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

# Define the schema for the dataset
schema = StructType([
    StructField("name", StringType(), True),
    StructField("age", StringType(), True),      # Stored as
StringType initially
    StructField("height", StringType(), True)   # Stored as
StringType initially
])

# Create a sample dataset
data = [
    ("Alice", "25", "5.5"),
    ("Bob", "35", "6.1"),
    ("Charlie", "40", "5.8"),
]

# Create DataFrame
df = spark.createDataFrame(data, schema)

# Assuming you have already created a DataFrame 'df'
df.printSchema()

df.show()
```

```
root
|-- name: string (nullable = true)
|-- age: string (nullable = true)
|-- height: string (nullable = true)
```

```
+-----+---+-----+
|  name|age|height|
+-----+---+-----+
|  Alice| 25|   5.5|
|   Bob| 35|   6.1|
|Charlie| 40|   5.8|
+-----+---+-----+
```

Define cast expressions for multiple columns

```
cast_expr = [
    col("name").cast("string"),
    col("age").cast("int"),          # Casting age to IntegerType
    col("height").cast("double")    # Casting height to DoubleType
]
```

Apply the cast expressions to the DataFrame

```
df = df.select(*cast_expr)
```

Show the result

```
df.printSchema()
df.show()
```

```
root
|-- name: string (nullable = true)
|-- age: integer (nullable = true)
|-- height: double (nullable = true)
```

```
+-----+---+-----+
|  name|age|height|
+-----+---+-----+
|  Alice| 25|   5.5|
|   Bob| 35|   6.1|
|Charlie| 40|   5.8|
+-----+---+-----+
```

Explanation

- **"age" column:** Initially stored as StringType, it's cast to IntegerType (or "int").
- **"height" column:** Initially stored as StringType, it's cast to DoubleType (or "double").

Advantages of Using cast()

- **Schema Alignment:** Ensures data types in different tables or DataFrames are compatible for joining or union operations.
- **Data Consistency:** Ensures all columns conform to expected data types for downstream data processing.
- **Error Reduction:** Minimizes issues arising from mismatched data types in computations or transformations.

This approach using cast() provides a flexible and powerful way to manage data types in PySpark.

printSchema() Method in PySpark

- **Purpose:**
 - To display the schema of a DataFrame, which includes the column names, data types, and nullability of each column.
- **Output Structure:**
 - The schema is presented in a tree-like structure showing:
 - **Column Name:** The name of the column.
 - **Data Type:** The data type of the column (e.g., string, integer, double, boolean, etc.).
 - **Nullability:** Indicates whether the column can contain null values (e.g., nullable = true).

Usage:

- Call `df.printSchema()` on a DataFrame `df` to see its structure.
- Useful for verifying the structure of the DataFrame after operations like `select()`, `withColumn()`, or `cast()`.

Master PySpark: From Zero to Big Data Hero!!

union and unionAll in PySpark

Overview

- **Purpose:** Both union and unionAll are used to combine two DataFrames into a single DataFrame.
- **DataFrame Compatibility:** The two DataFrames must have the same schema (i.e., the same column names and data types) to perform the union operation.

union()

- **Functionality:**
 - Combines two DataFrames and retains all rows, duplicate rows from the result.
- **Behavior:**
 - The union() method doesnot retains unique rows across both DataFrames, resulting in a DataFrame with duplicates.

unionAll()

- **Functionality:**
 - Combines two DataFrames and retains all rows, including duplicates.
- **Behavior:**
 - The unionAll() method performs the union operation but does not eliminate duplicate rows, similar to Unionall

Syntax

```
# Using union to retain all rows including duplicates
unioned_df = df1.union(df2)
```

```
# Using unionAll to retain all rows including duplicates
unionAll_df = df1.unionAll(df2)
```

Example Code

Here's a complete example demonstrating both union and unionAll:

```

from pyspark.sql import SparkSession

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

# Sample DataFrames
data1 = [("Alice", 25), ("Bob", 30), ("Charlie", 35)]
data2 = [("David", 40), ("Eve", 45), ("Alice", 25)]
columns = ["name", "age"]

df1 = spark.createDataFrame(data1, columns)
df2 = spark.createDataFrame(data2, columns)

# Using union to retain all rows including duplicates
unioned_df = df1.union(df2)

# Using unionAll to retain all rows
unionAll_df = df1.unionAll(df2)

# Show the results
print("unioned_df (No duplicates removed):")
unioned_df.show()

```

```
unioned_df (No duplicates removed in pyspark):
```

```

+-----+----+
|  name|age|
+-----+----+
|  Alice| 25|
|   Bob| 30|
|Charlie| 35|
|  David| 40|
|   Eve| 45|
|  Alice| 25|
+-----+----+

```

```

print("unionAll_df (duplicates retained):")
unionAll_df.show()

```

```
unionAll_df (duplicates retained):
```

```
+-----+---+
|  name|age|
+-----+---+
|  Alice| 25|
|    Bob| 30|
|Charlie| 35|
|  David| 40|
|    Eve| 45|
|  Alice| 25|
+-----+---+
```

Remove duplicate rows and create a new DataFrame

```
unique_df = unioned_df.dropDuplicates()
```

or

```
unique_df = unioned_df.distinct()
```

```
print("unique_df (after removing duplicates):")
```

```
unique_df.show()
```

```
unique_df (after removing duplicates):
```

```
+-----+---+
|  name|age|
+-----+---+
|  Alice| 25|
|    Bob| 30|
|Charlie| 35|
|  David| 40|
|    Eve| 45|
+-----+---+
```


Master PySpark: From Zero to Big Data Hero!!

Union and UnionByName in PySpark

In PySpark, both Union and UnionByName are operations that allow you to combine two or more DataFrames. However, they do this in slightly different ways, particularly regarding how they handle column names.

1. Union

Definition: The union() function is used to combine two DataFrames with the same schema (i.e., the same number of columns with the same data types). It appends the rows of one DataFrame to the other.

Key Characteristics:

- The DataFrames must have the same number of columns.
- The columns must have compatible data types.
- It does not automatically handle column names that differ between DataFrames.

Syntax:

```
DataFrame.union(otherDataFrame)
```

```
from pyspark.sql import SparkSession
# Create a Spark session
spark = SparkSession.builder.appName("Union Example").getOrCreate()

# Create two DataFrames with the same schema
data1 = [("Alice", 1), ("Bob", 2)]
data2 = [("Cathy", 3), ("David", 4)]

columns = ["Name", "Id"]

df1 = spark.createDataFrame(data1, columns)
df2 = spark.createDataFrame(data2, columns)

# Perform union
result_union = df1.union(df2)

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

```
+-----+-----+
| Name | Id |
+-----+-----+
| Alice | 1 |
| Bob   | 2 |
| Cathy | 3 |
| David | 4 |
+-----+-----+
```

2. UnionByName

Definition: The `unionByName()` function allows you to combine two DataFrames by matching column names. If the DataFrames do not have the same schema, it will fill in missing columns with null.

Key Characteristics:

- It matches DataFrames by column names rather than position.
- If the DataFrames have different columns, it will include all columns and fill in null for missing values in any DataFrame.
- You can specify `allowMissingColumns=True` to ignore missing columns.

Syntax:

```
DataFrame.unionByName(otherDataFrame, allowMissingColumns=False)
```

```
# Create two DataFrames with different schemas
```

```
data3 = [("Eve", 5), ("Frank", 6)]
```

```
data4 = [("Grace", "New York"), ("Hannah", "Los Angeles")]
```

```
columns1 = ["Name", "Id"]
```

```
columns2 = ["Name", "City"]
```

```
df3 = spark.createDataFrame(data3, columns1)
```

```
df4 = spark.createDataFrame(data4, columns2)
```

```
# Perform unionByName
```

```
result_union_by_name = df3.unionByName(df4,
allowMissingColumns=True)
```

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

```
Name: string
Id: long
City: string
```

```
+-----+-----+-----+
| Name|  Id|      City|
+-----+-----+-----+
|  Eve|   5|      null|
| Frank|   6|      null|
| Grace|null| New York|
|Hannah|null|Los Angeles|
+-----+-----+-----+
```

Summary of Differences

Feature	Union	UnionByName
Column Matching	Positional	By Name
Missing Columns Handling	Does not allow	Allows with <code>null</code> for missing
Schema Requirement	Must be identical	Can differ

Conclusion

In PySpark, use `union()` when you have DataFrames with the same schema and need a straightforward concatenation. Use `unionByName()` when your DataFrames have different schemas and you want to combine them by matching column names while handling missing columns.

Master PySpark: From Zero to Big Data Hero!!

Windows Function in PySpark

1. Introduction to Window Functions

Window functions allow you to perform calculations across a set of rows related to the current row within a specified partition. Unlike groupBy functions, window functions do not reduce the number of rows in the result; instead, they calculate a value for each row based on the specified window.

2. Importing Required Libraries

To use window functions, import the necessary modules from PySpark:

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.window import Window
```

3. Creating a Window Specification

A window specification defines how the rows will be grouped (partitioned) and ordered within each group.

Example: Basic Window Specification

```
window_spec = Window.partitionBy("category").orderBy("timestamp")
```

Example: Advanced Window Specification with Multiple Partition and Order Columns

```
window_spec = Window.partitionBy("category",
"sub_category").orderBy(F.col("timestamp"), F.col("score"))
```

4. Common Window Functions

Here are several commonly used window functions with explanations and examples:

a. Row Number

- **Function:** row_number()
- **Description:** Assigns a unique integer to each row within the partition. The numbering starts from 1.
- **Example**

```
df = df.withColumn("row_number", F.row_number().over(window_spec))
```

b. Rank

- **Function:** `rank()`
- **Description:** Assigns the same rank to rows with the same values in the order criteria. The next rank has a gap.
- **Example:**

```
df = df.withColumn("rank", F.rank().over(window_spec))
```

c. Dense Rank

- **Function:** `dense_rank()`
- **Description:** Similar to `rank()`, but does not leave gaps in the ranking.
- **Example:**

```
df = df.withColumn("dense_rank", F.dense_rank().over(window_spec))
```

d. Lead and Lag Functions

- **Functions:** `lead()`, `lag()`
- **Description:**
 - `lead()` returns the value of the next row within the window.
 - `lag()` returns the value of the previous row.

- **Example:**

- `df = df.withColumn("next_value", F.lead("value").over(window_spec))`
- `df = df.withColumn("previous_value", F.lag("value").over(window_spec))`

e. Aggregation Functions

Window functions can also be used to compute aggregated values over a specified window.

- **Example for Average:**

```
df = df.withColumn("avg_value", F.avg("value").over(window_spec))
```

- Other common aggregation functions that can be used include:
 - **Sum:** `F.sum("column_name").over(window_spec)`
 - **Min:** `F.min("column_name").over(window_spec)`
 - **Max:** `F.max("column_name").over(window_spec)`

5. Putting It All Together

Here's a complete example of how to use the various window functions in PySpark:

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.window import Window

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

# Sample DataFrame
data = [
    ("A", "X", 1, "2023-01-01"),
    ("A", "X", 2, "2023-01-02"),
    ("A", "Y", 3, "2023-01-01"),
    ("A", "Y", 3, "2023-01-02"),
    ("B", "X", 5, "2023-01-01"),
    ("B", "X", 4, "2023-01-02"),
]
columns = ["category", "sub_category", "value", "timestamp"]
df = spark.createDataFrame(data, columns)

# Define the window specification
window_spec = Window.partitionBy("category",
"sub_category").orderBy(F.col("timestamp"), F.col("value"))

# Apply window functions
df = df.withColumn("row_number", F.row_number().over(window_spec))
df = df.withColumn("rank", F.rank().over(window_spec))
df = df.withColumn("dense_rank", F.dense_rank().over(window_spec))
df = df.withColumn("next_value", F.lead("value").over(window_spec))
df = df.withColumn("previous_value",
F.lag("value").over(window_spec))
df = df.withColumn("avg_value", F.avg("value").over(window_spec))

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

category	sub_category	value	timestamp	row_number	rank	dense_rank	next_value	previous_value	avg_value
A	X	1	2023-01-01	1	1	1	2	null	1.0
A	X	2	2023-01-02	2	2	2	null	1	1.5
A	Y	3	2023-01-01	1	1	1	3	null	3.0
A	Y	3	2023-01-02	2	2	2	null	3	3.0
B	X	5	2023-01-01	1	1	1	4	null	5.0
B	X	4	2023-01-02	2	2	2	null	5	4.5

6. Conclusion

Window functions in PySpark are powerful tools for analyzing data within groups while retaining row-level detail. By understanding how to define window specifications and apply various functions, you can perform complex data analyses efficiently.

Master PySpark: From Zero to Big Data Hero!!

Windows Function in PySpark Part 2

```
from pyspark.sql import SparkSession
from pyspark.sql.window import Window
import pyspark.sql.functions as F

# Sample data
data = [
    ("Alice", 100),
    ("Bob", 200),
    ("Charlie", 200),
    ("David", 300),
    ("Eve", 400),
    ("Frank", 500),
    ("Grace", 500),
    ("Hank", 600),
    ("Ivy", 700),
    ("Jack", 800)
]

# Create a DataFrame
columns = ["Name", "Score"]
df = spark.createDataFrame(data, columns)
```

```
|  Name|Score|
+-----+-----+
|  Alice|  100|
|   Bob|  200|
|Charlie|  200|
|  David|  300|
|   Eve|  400|
|  Frank|  500|
|  Grace|  500|
|   Hank|  600|
|   Ivy|  700|
|   Jack|  800|
+-----+-----+
```



```
# Define a window specification
window_spec = Window.orderBy("Score")
```

```
# Using rank() to calculate rank
df1 = df.withColumn("Rank", F.rank().over(window_spec))
```

```
rank:
+-----+-----+
| Name | Score | Rank |
+-----+-----+
| Alice | 100 | 1 |
| Bob | 200 | 2 |
| Charlie | 200 | 2 |
| David | 300 | 4 |
| Eve | 400 | 5 |
| Frank | 500 | 6 |
| Grace | 500 | 6 |
| Hank | 600 | 8 |
| Ivy | 700 | 9 |
| Jack | 800 | 10 |
+-----+-----+
```

```
# Using dense_rank() to calculate dense rank
df2 = df.withColumn("DenseRank", F.dense_rank().over(window_spec))
```

```
dense_rank:
+-----+-----+
| Name | Score | DenseRank |
+-----+-----+
| Alice | 100 | 1 |
| Bob | 200 | 2 |
| Charlie | 200 | 2 |
| David | 300 | 3 |
| Eve | 400 | 4 |
| Frank | 500 | 5 |
| Grace | 500 | 5 |
| Hank | 600 | 6 |
| Ivy | 700 | 7 |
| Jack | 800 | 8 |
+-----+-----+
```

```
# Using row_number() to calculate row number
df3 = df.withColumn("RowNumber", F.row_number().over(window_spec))
```

```
Rownumber:
+-----+-----+
| Name | Score | RowNumber |
+-----+-----+
| Alice | 100 | 1 |
| Bob | 200 | 2 |
| Charlie | 200 | 3 |
| David | 300 | 4 |
| Eve | 400 | 5 |
| Frank | 500 | 6 |
| Grace | 500 | 7 |
| Hank | 600 | 8 |
| Ivy | 700 | 9 |
| Jack | 800 | 10 |
+-----+-----+
```

```
# Using lead() to calculate the difference with the next row
df4 = df.withColumn("ScoreDifferenceWithNext",
F.lead("Score").over(window_spec) - df["Score"])
```

lead:

Name	Score	ScoreDifferenceWithNext
Alice	100	100
Bob	200	0
Charlie	200	100
David	300	100
Eve	400	100
Frank	500	0
Grace	500	100
Hank	600	100
Ivy	700	100
Jack	800	null

```
# Using lag() to calculate the difference with the previous row
df5 = df.withColumn("ScoreDifferenceWithPrevious", df["Score"] -
F.lag("Score").over(window_spec))
```

lag:

Name	Score	ScoreDifferenceWithPrevious
Alice	100	null
Bob	200	100
Charlie	200	0
David	300	100
Eve	400	100
Frank	500	100
Grace	500	0
Hank	600	100
Ivy	700	100
Jack	800	100

Master PySpark: From Zero to Big Data Hero!!

Windows Function in PySpark Part 3

```
from pyspark.sql import SparkSession
from pyspark.sql.window import Window
import pyspark.sql.functions as F

# Updated sample data with students, different subjects, marks, and
semesters
data = [
    ("Alice", "Math", 90, 1),
    ("Alice", "Science", 85, 1),
    ("Alice", "History", 78, 1),
    ("Bob", "Math", 80, 1),
    ("Bob", "Science", 81, 1),
    ("Bob", "History", 77, 1),
    ("Charlie", "Math", 75, 1),
    ("Charlie", "Science", 82, 1),
    ("Charlie", "History", 79, 1),
    ("Alice", "Physics", 86, 2),
    ("Alice", "Chemistry", 92, 2),
    ("Alice", "Biology", 80, 2),
    ("Bob", "Physics", 94, 2),
    ("Bob", "Chemistry", 91, 2),
    ("Bob", "Biology", 96, 2),
    ("Charlie", "Physics", 89, 2),
    ("Charlie", "Chemistry", 88, 2),
    ("Charlie", "Biology", 85, 2),
    ("Alice", "Computer Science", 95, 3),
    ("Alice", "Electronics", 91, 3),
    ("Alice", "Geography", 97, 3),
    ("Bob", "Computer Science", 88, 3),
    ("Bob", "Electronics", 66, 3),
    ("Bob", "Geography", 92, 3),
    ("Charlie", "Computer Science", 92, 3),
    ("Charlie", "Electronics", 97, 3),
    ("Charlie", "Geography", 99, 3)
]
```

```
# Create a DataFrame
```

```
columns = ["First Name", "Subject", "Marks", "Semester"]  
df = spark.createDataFrame(data, columns)
```

```
# 1. Which student scored max marks in each semester considering  
all subjects
```

```
window_spec_max_marks =  
Window.partitionBy("Semester").orderBy(F.desc("Marks"))  
max_marks_df = df.withColumn("Rank",  
F.rank().over(window_spec_max_marks))  
top_scorer = max_marks_df.filter(max_marks_df["Rank"] == 1)  
print("top_scorer:")  
top_scorer.show()
```

```
top_scorer:
```

First Name	Subject	Marks	Semester	Rank
Alice	Math	90	1	1
Bob	Biology	96	2	1
Charlie	Geography	99	3	1

```
# 2. Percentage of each student considering all subjects
```

```
window_spec_total_marks = Window.partitionBy("First Name",  
"Semester")  
df = df.withColumn("TotalMarks",  
F.sum("Marks").over(window_spec_total_marks))  
df = df.withColumn("Percentage", (F.col("TotalMarks") / (3 *  
100)).cast("decimal(5, 2)"))*100)  
df2 = df.groupBy("First Name",  
"Semester").agg(F.max("TotalMarks").alias("TotalMarks"),  
F.max("Percentage").alias("Percentage"))  
print("percentage:")  
df2.show()
```

```
percentage:
```

First Name	Semester	TotalMarks	Percentage
Alice	1	253	84.00
Alice	2	258	86.00
Alice	3	283	94.00
Bob	1	238	79.00
Bob	2	281	94.00
Bob	3	246	82.00
Charlie	1	236	79.00
Charlie	2	262	87.00
Charlie	3	288	96.00

```
# 3. Who is the top rank holder in each semester considering all subjects
window_spec_rank =
Window.partitionBy("Semester").orderBy(F.desc("Percentage"))
rank_df = df.withColumn("Rank", F.rank().over(window_spec_rank))
top_rank_holder = rank_df.filter(rank_df["Rank"] ==
1).select("First Name", "Semester", "Rank", "Percentage").distinct()
print("top_rank_holder:")
top_rank_holder.show()
```

top_rank_holder:

First Name	Semester	Rank	Percentage
Alice	1	1	84.00
Bob	2	1	94.00
Charlie	3	1	96.00

```
# 4. Who scored max marks in each subject in each semester
window_spec_max_subject_marks = Window.partitionBy("Semester",
"Subject").orderBy(F.desc("Marks"))
max_subject_marks_df = df.withColumn("Rank",
F.rank().over(window_spec_max_subject_marks))
max_subject_scorer =
max_subject_marks_df.filter(max_subject_marks_df["Rank"] == 1)
print("max_subject_scorer")
max_subject_scorer.show()
```

max_subject_scorer

First Name	Subject	Marks	Semester	TotalMarks	Percentage	Rank
Charlie	History	79	1	236	79.00	1
Alice	Math	90	1	253	84.00	1
Alice	Science	85	1	253	84.00	1
Bob	Biology	96	2	281	94.00	1
Alice	Chemistry	92	2	258	86.00	1
Bob	Physics	94	2	281	94.00	1
Alice	Computer Science	95	3	283	94.00	1
Charlie	Electronics	97	3	288	96.00	1
Charlie	Geography	99	3	288	96.00	1

Master PySpark: From Zero to Big Data Hero!!

Windows Function in PySpark Part 4

highest sal in dept

```
from pyspark.sql import functions as F
from pyspark.sql.window import Window
# Updated sample data for employees
emp_data = [(1, "Alice", 1, 6300),
            (2, "Bob", 1, 6200),
            (3, "Charlie", 2, 7000),
            (4, "David", 2, 7200),
            (5, "Eve", 1, 6300),
            (6, "Frank", 2, 7100)]

# Sample data for departments
dept_data = [(1, "HR"),
            (2, "Finance")]

# Create DataFrames for employees and departments
emp_df = spark.createDataFrame(emp_data, ["EmpId", "EmpName",
                                           "DeptId", "Salary"])
dept_df = spark.createDataFrame(dept_data, ["DeptId", "DeptName"])
```

```
+-----+-----+-----+-----+
| EmpId | EmpName | DeptId | Salary |
+-----+-----+-----+-----+
| 1 | Alice | 1 | 6300 |
| 2 | Bob | 1 | 6200 |
| 3 | Charlie | 2 | 7000 |
| 4 | David | 2 | 7200 |
| 5 | Eve | 1 | 6300 |
| 6 | Frank | 2 | 7100 |
+-----+-----+-----+-----+
```

```
+-----+-----+
| DeptId | DeptName |
+-----+-----+
| 1 | HR |
| 2 | Finance |
+-----+-----+
```

```
# Window specification for ranking salaries within each department
window_spec =
Window.partitionBy("DeptId").orderBy(F.desc("Salary"))

# Add a rank column based on the highest salary within each
department
ranked_salary_df = emp_df.withColumn("Rank",
F.rank().over(window_spec))

# Filter to get only the top rank (highest salary) for each
department
result_df = ranked_salary_df.filter(F.col("Rank") == 1)
print("result_df")
result_df.show()
```

```
+-----+-----+-----+-----+
| EmpId | EmpName | DeptId | Salary | Rank |
+-----+-----+-----+-----+
|    1  |  Alice  |    1   |  6300  |    1  |
|    5  |   Eve   |    1   |  6300  |    1  |
|    4  |  David  |    2   |  7200  |    1  |
+-----+-----+-----+-----+
```

```
# Join the department names to get department names
result_df = result_df.join(dept_df, ["DeptId"], "left")

# Show the employees with the highest salary in each department
result_df.select("EmpName", "DeptName", "Salary").show()
```

```
+-----+-----+-----+
| EmpName | DeptName | Salary |
+-----+-----+-----+
|  Alice  |    HR   |  6300  |
|   Eve   |    HR   |  6300  |
|  David  | Finance |  7200  |
+-----+-----+-----+
```

Master PySpark: From Zero to Big Data Hero!!

Explode vs Explode_outer

In PySpark, `explode` and `explode_outer` are functions used to work with nested data structures, like arrays or maps, by “exploding” (flattening) each element of an array or key-value pair in a map into separate rows. The key difference between `explode` and `explode_outer` is in handling null or empty arrays, which makes them useful in different scenarios.

Here's a detailed breakdown of each function, including examples:

1. `explode()`

The `explode()` function takes a column with array or map data and creates a new row for each element in the array (or each key-value pair in the map). If the array is empty or null, `explode()` will drop the row entirely.

Key Characteristics

- Converts each element in an array or each entry in a map into its own row.
- **Drops rows** with null or empty arrays.

Syntax

```
from pyspark.sql.functions import explode
df.select(explode(df["column_with_array"])).show()

from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.functions import explode
# Initialize Spark session
spark =
SparkSession.builder.appName("ExplodeExample").getOrCreate()
# Sample DataFrame with arrays
data = [
    ("Alice", ["Math", "Science"]),
    ("Bob", ["History"]),
    ("Cathy", []), # Empty array
    ("David", None) # Null array
]
```



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

```
df.show()
```

```
+-----+-----+
| Name|      Subjects|
+-----+-----+
|Alice|[Math, Science]|
|  Bob|    [History]|
|Cathy|         []|
|David|         null|
+-----+-----+
```

```
# Use explode to flatten the array
```

```
exploded_df = df.select("Name",  
explode("Subjects").alias("Subject"))
```

```
# Show the result
```

```
exploded_df.show()
```

```
+-----+-----+
| Name|Subject|
+-----+-----+
|Alice|  Math|
|Alice|Science|
|  Bob|History|
+-----+-----+
```

Explanation:

- explode() expands the Subjects array into individual rows.
- Rows with empty ([]) or null arrays (None) are removed, which is why Cathy and David do not appear in the output.

2. explode_outer()

The explode_outer() function works similarly to explode(), but it keeps rows with null or empty arrays. When explode_outer() encounters a null or empty array, it still generates a row for that entry, with null as the value in the resulting column.

Key Characteristics

- Converts each element in an array or each entry in a map into its own row.
- **Retains rows** with null or empty arrays, using null values in the exploded column.

Syntax

```
# Use explode_outer to flatten the array while keeping null or empty rows
exploded_outer_df = df.select("Name", F.explode_outer("Subjects").alias("Subject"))

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

▶ (3) Spark Jobs

▶  exploded_outer_df: pyspark.sql.dataframe.DataFrame = [Name: string, Subject: string]

```
+-----+-----+
| Name|Subject|
+-----+-----+
|Alice|  Math|
|Alice|Science|
|  Bob|History|
|Cathy|  null|
|David|  null|
+-----+-----+
```

Explanation:

- `explode_outer()` expands the Subjects array into individual rows.
- Unlike `explode()`, rows with empty (`[]`) or null arrays (`None`) are kept in the result, with null values in the Subject column for these cases.

Summary Table of Differences

Function	Description	Null/Empty Arrays Behavior
<code>explode()</code>	Expands each element of an array or map into individual rows	Drops rows with null or empty arrays
<code>explode_outer()</code>	Similar to <code>explode()</code> , but retains rows with null or empty arrays	Keeps rows with null or empty arrays, filling with null

These functions are very useful when working with complex, nested data structures, especially when dealing with JSON or other hierarchical data.

Master PySpark Zero to Hero:

Pivot in PySpark

The pivot operation in PySpark is used to transpose rows into columns based on a specified column's unique values. It's particularly useful for creating wide-format data where values in one column become new column headers, and corresponding values from another column fill those headers.

Key Concepts

1. groupBy and pivot:

- The pivot method is typically used in combination with groupBy. You group by certain columns and pivot one column to create new columns.

2. Aggregation Function:

- You need to specify an aggregation function (like sum, avg, count, etc.) to fill the values in the pivoted columns.

3. Performance Consideration:

- Pivoting can be computationally expensive, especially with a high number of unique values in the pivot column. For better performance, explicitly specify the values to pivot if possible.

4. Syntax:

```
dataframe.groupBy("group_column").pivot("pivot_column").agg(aggregation_function)
```

Example Code: Pivot in PySpark

Sample Data

Imagine we have a DataFrame of sales data with the following schema:

Product	Region	Sales
A	North	100
B	North	150
A	South	200
B	South	300

We want to pivot the data so that regions (North, South) become columns and the sales values are aggregated.

Code Implementation

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import sum

# Create a Spark session
spark = SparkSession.builder.appName("PivotExample").getOrCreate()

# Create a sample DataFrame
data = [
    ("A", "North", 100),
    ("B", "North", 150),
    ("A", "South", 200),
    ("B", "South", 300)
]
columns = ["Product", "Region", "Sales"]

df = spark.createDataFrame(data, columns)

# Pivot the DataFrame
pivoted_df = df.groupBy("Product").pivot("Region").agg(sum("Sales"))

# Show the results
pivoted_df.show()
```

Output

Product	North	South
A	100	200
B	150	300

Explanation of Code

1. `groupBy("Product")`:

- Groups the data by the Product column.

2. `pivot("Region")`:

- Transforms unique values in the Region column (North, South) into new columns.

3. `agg(sum("Sales"))`:

- Computes the sum of Sales for each combination of Product and new columns created by the pivot.

Notes

- Explicit Pivot Values:** To improve performance, you can specify the pivot values explicitly:

```
df.groupBy("Product").pivot("Region", ["North", "South"]).agg(sum("Sales"))
```

- Handling Null Values:** If some combinations of groupBy and pivot values have no corresponding rows, the resulting cells will contain null.
- Alternative Aggregations:** You can use other aggregation functions like avg, max, min, etc.

This approach is commonly used in creating summary reports or preparing data for machine learning models where wide-format data is required.

Master PySpark Zero to Hero:

Unpivot in PySpark

The **unpivot** operation (also called **melting**) is used to transform a **wide-format table** into a **long-format table**. This means columns are turned into rows, effectively reversing the pivot operation. PySpark doesn't have a direct unpivot function like Pandas' melt, but you can achieve it using the **selectExpr** method or a combination of **stack** and other DataFrame transformations.

Key Concepts

1. Purpose of Unpivot:

- Simplifies data analysis by converting column headers into a single column (e.g., categorical variables).
- Ideal for scenarios where you need to aggregate data further or visualize it in a long format.

2. Syntax Overview:

- Use the **stack** function inside a **selectExpr** to unpivot.
- Stack reshapes the DataFrame by creating multiple rows for specified columns.

3. Performance:

- Unpivoting can generate many rows, especially if the original DataFrame is wide with numerous columns. Ensure your environment can handle the resulting data volume.

Example: Unpivot in PySpark

Sample Data

Suppose we have the following DataFrame:

Product	North	South	East	West
A	100	200	150	130
B	150	300	200	180

We want to unpivot it to the following format:

Product	Region	Sales
A	North	100
A	South	200
A	East	150
A	West	130
B	North	150
B	South	300
B	East	200
B	West	180

Code Implementation

```
from pyspark.sql import SparkSession

# Create a Spark session
spark = SparkSession.builder.appName("UnpivotExample").getOrCreate()

# Sample data
data = [
    ("A", 100, 200, 150, 130),
    ("B", 150, 300, 200, 180)
]
columns = ["Product", "North", "South", "East", "West"]

# Create the DataFrame
df = spark.createDataFrame(data, columns)

# Unpivot the DataFrame using stack
unpivoted_df = df.selectExpr(
    "Product",
    "stack(4, 'North', North, 'South', South, 'East', East, 'West', West) as (Region, Sales)"
)

# Show the results
unpivoted_df.show()
```

Explanation of Code

1. Input DataFrame:

- Each column (North, South, East, West) represents a region's sales for each product.

2. selectExpr with stack:

- The **stack** function takes two arguments:
 - The number of columns being unpivoted (4 in this case).
 - A sequence of column-value pairs: 'ColumnName1', ColumnValue1, 'ColumnName2', ColumnValue2,
- The result is two new columns: the first contains the column names (now rows, Region), and the second contains the corresponding values (Sales).

3. Aliasing Columns:

- The stack result is aliased as (Region, Sales) to give meaningful names to the new columns.

Alternative Methods

Using withColumn and union:

If stack isn't flexible enough, you can manually combine rows for each column:

```
from pyspark.sql import functions as F

# Create a DataFrame with union operations for unpivoting
north = df.select("Product", F.lit("North").alias("Region"), F.col("North").alias("Sales"))
south = df.select("Product", F.lit("South").alias("Region"), F.col("South").alias("Sales"))
east = df.select("Product", F.lit("East").alias("Region"), F.col("East").alias("Sales"))
west = df.select("Product", F.lit("West").alias("Region"), F.col("West").alias("Sales"))

# Combine all rows using union
unpivoted_df = north.union(south).union(east).union(west)

# Show results
unpivoted_df.show()
```

Notes

1. Performance Considerations:

- stack is efficient for unpivoting a large number of columns.
- The union method may become unwieldy for many columns, but it offers more control over the transformation process.

2. Dynamic Column Unpivoting: If the column names are not fixed (dynamic), you can:

- Collect the column names dynamically using `df.columns`.
- Construct the `selectExpr` or union queries programmatically.

3. Resulting Format:

- After unpivoting, the data will have more rows but fewer columns.
- Ensure downstream processes are optimized to handle the increased row count.

Unpivoting is a powerful operation for restructuring data and is frequently used in data preprocessing, reporting, and machine learning pipelines.

Key Differences in Apache Spark Components and Concepts

Hadoop vs. Spark Architecture

Aspect	Hadoop	Spark
Storage	Uses HDFS for storage	Uses in-memory processing for speed
Processing	MapReduce is disk-based	In-memory processing improves performance
Integration	Runs independently or with Hadoop ecosystem	Can run on top of Hadoop; more flexible
Complexity	More complex setup and deployment	Simpler to deploy and configure
Performance	Slower for iterative tasks due to disk I/O	Better performance for iterative tasks

RDD vs. DataFrame vs. Dataset

Aspect	RDD	DataFrame	Dataset
API Level	Low-level, more control	High-level, optimized with Catalyst	High-level, type-safe
Schema	No schema, unstructured	Uses schema for structured data	Strongly typed, compile-time type safety
Optimization	No built-in optimization	Optimized using Catalyst	Optimized using Catalyst, with type safety
Type Safety	No type safety	No compile-time type safety	Provides compile-time type safety
Performance	Less optimized for performance	Better performance due to optimizations	Combines type safety with optimization

Action vs. Transformation

Aspect	Action	Transformation
Execution	Triggers execution of the Spark job	Builds up a logical plan of data operations
Return Type	Returns results or output	Returns a new RDD/DataFrame
Evaluation	Eager evaluation; executes immediately	Lazy evaluation; executed when an action is triggered
Computation	Involves actual computation (e.g., collect())	Defines data transformations (e.g., map())
Performance	Can cause data processing; affects performance	Does not affect performance until an action is called

Map vs. FlatMap

Aspect	Map	FlatMap
Output	Returns one output element per input element	Can return zero or more output elements per input
Flattening	Does not flatten output	Flattens the output into a single level
Use Case	Suitable for one-to-one transformations	Suitable for one-to-many transformations
Complexity	Simpler, straightforward	More complex due to variable number of outputs
Examples	<code>map(x => x * 2)</code>	<code>flatMap(x => x.split(" "))</code>

GroupByKey vs ReduceByKey

Aspect	GroupByKey	ReduceByKey
Operation	Groups all values by key	Aggregates values with the same key
Efficiency	Can lead to high shuffling	More efficient due to partial aggregation
Data Movement	Requires shuffling of all values	Minimizes data movement through local aggregation
Use Case	Useful for simple grouping	Preferred for aggregations and reductions
Performance	Less efficient with large datasets	Better performance for large datasets

Repartition Vs Coalesce

Aspect	Repartition	Coalesce
Partitioning	Can increase or decrease the number of partitions	Only decreases the number of partitions
Shuffling	Involves full shuffle	Avoids full shuffle, more efficient
Efficiency	More expensive due to shuffling	More efficient for reducing partitions
Use Case	Used for increasing partitions or balancing load	Used for reducing partitions, typically after filtering
Performance	Can be costly for large datasets	More cost-effective for reducing partitions

Cache Vs Persist

Aspect	Cache	Persist
Storage Level	Defaults to MEMORY_ONLY	Can use various storage levels (e.g., MEMORY_AND_DISK)
Flexibility	Simplified, with default storage level	Offers more options for storage levels
Use Case	Suitable for simple caching scenarios	Suitable for complex caching scenarios requiring different storage levels
Implementation	Easier to use, shorthand for MEMORY_ONLY	More flexible, allows custom storage options
Performance	Suitable when memory suffices	More efficient when dealing with larger datasets and limited memory

Narrow Vs Wide Transformation

Aspect	Narrow Transformation	Wide Transformation
Partitioning	Each parent partition is used by one child partition	Requires data from multiple partitions
Shuffling	No shuffling required	Involves shuffling of data
Performance	More efficient and less costly	Less efficient due to data movement
Examples	map(), filter()	groupByKey(), join()
Complexity	Simpler and faster	More complex and slower due to data movement

Collect vs Take

Aspect	Collect	Take
Output	Retrieves all data from the RDD/DataFrame	Retrieves a specified number of elements
Memory Usage	Can be expensive and use a lot of memory	More memory-efficient
Use Case	Used when you need the entire dataset	Useful for sampling or debugging
Performance	Can cause performance issues with large data	Faster and more controlled
Action Type	Triggers full data retrieval	Triggers partial data retrieval

Broadcast Variable vs Accumulator

Aspect	Broadcast Variable	Accumulator
Purpose	Efficiently shares read-only data across tasks	Tracks metrics and aggregates values
Data Type	Data that is shared and read-only	Counters and sums, often numerical
Use Case	Useful for large lookup tables or configurations	Useful for aggregating metrics like counts
Efficiency	Reduces data transfer by broadcasting data once	Efficient for aggregating values across tasks
Mutability	Immutable, read-only	Mutable, can be updated during computation

Spark SQL vs DataFrame API

Aspect	Spark SQL	DataFrame API
Interface	Executes SQL queries	Provides a programmatic interface
Syntax	Uses SQL-like syntax	Uses function-based syntax
Optimization	Optimized with Catalyst	Optimized with Catalyst
Use Case	Preferred for complex queries and legacy SQL code	Preferred for programmatic data manipulations
Integration	Can integrate with Hive and other SQL databases	Provides a unified interface for different data sources

Spark Streaming Vs Structured Streaming

Aspect	Spark Streaming	Structured Streaming
Processing	Micro-batch processing	Micro-batch and continuous processing
API	RDD-based API	SQL-based API with DataFrame/Dataset support
Complexity	More complex and lower-level	Simplified with high-level APIs
Consistency	Can be less consistent due to micro-batches	Provides stronger consistency guarantees
Performance	Can be slower for complex queries	Better performance with optimizations

Shuffle vs MapReduce

Aspect	Shuffle	MapReduce
Operation	Data reorganization across partitions	Data processing model for distributed computing
Efficiency	Can be costly due to data movement	Designed for batch processing with high I/O
Performance	Affects performance based on the amount of data movement	Optimized for large-scale data processing but less efficient for iterative tasks
Use Case	Used in Spark for data redistribution	Used in Hadoop for data processing tasks
Implementation	Integrated into Spark operations	Core component of the Hadoop ecosystem

Union vs Join

Aspect	Union	Join
Operation	Combines two DataFrames/RDDs into one	Combines rows from two DataFrames/RDDs based on a key
Data Requirements	Requires same schema for both DataFrames/RDDs	Requires a common key for joining
Performance	Generally faster as it does not require key matching	Can be slower due to key matching and shuffling
Output	Stacks data vertically	Merges data horizontally based on keys
Use Case	Appending data or combining datasets	Merging related data based on keys

Executor vs Driver

Aspect	Executor	Driver
Role	Executes tasks and processes data	Coordinates and manages the Spark application
Memory	Memory allocated per executor for data processing	Memory used for managing application execution
Lifecycle	Exists throughout the application execution	Starts and stops the Spark application
Tasks	Runs the tasks assigned by the driver	Schedules and coordinates tasks and jobs
Parallelism	Multiple executors run in parallel	Single driver coordinates multiple executors

Checkpointing vs Caching

Aspect	Checkpointing	Caching
Purpose	Provides fault tolerance and reliability	Improves performance by storing intermediate data
Storage	Writes data to stable storage (e.g., HDFS)	Stores data in memory or on disk (depends on storage level)
Use Case	Used for recovery in case of failures	Used for optimizing repeated operations
Impact	Can be more costly and slow	Generally faster but not suitable for fault tolerance
Data	Data is written to external storage	Data is kept in memory or disk storage for quick access

ReduceByKey vs AggregateByKey

Aspect	ReduceByKey	AggregateByKey
Operation	Combines values with the same key using a function	Performs custom aggregation and combinatory operations
Efficiency	More efficient for simple aggregations	Flexible for complex aggregation scenarios
Shuffling	Involves shuffling but can be optimized	Can be more complex due to custom aggregation
Use Case	Suitable for straightforward aggregations	Ideal for advanced and custom aggregations
Performance	Generally faster for simple operations	Performance varies with complexity

SQL Context vs Hive Context vs Spark Session

Aspect	SQL Context	Hive Context	Spark Session
Purpose	Provides SQL query capabilities	Provides integration with Hive for SQL queries	Unified entry point for Spark functionality
Integration	Basic SQL capabilities	Integrates with Hive Metastore	Combines SQL, DataFrame, and Streaming APIs
Usage	Legacy, less functionality	Supports HiveQL and Hive UDFs	Supports all Spark functionalities including Hive
Configuration	Less flexible and older	Requires Hive setup and configuration	Modern and flexible, manages configurations
Capabilities	Limited to SQL queries	Extends SQL capabilities with Hive integration	Comprehensive access to all Spark features

Broadcast Join Vs Shuffle Join

Aspect	Broadcast Join	Shuffle Join
Operation	Broadcasts a small dataset to all nodes	Shuffles data across nodes for joining
Data Size	Suitable for small datasets	Suitable for larger datasets
Efficiency	More efficient for small tables	More suited for large datasets
Performance	Faster due to reduced shuffling	Can be slower due to extensive shuffling
Use Case	Use when one dataset is small relative to others	Use when both datasets are large

Spark Context vs Spark Session

Aspect	Spark Context	Spark Session
Purpose	Entry point for Spark functionality	Unified entry point for Spark functionalities
Lifecycle	Created before Spark jobs start	Manages the Spark application lifecycle
Functionality	Provides access to RDD and basic Spark functionality	Provides access to RDD, DataFrame, SQL, and Streaming APIs
Configuration	Configuration is less flexible	More flexible and easier to configure
Usage	Older, used for legacy applications	Modern and recommended for new applications

Structured Streaming vs Spark Streaming

Aspect	Structured Streaming	Spark Streaming
Processing	Micro-batch and continuous processing	Micro-batch processing
API	SQL-based API with DataFrame/Dataset support	RDD-based API
Complexity	Simplified and high-level	More complex and low-level
Consistency	Provides stronger consistency guarantees	Can be less consistent due to micro-batches
Performance	Better performance with built-in optimizations	Can be slower for complex queries

Partitioning vs Bucketing

Aspect	Partitioning	Bucketing
Purpose	Divides data into multiple partitions based on a key	Divides data into buckets based on a hash function
Usage	Used to optimize queries by reducing data scanned	Used to improve join performance and maintain sorted data
Shuffling	Reduces shuffling by placing related data together	Reduces shuffle during joins and aggregations
Data Layout	Data is physically separated based on partition key	Data is organized into fixed-size buckets
Performance	Improves performance for queries involving partition keys	Enhances performance for join operations