PySpark DataFrame Cheat Sheet

Getting Started with PySpark DataFrames

Initializing a SparkSession:

```
# Step 1: Import the required modules
from pyspark.sql import SparkSession

# Step 2: Create a SparkSession
spark = SparkSession.builder \
    .appName("PySpark DataFrame Tutorial") \  # Set a name for your Spark application
    .config("spark.some.config.option", "config-value") \  # Add any additional configurations if needed
    .getOrCreate()

# Step 3: Verify the SparkSession
print(spark.version)  # Print the Spark version to ensure the SparkSession is created successfully
```

Loading data from CSV

```
# Create a SparkSession
spark = SparkSession.builder \
    .appName("Loading Data from CSV") \
    .getOrCreate()

# Load data from CSV file into a DataFrame
csv_file_path = "path/to/your/csv/file.csv"
df_csv = spark.read.csv(csv_file_path, header=True, inferSchema=True)

# Show the first few rows of the DataFrame
df_csv.show()
```

Loading data from JSON

```
# Load data from JSON file into a DataFrame
json_file_path = "path/to/your/json/file.json"
df_json = spark.read.json(json_file_path)

# Show the first few rows of the DataFrame
df_json.show()
```

Loading data from Parquet

```
# Load data from Parquet file into a DataFrame
parquet_file_path = "path/to/your/parquet/file.parquet"
df_parquet = spark.read.parquet(parquet_file_path)

# Show the first few rows of the DataFrame
df_parquet.show()
```

DataFrame Basics

In PySpark, a DataFrame is a distributed collection of data organized into named columns.For example, consider the following simple DataFrame representing sales data with distinct rows and columns:

OrderID	Item	Quantity	UnitPrice
101	Apple	5	1.50
102	Banana	3	0.75
103	Orange	2	1.00

Creating DataFrames

PySpark's versatility extends beyond just working with structured data; it also offers the flexibility to create DataFrames from Resilient Distributed Datasets (RDDs) and various external data sources. Let's explore how you can harness these capabilities.

Creating DataFrames from RDDs

```
# Sample RDD
rdd = spark.sparkContext.parallelize( [ ( 1, "Alice" ), ( 2, "Bob" ), ( 3, "Charlie" ) ] )

# Convert RDD to DataFrame
df_from_rdd = rdd.toDF( [ "ID", "Name" ] )

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

Working with Various Data Sources

```
# Creating DataFrame from CSV
csv_path = "path/to/data.csv"
df_from_csv = spark.read.csv ( csv_path, header=True, inferSchema=True )

# Creating DataFrame from JSON
json_path = "path/to/data.json"
df_from_json = spark.read.json ( json_path )

# Creating DataFrame from Parquet
parquet_path = "path/to/data.parquet"
df_from_parquet = spark.read.parquet ( parquet_path )
```

Basic DataFrame Operations

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

# Create a SparkSession
spark = SparkSession.builder \
    .appName("DataFrame Operations Example") \
    .getOrCreate()

# Sample data for the DataFrame
data = [
    (1, "Alice", 28),
    (2, "Bob", 22),
    (3, "Charlie", 35)
]

# Define the DataFrame schema
columns = ["ID", "Name", "Age"]

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

Display DataFrame contents

Input:

```
# Display the first 20 rows of the DataFrame
df.show()
```

Output:

ID	Name	Age
1	Alice	28
2	Bob	22
3	Charlie	35

Check DataFrame schema

Input:

```
# Check the DataFrame schema
df.printSchema()
```

Output:

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

Selecting and Filtering Data

Original DataFrame:

ID	Name	Age
1	Alice	28
2	Bob	22
3	Charlie	35

Selecting specific columns from the DataFrame:

```
# Select specific columns from the DataFrame
selected_columns = df.select("Name", "Age")

# Show the DataFrame with selected columns
print("DataFrame with Selected Columns:")
selected_columns.show()
```

Output:

DataFrame with Selected Columns:

Name	Age
Alice	28
Bob	22
Charlie	35

Additionally, you can also use the col() function from pyspark.sql.functions to select columns. Here's how you can do it:

```
# Using col() function to select columns
selected_columns_v2 = df.select(col("Name"), col("Age"))

# Show the DataFrame with selected columns using col() function
print("DataFrame with Selected Columns (using col() function):")
selected_columns_v2.show()
```

Output:

DataFrame with Selected Columns (using col() function):

Name	Age
Alice	28
Bob	22
Charlie	35

Filtering Data Elements

Creating a sample DataFrame to demonstrate each method:

```
# Sample data for the DataFrame
data = [
    (1, "Alice", 28),
    (2, "Bob", 22),
    (3, "Charlie", 35),
    (4, "David", 30),
    (5, "Eva", 25)
]

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

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

Using filter() method

```
# Method 1: Using filter() method
filtered_df = df.filter(df.Age > 25)

# Show the filtered DataFrame
print("Filtered DataFrame (using filter()):")
filtered_df.show()
```

Output:

Filtered DataFrame (using filter()):

ID	Name	Age
1	Alice	28
3	Charlie	35
4	David	30

Using expr() Function

```
from pyspark.sql.functions import expr

# Method 3: Using expr() function
expr_filtered_df = df.filter(expr("Age > 25"))

# Show the filtered DataFrame
print("Filtered DataFrame (using expr() function):")
expr_filtered_df.show()
```

Output:

Filtered DataFrame (using expr() function):

ID	Name	Age
1	Alice	28
3	Charlie	35
4	David	30

Using where() Method

```
# Method 4: Using where() method
where_filtered_df = df.where(df.Age > 25)

# Show the filtered DataFrame
print("Filtered DataFrame (using where() method):")
where_filtered_df.show()
```

Filtering Data Elements

Output:

Filtered DataFrame (using where() method):

ID	Name	Age
1	Alice	28
3	Charlie	35
4	David	30

Data Manipulation and Transformation

A sample DataFrame to demonstrate each operation:

ID	Name	Age
1	Alice	28
2	Bob	22
3	Charlie	35

Adding new columns

```
# Adding a new column "City" with a default value
df_with_new_column = df.withColumn("City", lit("New York"))

# Show the DataFrame with the new column
print("DataFrame with New Column:")
df_with_new_column.show()
```

Output:

DataFrame with New Column:

ID	Name	Age	City
1	Alice	28	New York
2	Bob	22	New York
3	Charlie	35	New York

Renaming Columns

```
# Renaming the column "Name" to "Full_Name"
df_with_renamed_column = df.withColumnRenamed("Name", "Full_Name")

# Show the DataFrame with the renamed column
print("DataFrame with Renamed Column:")
df_with_renamed_column.show()
```

Output:

DataFrame with Renamed Column:

ID	Full_Name	Age
1	Alice	28
2	Bob	22
3	Charlie	35

Dropping Columns

```
# Dropping the column "Age"
df_dropped_column = df.drop("Age")

# Show the DataFrame with the dropped column
print("DataFrame with Dropped Column:")
df_dropped_column.show()
```

Output:

DataFrame with Dropped Column:

ID	Name
1	Alice
2	Bob
3	Charlie

Let's use a sample DataFrame to demonstrate some common data transformation examples:

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

# Create a SparkSession
spark = SparkSession.builder \
    .appName("Data Transformation") \
    .getOrCreate()

# Sample data for the DataFrame
data = [
    (1, "Alice", 28),
    (2, "Bob", 22),
    (3, "Charlie", 35)
]

# Define the DataFrame schema
columns = ["ID", "Name", "Age"]

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

Using Functions to Transform Data

```
# Using functions to transform data
df_with_status = df.withColumn("Status", when(col("Age") > 25, "Adult").otherwise("Young"))

# Show the DataFrame with the new "Status" column
print("DataFrame with 'Status' Column:")
df_with_status.show()
```

Output:

DataFrame with Dropped Column:

ID	Name	Age	Status
1	Alice	28	Adult
2	Bob	22	Young
3	Charlie	35	Adult

Data Manipulation and Transformation

Using Expressions to Transform Data

```
# Using expressions to transform data
df_with_id_name = df.withColumn("ID_Name", concat(col("ID"), lit("_"), col("Name")))

# Show the DataFrame with the new "ID_Name" column
print("DataFrame with 'ID_Name' Column:")
df_with_id_name.show()
```

Output:

DataFrame with 'ID_Name' Column:

ID	Name	Age	ID_Name
1	Alice	28	1_Alice
2	Bob	22	2_Bob
3	Charlie	35	3_Charlie

Using SQL-like Expressions

```
# Using SQL-like expressions to transform data
df_with_age_group = df.withColumn("Age_Group", expr("CASE WHEN Age <= 25 THEN 'Young' ELSE 'Adult' END"))

# Show the DataFrame with the new "Age_Group" column
print("DataFrame with 'Age_Group' Column:")
df_with_age_group.show()
```

Output:

DataFrame with 'Age_Group' Column:

ID	Name	Age	Age_Group
1	Alice	28	Adult
2	Bob	22	Young
3	Charlie	35	Adult

Aggregating and Grouping Data

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, sum, avg, count, min, max

# Create a SparkSession
spark = SparkSession.builder \
    .appName("Aggregation Functions") \
    .getOrCreate()

# Sample data for the DataFrame
data = [
    ("Alice", 100),
    ("Bob", 200),
    ("Charlie", 150),
    ("David", 120),
    ("Eva", 180)
]

# Define the DataFrame schema
columns = ["Name", "Salary"]

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

# Show the DataFrame
print("DataFrame:")
df.show()
```

Output:

DataFrame:

Name	Salary
Alice	100
Bob	200
Charlie	150
David	120
Eva	180

Aggregation using sum() Function

```
# Using sum() function to calculate total salary
total_salary = df.select(sum("Salary")).collect()[0][0]

print("Total Salary:", total_salary)
```

Output:

Total Salary: 750

Aggregation using avg() Function

```
# Using avg() function to calculate average salary
average_salary = df.select(avg("Salary")).collect()[0][0]

print("Average Salary:", average_salary)
```

Output:

Average Salary: 150.0

Aggregation using count() Function

```
# Using count() function to count the number of employees
employee_count = df.select(count("Name")).collect()[0][0]

print("Number of Employees:", employee_count)
```

Output:

Number of Employees: 5

Aggregating and Grouping Data

Aggregation using min() and max() Functions

```
# Using min() and max() functions to find minimum and maximum salary
min_salary = df.select(min("Salary")).collect()[0][0]
max_salary = df.select(max("Salary")).collect()[0][0]

print("Minimum Salary:", min_salary)
print("Maximum Salary:", max_salary)
```

Output:

Minimum Salary: 100
Maximum Salary: 200

Grouping Data Elements

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

# Create a SparkSession
spark = SparkSession.builder \
    .appName("Grouping Data") \
    .getOrCreate()

# Sample data for the DataFrame
data = [
    ("Alice", "Department A", 100),
    ("Bob", "Department B", 200),
    ("Charlie", "Department A", 150),
    ("David", "Department C", 120),
    ("Eva", "Department B", 180)
]

# Define the DataFrame schema
columns = ["Name", "Department", "Salary"]

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

# Show the DataFrame
print("DataFrame:")
df.show()
```

Output:

DataFrame:

Name	Department	Salary
Alice	Department A	100
Bob	Department B	200
Charlie	Department A	150
David	Department C	120
Eva	Department B	180

```
# Grouping data based on "Department" and calculating average salary
grouped_data = df.groupBy("Department").agg(avg("Salary").alias("AvgSalary"))

# Show the grouped data
print("Grouped Data:")
grouped_data.show()
```

Output:

DataFrame:

Department	AvgSalary
Department B	190.0
Department C	120.0
Department A	125.0

Joins and Combining DataFrames

A sample example to illustrate the different join types:

DataFrame 1:

ID	Name
1	Alice
2	Bob
3	Charlie

DataFrame 1:

ID	Role
2	Manager
3	Employee
4	Intern

Now, let's demonstrate different join types:

1. Inner Join:

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

2. Outer Join:

```
outer_join = df1.join(df2, on="ID", how="outer")
outer_join.show()
```

3. Left Join:

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

4. Right Join:

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

Let's now use a few examples to illustrate how to combine DataFrames using different join types.

Consider two sample DataFrames:

a. DataFrame orders:

order	product	quantity
101	apple	3
102	banana	2
103	orange	4

b. DataFrame products:

product	price
apple	1.5
banana	0.75
grape	2.0

Joins and Combining DataFrames

1. Inner Join:

```
inner_join = orders.join(products, on="product", how="inner")
inner_join.show()
```

Output:

product	order	quantity	price
apple	101	3	1.5
banana	102	2	0.75

2. Left Join:

```
left_join = orders.join(products, on="product", how="left")
left_join.show()
```

Output:

product	order	quantity	price
apple	101	3	1.5
banana	102	2	0.75
orange	103	5	null

3. Right Join:

```
right_join = orders.join(products, on="product", how="right")
right_join.show()
```

Output:

product	order	quantity	price
apple	101	3	1.5
banana	102	2	0.75
grape	null	null	2.0

4. Outer Join:

```
outer_join = orders.join(products, on="product", how="outer")
outer_join.show()
```

Output:

product	order	quantity	price
apple	101	3	1.5
banana	102	2	0.75
orange	103	5	null
grape	null	null	2.0

Handling Missing Data

Identifying and handling missing data is a crucial step in data preprocessing and analysis. PySpark provides several techniques to identify and handle missing data in DataFrames. Let's explore these techniques:

Identifying Missing Data

PySpark represents missing data as null values. You can use various methods to identify missing data in DataFrames:

- isNull() and isNotNull()
- dropna()
- fillna()

Handling Missing Data

- **Dropping Rows with Null Values**

```
df _without _null = df.dropna( )
```

- **Filling Null Values**

```
df _filled_mean = df.fillna( { 'age': df.select( avg( 'age' ) ).first( ) [ 0 ] } )
```

- **Imputation**

```
from pyspark.ml.feature import Imputer

imputer = Imputer( inputCols=[ 'age' ], outputCols=['imputed_age'])
imputed_df = imputer.fit(df).transform(df)
```

- **Adding an Indicator Column**

```
df_with_indicator = df.withColumn('age_missing', df['age'].isNull())
```

- **Handling Categorical Missing Data**

```
df_filled_category = df.fillna({'gender': 'unknown'})
```

Working with Dates and Timestamps

Sample DataFrame with date and timestamp data as mentioned below.

DataFrame:

Name	Date	Timestamp
Alice	2022-01-15	2022-01-15 08:30:00
Bob	2021-12-20	2021-12-20 15:45:00
Charlie	2022-02-28	2022-02-28 11:00:00

Current Date and Timestamp

```
from pyspark.sql.functions import current_date, current_timestamp

# Adding columns for current date and timestamp
df_with_current_date = df.withColumn("CurrentDate", current_date())
df_with_current_timestamp = df.withColumn("CurrentTimestamp", current_timestamp())

# Show the DataFrames
print("DataFrame with Current Date:")
df_with_current_date.show()

print("DataFrame with Current Timestamp:")
df_with_current_timestamp.show()
```

Date Difference

```
from pyspark.sql.functions import datediff

date_diff_df = df.withColumn("DaysSince", datediff(current_date(), col("Date")))

# Show the DataFrame with date difference
print("DataFrame with Date Difference:")
date_diff_df.show()
```

Working with Dates and Timestamps

Months Between

```
from pyspark.sql.functions import months_between

months_between_df = df.withColumn("MonthsBetween", months_between(current_date(), col("Date")))

# Show the DataFrame with months between
print("DataFrame with Months Between:")
months_between_df.show()
```

Date Addition and Subtraction

```
from pyspark.sql.functions import date_add, date_sub

date_add_df = df.withColumn("DatePlus10Days", date_add(col("Date"), 10))
date_sub_df = df.withColumn("DateMinus5Days", date_sub(col("Date"), 5))

# Show the DataFrames with date addition and subtraction
print("DataFrame with Date Addition:")
date_add_df.show()

print("DataFrame with Date Subtraction:")
date_sub_df.show()
```

Advanced DataFrame Operations

Window Functions

Suppose you have a DataFrame with sales data and you want to calculate the rolling average of sales for each product over a specific window size.

```
from pyspark.sql import SparkSession
from pyspark.sql.window import Window
from pyspark.sql.functions import col, avg

# Create a SparkSession
spark = SparkSession.builder \
    .appName("Window Functions") \
    .getOrCreate()

# Sample data for the DataFrame
data = [
    ("ProductA", "2022-01-01", 100),
    ("ProductA", "2022-01-02", 150),
    ("ProductA", "2022-01-03", 200),
    ("ProductA", "2022-01-04", 120),
    ("ProductA", "2022-01-05", 180),
    ("ProductB", "2022-01-01", 50),
    ("ProductB", "2022-01-02", 70),
    ("ProductB", "2022-01-03", 90),
]

# Define the DataFrame schema
columns = ["Product", "Date", "Sales"]

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

# Define the window specification
window_spec = Window.partitionBy("Product").orderBy("Date").rowsBetween(-1, 1)

# Calculate rolling average using window function
df_with_rolling_avg = df.withColumn("RollingAvg", avg(col("Sales")).over(window_spec))

# Show the DataFrame with rolling average
df_with_rolling_avg.show()
```

Pivot Tables

Suppose you have a DataFrame with sales data and you want to create a pivot table showing total sales for each product in different months.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
from pyspark.sql.pivot import PivotTable

# Create a SparkSession
spark = SparkSession.builder \
    .appName("Pivot Tables") \
    .getOrCreate()

# Sample data for the DataFrame
data = [
    ("ProductA", "2022-01-01", 100),
    ("ProductA", "2022-02-01", 150),
    ("ProductA", "2022-01-01", 200),
    ("ProductB", "2022-02-01", 120),
    ("ProductB", "2022-02-01", 180),
]

# Define the DataFrame schema
columns = ["Product", "Date", "Sales"]

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

# Create a pivot table
pivot_table = df.groupBy("Product").pivot("Date").sum("Sales")

# Show the pivot table
pivot_table.show()
```

PySpark SQL Cheat Sheet: SQL Functions for DataFrames

Filtering Data

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

# Create a SparkSession
spark = SparkSession.builder \
    .appName("PySpark SQL Functions") \
    .getOrCreate()

# Sample data for the DataFrame
data = [
    ("Alice", 28),
    ("Bob", 22),
    ("Charlie", 35)
]

# Define the DataFrame schema
columns = ["Name", "Age"]

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

# Using SQL-like functions to filter data
filtered_data = df.filter(col("Age") > 25)
selected_columns = df.select("Name", "Age")

# Show the filtered and selected data
print("Filtered Data:")
filtered_data.show()

print("Selected Columns:")
selected_columns.show()
```

Aggregation

```
from pyspark.sql.functions import avg, max

# Using SQL-like functions for aggregation
grouped_data = df.groupBy("Age").agg(avg("Age"), max("Age"))

# Show the aggregated data
print("Aggregated Data:")
grouped_data.show()
```

PySpark SQL Cheat Sheet: SQL Functions for DataFrames

Sorting Data

```
from pyspark.sql.functions import desc

# Using SQL-like functions for sorting
sorted_data = df.orderBy("Age")
desc_sorted_data = df.sort(col("Age").desc())

# Show the sorted data
print("Sorted Data:")
sorted_data.show()

print("Descending Sorted Data:")
desc_sorted_data.show()
```

Joining DataFrames

```
# Sample data for another DataFrame
data2 = [
    ("Alice", "New York"),
    ("Bob", "San Francisco"),
    ("Eva", "Los Angeles")
]

columns2 = ["Name", "City"]
df2 = spark.createDataFrame(data2, columns2)

# Using SQL-like functions to join DataFrames
joined_data = df.join(df2, on="Name", how="inner")

# Show the joined data
print("Joined Data:")
joined_data.show()
```

Performance Optimization Tips

Optimizing PySpark DataFrame operations is essential to achieve better performance and efficient data processing. Here are some tips and best practices to consider:

1. Use Lazy Evaluation: PySpark uses lazy evaluation, which means that transformations on DataFrames are not executed immediately but are queued up. This allows PySpark to optimize and optimize the execution plan before actually performing computations.

2. Caching: Caching involves storing a DataFrame or RDD in memory so that it can be reused efficiently across multiple operations. Use .cache() or .persist() to cache intermediate DataFrames that are used in multiple transformations or actions.

3. Partitioning: Partitioning involves dividing your data into smaller subsets (partitions) based on certain criteria. This can significantly improve query performance as it reduces the amount of data that needs to be processed. Use .repartition() or .coalesce() to manage partitions.

4. Broadcasting: Broadcasting is a technique where smaller DataFrames are distributed to worker nodes and cached in memory for join operations. This is particularly useful when you have a small DataFrame that needs to be joined with a larger one.

5. Avoid Using .collect(): Using .collect() brings all the data to the driver node, which can lead to memory issues. Instead, try to perform operations using distributed computations.

6. Use Built-in Functions: Whenever possible, use built-in functions from the pyspark.sql.functions module. These functions are optimized for distributed processing and provide better performance than custom Python functions.

7. Avoid Shuffling: Shuffling is an expensive operation that involves data movement between partitions. Minimize shuffling by using appropriate partitioning and avoiding operations that require data to be rearranged.

8. Optimize Joins: Joins can be performance-intensive. Try to avoid shuffling during joins by ensuring both DataFrames are appropriately partitioned and using the appropriate join strategy (broadcast, sortMerge, etc.).

9. Use explain(): Use the explain() method on DataFrames to understand the execution plan and identify potential optimization opportunities.

10. Hardware Considerations: Consider the cluster configuration, hardware resources, and the amount of data being processed. Properly allocating resources and scaling up the cluster can significantly impact performance.

11. Monitor Resource Usage: Keep an eye on resource usage, including CPU, memory, and disk I/O. Monitoring can help identify performance bottlenecks and resource constraints.

12. Use Parquet Format: Parquet is a columnar storage format that is highly efficient for both reading and writing. Consider using Parquet for storage as it can improve read and write performance.

ProjectPro