Databricks pyspark.sql module

CLOUD DATA DRIVEN – OCTOBER 2023

About me

- ✓ Passionate about Data ☺
- ✓ Founder at Softentity, 8+ years of experience in the industry
- ✓ Certified Microsoft Data Engineer Associate
- ✓ Certified Microsoft Power BI Analyst Associate
- ✓ Databricks Certified Data Engineer Associate

Connect with me:
LinkedIn: https://www.linkedin.com/in/iulianatuhasu/
Email: tuhasu.Iuliana@gmail.com
Blog: iulianatuhasu.com

Introduction

Apache Spark = is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters.

PySpark = Python API for Apache Spark

- Real-time, large-scale data processing
- Supports Spark’s features such as: Spark SQL and Dataframes, Structured Streaming, Mlib, Pandas API on Spark and Spark Core

2 approaches of processing data:

- ✓ Spark SQL = SQL-like syntax for working with structured data
- ✓ Dataframe API = programmatic interface for working with structured data

Spark SQL vs Dataframe API

- Dataframe example

```
+---+-------+---+-----------------+
| id|   name|age|     emailaddress|
+---+-------+---+-----------------+
|  1|   John| 25|   john@example.com|
|  2|  Alice| 30|  alice@example.com|
|  3|    Bob| 28|    bob@example.com|
|  4|    Eve| 35|    eve@example.com|
|  5|Charlie| 22|charlie@example.com|
+---+-------+---+-----------------+
```

```
df.select("id").show()
```

=

```
spark.sql("SELECT id FROM table_name").show()
```

Core classes

from pyspark.sql import *class*

pyspark.sql.SparkSession Main entry point for **DataFrame** and SQL functionality.

pyspark.sql.DataFrame A distributed collection of data grouped into named columns.

pyspark.sql.Column A column expression in a **DataFrame**.

pyspark.sql.Row A row of data in a **DataFrame**.

pyspark.sql.GroupedData Aggregation methods, returned by **DataFrame.groupBy()**.

pyspark.sql.DataFrameNaFunctions Methods for handling missing data (null values).

pyspark.sql.DataFrameStatFunctions Methods for statistics functionality.

pyspark.sql.functions List of built-in functions available for **DataFrame**.

pyspark.sql.types List of data types available.

pyspark.sql.Window For working with window functions.

How to create a dataframe

```
# Create dataframe
df = spark.createDataFrame([("1", "John", 25, "john@example.com"),
                            ("2", "Alice", 30, "alice@example.com"),
                            ("3", "Bob", 28, "bob@example.com"),
                            ("4", "Eve", 35, "eve@example.com"),
                            ("5", "Charlie", 22, "charlie@example.com")],
                           ["id", "name", "age", "emailaddress"])
```

print(df)

```
DataFrame[id: string, name: string, age: bigint, emailaddress: string]
```

df.show()

```
+---+-------+---+-------------------+
| id|   name|age|       emailaddress|
+---+-------+---+-------------------+
|  1|   John| 25|   john@example.com|
|  2|  Alice| 30|  alice@example.com|
|  3|    Bob| 28|    bob@example.com|
|  4|    Eve| 35|    eve@example.com|
|  5|Charlie| 22|charlie@example.com|
+---+-------+---+-------------------+
```

How to create a dataframe – from CSV

```
# Create dataframe from CSV - without taking into account the header
df_csv_without_header = spark.read.csv("/FileStore/tables/pyspark-demo/sample1.csv")
```

```
print(df_csv_without_header)
```

```
DataFrame[_c0: string, _c1: string, _c2: string, _c3: string]
```

df_csv_without_header.show()

```
+---+-------+---+-----------------+
|_c0|    _c1|_c2|              _c3|
+---+-------+---+-----------------+
| id|   name|age|     emailaddress|
|  1|   John| 25|   john@example.com|
|  2|  Alice| 30|  alice@example.com|
|  3|    Bob| 28|    bob@example.com|
|  4|    Eve|  5|    eve@example.com|
|  5|Charlie| 22|charlie@example.com|
+---+-------+---+-----------------+
```

```
# Create dataframe from CSV - by taking into account the header
df_csv_with_header = spark.read.option("header", "true").csv("/FileStore/tables/pyspark-demo/sample1.csv")
```

```
print(df_csv_with_header)
```

```
DataFrame[id: string, name: string, age: string, emailaddress: string]
```

df_csv_with_header.show()

```
+---+-------+---+-------------------+
| id|   name|age|       emailaddress|
+---+-------+---+-------------------+
|  1|   John| 25|   john@example.com|
|  2|  Alice| 30|  alice@example.com|
|  3|    Bob| 28|    bob@example.com|
|  4|    Eve| 35|    eve@example.com|
|  5|Charlie| 22|charlie@example.com|
+---+-------+---+-------------------+
```

How to create a dataframe – from JSON

```
# Create dataframe from JSON
df_json = spark.read.json("/FileStore/tables/pyspark-demo/sample3.json")
```

print(df_json)

```
DataFrame[age: bigint, emailaddress: string, id: bigint, name: string]
```

<u>JSON file example:</u>

[{"id":1,"name":"John","age":25,"emailaddress":"john@example.com"},{"id":2,"name":"Alice","age":30,"emailaddress":"alice@example.com"},{"id":3,"name":"Bob","age":28,"emailaddress":"bob@example.com"},{"id":4,"name":"Eve","age":5,"emailaddress":"eve@example.com"},{"id":5,"name":"Charlie","age":22,"emailaddress":"charlie@example.com"}]

df_json.show()

```
+---+-------------------+---+-------+
|age|       emailaddress| id|   name|
+---+-------------------+---+-------+
| 25|   john@example.com|  1|   John|
| 30|  alice@example.com|  2|  Alice|
| 28|    bob@example.com|  3|    Bob|
|  5|    eve@example.com|  4|    Eve|
| 22|charlie@example.com|  5|Charlie|
+---+-------------------+---+-------+
```

How to create a dataframe – from JSON

By specifying a schema (pyspark.sql.types) – part 1

```
# Create dataframe from JSON by specifying a schema
custom_schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True),
    StructField("emailaddress", StringType(), True)
])
df_json_schema = spark.read.schema(custom_schema).json("/FileStore/tables/pyspark-demo/sample3.json")
```

df_json_schema.show()

```
+---+-------+---+-----------------+
| id|   name|age|     emailaddress|
+---+-------+---+-----------------+
|  1|   John| 25|   john@example.com|
|  2|  Alice| 30|  alice@example.com|
|  3|    Bob| 28|    bob@example.com|
|  4|    Eve|  5|    eve@example.com|
|  5|Charlie| 22|charlie@example.com|
+---+-------+---+-----------------+
```

print(df_json_schema)

```
DataFrame[id: int, name: string, age: int, emailaddress: string]
```

How to create a dataframe – from JSON
By specifying a schema (pyspark.sql.types) – part 2

What happens if the imported JSON contains more columns than defined in the schema?

```
# Create dataframe from JSON by specifying a schema
custom_schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True),
    StructField("emailaddress", StringType(), True)
])
```

```
df_json_schema_more_cols = spark.read.schema(custom_schema).json("/FileStore/tables/pyspark-demo/sample4.json")
```

df_json_schema_more_cols.show()

```
+---+-------+---+-----------------+
| id|   name|age|     emailaddress|
+---+-------+---+-----------------+
|  1|   John| 25|   john@example.com|
|  2|  Alice| 30|  alice@example.com|
|  3|    Bob| 28|    bob@example.com|
|  4|    Eve|  5|    eve@example.com|
|  5|Charlie| 22|charlie@example.com|
+---+-------+---+-----------------+
```

<u>JSON file example:</u>

[{"id":1,"name":"John","age":25,"emailaddress":"john@example.com","phoneNumber":"123-456-7890"},{"id":2,"name":"Alice","age":30,"emailaddress":"alice@example.com","phoneNumber":"987-654-3210"},{"id":3,"name":"Bob","age":28,"emailaddress":"bob@example.com","phoneNumber":"555-555-5555"},{"id":4,"name":"Eve","age":5,"emailaddress":"eve@example.com","phoneNumber":"777-123-4567"},{"id":5,"name":"Charlie","age":22,"emailaddress":"charlie@example.com","phoneNumber":"444-777-8888"}]

Quick tip – print vs printSchema

```
print(df)
```

```
DataFrame[id: string, name: string, age: bigint, emailaddress: string]
```

```
df.printSchema()
```

```
root
 |-- id: string (nullable = true)
 |-- name: string (nullable = true)
 |-- age: long (nullable = true)
 |-- emailaddress: string (nullable = true)
```

Save dataframe as CSV, Parquet, JSON

```
# Save df as Parquet and overwrite the previous file
df.write.mode("overwrite").parquet("/FileStore/tables/pyspark-demo/output_1/parquet_output_overwrite")

# Save df as CSV with custom delimiter and header
df.write.option("delimiter", ",").option("header", "true").csv("/FileStore/tables/pyspark-demo/output_1/csv_output_custom_delimiter")

# Save df as JSON and compress it
df.write.option("compression", "gzip").json("/FileStore/tables/pyspark-demo/output_1/json_output_gzip")
```

General Dataframe functions

List columns of a dataframe -> **df.columns**

Get the number of records -> **df.count()**

Sort the dataframe by a specified column -> **df.orderBy(desc(“column_name"))**

Create a list of rows from a dataframe -> **df.collect()**

Dataframes – Columns – Select & Add

```
# Select 2 columns
selected_columns = df.select(col("id"), col("name"))
print("DataFrame with only 2 columns:")
selected_columns.show()

# Add another column
df_with_added_column = df.withColumn("is_valid", col("id").isNotNull())
print("DataFrame with the new column is_valid):")
df_with_added_column.show()
```

```
DataFrame with only 2 columns:
+---+-------+
| id|   name|
+---+-------+
|  1|   John|
|  2|  Alice|
|  3|    Bob|
|  4|    Eve|
|  5|Charlie|
+---+-------+
```

```
DataFrame with the new column is_valid):
+---+-------+---+-------------------+--------+
| id|   name|age|       emailaddress|is_valid|
+---+-------+---+-------------------+--------+
|  1|   John| 25|   john@example.com|    true|
|  2|  Alice| 30|  alice@example.com|    true|
|  3|    Bob| 28|    bob@example.com|    true|
|  4|    Eve| 35|    eve@example.com|    true|
|  5|Charlie| 22|charlie@example.com|    true|
+---+-------+---+-------------------+--------+
```

Dataframes – Columns – Remove & Change type

```
# Remove a column
df_without_email = df.drop("emailaddress")
print("DataFrame without email address field:")
df_without_email.show()

# Change data type of age column
df_with_changed_data_type = df.withColumn("age", col("age").cast("string"))
print("DataFrame with column age as string data type instead of int:")
df_with_changed_data_type.show()
```

```
DataFrame without email address field:
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  1|   John| 25|
|  2|  Alice| 30|
|  3|    Bob| 28|
|  4|    Eve| 35|
|  5|Charlie| 22|
+---+-------+---+
```

```
DataFrame with column age as string data type instead of int:
+---+-------+---+-----------------+
| id|   name|age|     emailaddress|
+---+-------+---+-----------------+
|  1|   John| 25|   john@example.com|
|  2|  Alice| 30|  alice@example.com|
|  3|    Bob| 28|    bob@example.com|
|  4|    Eve| 35|    eve@example.com|
|  5|Charlie| 22|charlie@example.com|
+---+-------+---+-----------------+
```

Dataframes – Rows – Retrieve & Filter

```
# Get the first row
first_row = df.first()
```

```
Row(id='1', name='John', age=25, emailaddress='john@example.com', department='IT')
```

```
# Get the second row - collect the df into a list of rows
rows = df.collect()
# Access the second row
second_row = rows[1]
```

```
Row(id='2', name='Alice', age=30, emailaddress='alice@example.com', department='IT')
```

```
second_row[0]
```

```
First element of the second row is: 2
```

```
# Filter rows based on a condition
filtered_rows = df.filter(df["age"] > 28)
```

```
+---+-----+---+-----------------+----------+
| id| name|age|     emailaddress|department|
+---+-----+---+-----------------+----------+
|  2|Alice| 30|alice@example.com|        IT|
|  4|  Eve| 35|  eve@example.com|     Sales|
+---+-----+---+-----------------+----------+
```

Dataframes – Rows – Add & Remove

Add a new row

```
# Create a new row and append it to the DataFrame
new_row = Row(id="6", name="David", age=26, emailaddress="david@example.com", department="HR")
column_names = df.columns
df = df.union(spark.createDataFrame([new_row], column_names))
```

Remove a row

```
# Remove rows with a specific condition
df = df.filter(df["name"] != "David")
```

Dataframes – UPDATE

```
# Update age column with 32 for Alice
df_updated = df.withColumn("age", when(col("name") == "Alice", 32).otherwise(col("age")))
```

column to be updated **condition** **value if condition = true** **value if condition = false**

Dataframes – JOIN

```
joined_df_inner = df1.join(df2, "id", "inner")
```

first dataframe second dataframe join column join type

Dataframes – Grouped Data – COUNT() & MAX()

```
# Group by department and count employees
result_1 = df.groupBy("department").count()
result_1.show()
```

```
+----------+-----+
|department|count|
+----------+-----+
|        IT|    2|
|     Sales|    2|
|        HR|    1|
+----------+-----+
```

```
# Group the data by the department column and find the employee with the highest age in each department
result_2 = df.groupBy("department").agg(max("age").alias("oldest_employee_age"))
result_2.show()
```

```
+----------+-------------------+
|department|oldest_employee_age|
+----------+-------------------+
|        IT|                 30|
|     Sales|                 35|
|        HR|                 22|
+----------+-------------------+
```

Dataframes – Grouped Data – AVG() vs Dictionary

```
# Approach 1 - Group the data by the department column and calculate the average age of employees in each department
result_3 = df.groupBy("department").agg({"age": "avg"})
result_3.show()
```

```
+----------+--------+
|department|avg(age)|
+----------+--------+
|        IT|    27.5|
|     Sales|    31.5|
|        HR|    22.0|
+----------+--------+
```

```
# Approach 2 - Group the data by the department column and calculate the average age of employees in each department
result_3_1 = df.groupBy("department").agg(avg("age").alias("avg_age"))
result_3_1.show()
```

```
+----------+-------+
|department|avg_age|
+----------+-------+
|        IT|   27.5|
|     Sales|   31.5|
|        HR|   22.0|
+----------+-------+
```

Dataframes – Grouped Data – COLLECT_LIST()

```
# Group the data by the department column and create a list of employee names in each department
result_4 = df.groupBy("department").agg(collect_list("name").alias("employee_names"))
result_4.show()
```

```
+----------+--------------+
|department|employee_names|
+----------+--------------+
|        IT| [John, Alice]|
|     Sales|    [Bob, Eve]|
|        HR|     [Charlie]|
+----------+--------------+
```

```
# Group by 2 columns
result_5 = df.groupBy("department","age").agg(count("department"))
result_5.show()
```

```
+----------+---+-----------------+
|department|age|count(department)|
+----------+---+-----------------+
|        IT| 25|                1|
|        IT| 30|                1|
|     Sales| 28|                1|
|        HR| 22|                1|
|     Sales| 35|                1|
+----------+---+-----------------+
```

Dataframes – Window functions - ROW_NUMBER()

```
# Define the Window
window_spec_1 = Window.partitionBy("department").orderBy(F.desc("age"))

# Calculate the row_number of each row partitioned by dep and ordered by age
df.withColumn("row_number", F.row_number().over(window_spec_1)).orderBy("id").show()
```

```
+---+-------+---+------------------+----------+
| id|   name|age|      emailaddress|department|
+---+-------+---+------------------+----------+
|  1|   John| 25|  john@example.com|        IT|
|  2|  Alice| 30| alice@example.com|        IT|
|  3|    Bob| 28|   bob@example.com|     Sales|
|  4|    Eve| 35|   eve@example.com|     Sales|
|  5|Charlie| 22|charlie@example.com|        HR|
+---+-------+---+------------------+----------+
```

→

```
+---+-------+---+------------------+----------+----------+
| id|   name|age|      emailaddress|department|row_number|
+---+-------+---+------------------+----------+----------+
|  1|   John| 25|  john@example.com|        IT|         2|
|  2|  Alice| 30| alice@example.com|        IT|         1|
|  3|    Bob| 28|   bob@example.com|     Sales|         2|
|  4|    Eve| 35|   eve@example.com|     Sales|         1|
|  5|Charlie| 22|charlie@example.com|        HR|         1|
+---+-------+---+------------------+----------+----------+
```

Dataframes – Window functions - AVG()

```
# Define the Window
window_spec_2 = Window.partitionBy("department").orderBy(F.desc("department"))

# Calculate the age average within a department
df.withColumn("age_average", F.avg("age").over(window_spec_2)).orderBy("id").show()
```

```
+---+-------+---+-------------------+----------+
| id|   name|age|       emailaddress|department|
+---+-------+---+-------------------+----------+
|  1|   John| 25|   john@example.com|        IT|
|  2|  Alice| 30|  alice@example.com|        IT|
|  3|    Bob| 28|    bob@example.com|     Sales|
|  4|    Eve| 35|    eve@example.com|     Sales|
|  5|Charlie| 22|charlie@example.com|        HR|
+---+-------+---+-------------------+----------+
```

```
+---+-------+---+-------------------+----------+-----------+
| id|   name|age|       emailaddress|department|age_average|
+---+-------+---+-------------------+----------+-----------+
|  1|   John| 25|   john@example.com|        IT|       27.5|
|  2|  Alice| 30|  alice@example.com|        IT|       27.5|
|  3|    Bob| 28|    bob@example.com|     Sales|       31.5|
|  4|    Eve| 35|    eve@example.com|     Sales|       31.5|
|  5|Charlie| 22|charlie@example.com|        HR|       22.0|
+---+-------+---+-------------------+----------+-----------+
```

Dataframes – Window functions - LAG()

```
# Define the Window
window_spec_2 = Window.partitionBy("department").orderBy(F.desc("department"))

# Calculate the difference between the current row's "age" and the previous row's "age"
df.withColumn("age_diff", F.col("age") - F.lag("age").over(window_spec_2)).orderBy("id").show()
```

```
+---+-------+---+-------------------+----------+
| id|   name|age|       emailaddress|department|
+---+-------+---+-------------------+----------+
|  1|   John| 25|   john@example.com|        IT|
|  2|  Alice| 30|  alice@example.com|        IT|
|  3|    Bob| 28|    bob@example.com|     Sales|
|  4|    Eve| 35|    eve@example.com|     Sales|
|  5|Charlie| 22|charlie@example.com|        HR|
+---+-------+---+-------------------+----------+
```

```
+---+-------+---+-------------------+----------+--------+
| id|   name|age|       emailaddress|department|age_diff|
+---+-------+---+-------------------+----------+--------+
|  1|   John| 25|   john@example.com|        IT|    NULL|
|  2|  Alice| 30|  alice@example.com|        IT|       5|
|  3|    Bob| 28|    bob@example.com|     Sales|    NULL|
|  4|    Eve| 35|    eve@example.com|     Sales|       7|
|  5|Charlie| 22|charlie@example.com|        HR|    NULL|
+---+-------+---+-------------------+----------+--------+
```

Working with Tables

Managed table – delta table format

When a managed table is dropped, its data is deleted from your cloud tenant within 30 days

External table – DELTA, CSV, JSON, AVRO, PARQUET, ORC, TEXT

Dropping an external table does not delete the underlying data; it remains intact.

```
# Save df as table
df.write.mode("overwrite").saveAsTable("table_name")
```

```
# Query the table
result = spark.sql("SELECT name, age FROM table_name WHERE age > 25")
```

Working with Views

Temporary views

Session-scoped, no metastore persistance

```
df.createOrReplaceTempView("view_name")
```

Global Temporary views

Cross-session access, tied to a temporary db called global_temp;

```
df.createOrReplaceGlobalTempView("view_name")
```

SparkSQL vs Dataframes

Ease of Use:

- Spark SQL can be a better option when there are functionalities that might require use of CTE.

Performance:

- Spark SQL can optimize execution plans for SQL queries
- DataFrames offer fine-grained control over optimizations

Expressiveness:

- Spark SQL is limited to SQL operations, while DataFrames allow for custom operations. This modular approach is suitable for unit testing, as you can test individual components of your data transformations.

Schema Evolution: If your data's schema evolves over time, DataFrames allow you to adapt more easily.

Thank you!