PySpark Training – Draft

Overview

PySpark is the Python API for Apache Spark. It enables you to perform real-time, large-scale data processing in a distributed environment using Python. It also provides a PySpark shell for interactively analyzing your data.

PySpark combines Python's learnability and ease of use with the power of Apache Spark to enable processing and analysis of data at any size for everyone familiar with Python.

PySpark supports all of Spark's features such as Spark SQL, DataFrames, Structured Streaming, Machine Learning (MLlib) and Spark Core.

Spark SQL and DataFrames

Spark SQL is Apache Spark's module for working with structured data. It allows you to seamlessly mix SQL queries with Spark programs. With PySpark DataFrames you can efficiently read, write, transform, and analyze data using Python and SQL. Whether you use Python or SQL, the same underlying execution engine is used so you will always leverage the full power of Spark.

- Learn more about Dataframes : DataFrame
- Practice Notebook : Notebook
- Overview of all: Spark SQL APIs

Pandas API on Spark

Pandas API on Spark allows you to scale your pandas workload to any size by running it distributed across multiple nodes. If you are already familiar with pandas and want to leverage Spark for big data, pandas API on Spark makes you immediately productive and lets you migrate your applications without modifying the code. You can have a single codebase that works both with pandas (tests, smaller datasets) and with Spark (production, distributed datasets) and you can switch between the pandas API and the Pandas API on Spark easily and without overhead.

Pandas API on Spark aims to make the transition from pandas to Spark easy but if you are new to Spark or deciding which API to use, we recommend using PySpark (see Spark SQL and DataFrames).

- Quickstart: Pandas API on Spark
- Live Notebook: pandas API on Spark
- Pandas API on Spark Reference

Structured Streaming

Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. You can express your streaming computation the same way you would express a batch computation on static data. The Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive.

- Structured Streaming Programming Guide
- Structured Streaming API Reference

Machine Learning (MLlib)

Built on top of Spark, MLlib is a scalable machine learning library that provides a uniform set of high-level APIs that help users create and tune practical machine learning pipelines.

- Machine Learning Library (MLlib) Programming Guide
- Machine Learning (MLlib) API Reference

Spark Core and RDDs

Spark Core is the underlying general execution engine for the Spark platform that all other functionality is built on top of. It provides RDDs (Resilient Distributed Datasets) and in-memory computing capabilities.

Note that the RDD API is a low-level API which can be difficult to use and you do not get the benefit of Spark's automatic query optimization capabilities. We recommend using DataFrames (see Spark SQL and DataFrames above) instead of RDDs as it allows you to express what you want more easily and lets Spark automatically construct the most efficient query for you.

- Spark Core API Reference

IMPORTANT READ :

RDD vs DF

spark-rdd-vs-dataframe-vs-dataset

Spark Streaming (Legacy)

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.

Note that Spark Streaming is the previous generation of Spark's streaming engine. It is a legacy project and it is no longer being updated. There is a newer and easier to use streaming engine in Spark called Structured Streaming which you should use for your streaming applications and pipelines.

- Spark Streaming Programming Guide (Legacy)
- Spark Streaming API Reference (Legacy)

Getting Started

This page summarizes the basic steps required to setup and get started with PySpark. There are more guides shared with other languages such as [Quick Start] in Programming Guides at [the Spark documentation].

There are live notebooks where you can try PySpark out without any other step:

- Live Notebook: DataFrame
- Live Notebook: Spark Connect
- Live Notebook: pandas API on Spark

The list below is the contents of this quickstart page:

Optional : installation Guidelines

DataFrame

This is a short introduction and quickstart for the PySpark DataFrame API. PySpark DataFrames are lazily evaluated. They are implemented on top of RDDs. When Spark transforms data, it does not immediately compute the transformation but plans how to compute later. When actions such as `collect()` are explicitly called, the computation starts. This notebook shows the basic usages of the DataFrame, geared mainly for new users. You can run the latest version of these examples by yourself in 'Live Notebook: DataFrame' at the quickstart page.

There is also other useful information in Apache Spark documentation site, see the latest version of Spark SQL and DataFrames, RDD Programming Guide, Structured Streaming Programming Guide, Spark Streaming Programming Guide and Machine Learning Library (MLlib) Guide.

PySpark applications start with initializing `SparkSession` which is the entry point of PySpark as below. In case of running it in PySpark shell via pyspark executable, the shell automatically creates the session in the variable spark for users.

[1]:
```
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()
```

DataFrame Creation

A PySpark DataFrame can be created via `pyspark.sql.SparkSession.createDataFrame` typically by passing a list of lists, tuples, dictionaries and `pyspark.sql.Row`s, a pandas DataFrame and an RDD consisting of such a list. `pyspark.sql.SparkSession.createDataFrame` takes the `schema` argument to specify the schema of the DataFrame. When it is omitted, PySpark infers the corresponding schema by taking a sample from the data.

Firstly, you can create a PySpark DataFrame from a list of rows

[2]:
```
from datetime import datetime, date
import pandas as pd
from pyspark.sql import Row

df = spark.createDataFrame([
    Row(a=1, b=2., c='string1', d=date(2000, 1, 1), e=datetime(2000, 1, 1,
12, 0)),
    Row(a=2, b=3., c='string2', d=date(2000, 2, 1), e=datetime(2000, 1, 2,
12, 0)),
    Row(a=4, b=5., c='string3', d=date(2000, 3, 1), e=datetime(2000, 1, 3,
12, 0))
])
```

```
df
```

[2]:

```
DataFrame[a: bigint, b: double, c: string, d: date, e: timestamp]
```

Create a PySpark DataFrame with an explicit schema.

[3]:

```
df = spark.createDataFrame([
    (1, 2., 'string1', date(2000, 1, 1), datetime(2000, 1, 1, 12, 0)),
    (2, 3., 'string2', date(2000, 2, 1), datetime(2000, 1, 2, 12, 0)),
    (3, 4., 'string3', date(2000, 3, 1), datetime(2000, 1, 3, 12, 0))
], schema='a long, b double, c string, d date, e timestamp')
df
```

[3]:

```
DataFrame[a: bigint, b: double, c: string, d: date, e: timestamp]
```

Create a PySpark DataFrame from a pandas DataFrame

[4]:

```
pandas_df = pd.DataFrame({
    'a': [1, 2, 3],
    'b': [2., 3., 4.],
    'c': ['string1', 'string2', 'string3'],
    'd': [date(2000, 1, 1), date(2000, 2, 1), date(2000, 3, 1)],
    'e': [datetime(2000, 1, 1, 12, 0), datetime(2000, 1, 2, 12, 0),
datetime(2000, 1, 3, 12, 0)]
})
df = spark.createDataFrame(pandas_df)
df
```

[4]:

```
DataFrame[a: bigint, b: double, c: string, d: date, e: timestamp]
```

The DataFrames created above all have the same results and schema.

[6]:

```
# All DataFrames above result same.
df.show()
df.printSchema()
```

```
+---+---+-------+----------+-------------------+
|  a|  b|      c|         d|                  e|
+---+---+-------+----------+-------------------+
|  1|2.0|string1|2000-01-01|2000-01-01 12:00:00|
|  2|3.0|string2|2000-02-01|2000-01-02 12:00:00|
|  3|4.0|string3|2000-03-01|2000-01-03 12:00:00|
+---+---+-------+----------+-------------------+

root
 |-- a: long (nullable = true)
 |-- b: double (nullable = true)
 |-- c: string (nullable = true)
 |-- d: date (nullable = true)
 |-- e: timestamp (nullable = true)
```

Viewing Data

The top rows of a DataFrame can be displayed using `DataFrame.show()`.

[7]:

```
df.show(1)
```

```
+---+---+-------+----------+-------------------+
|  a|  b|      c|         d|                  e|
+---+---+-------+----------+-------------------+
|  1|2.0|string1|2000-01-01|2000-01-01 12:00:00|
+---+---+-------+----------+-------------------+
only showing top 1 row
```

Alternatively, you can enable `spark.sql.repl.eagerEval.enabled` configuration for the eager evaluation of PySpark DataFrame in notebooks such as Jupyter. The number of rows to show can be controlled via `spark.sql.repl.eagerEval.maxNumRows` configuration.

[8]:

```
spark.conf.set('spark.sql.repl.eagerEval.enabled', True)
df
```

[8]:

a	b	c	d	e
1	2.0	string1	2000-01-01	2000-01-01 12:00:00
2	3.0	string2	2000-02-01	2000-01-02 12:00:00
3	4.0	string3	2000-03-01	2000-01-03 12:00:00

The rows can also be shown vertically. This is useful when rows are too long to show horizontally.

[9]:

```
df.show(1, vertical=True)
```

```
-RECORD 0------------------
 a   | 1
 b   | 2.0
 c   | string1
 d   | 2000-01-01
 e   | 2000-01-01 12:00:00
only showing top 1 row
```

You can see the DataFrame's schema and column names as follows:

[10]:

```
df.columns
```

[10]:

```
['a', 'b', 'c', 'd', 'e']
```

[11]:

```
df.printSchema()
```

```
root
 |-- a: long (nullable = true)
 |-- b: double (nullable = true)
 |-- c: string (nullable = true)
```

```
 |-- d: date (nullable = true)
 |-- e: timestamp (nullable = true)
```

Show the summary of the DataFrame

[12]:

```
df.select("a", "b", "c").describe().show()
```

```
+-------+---+---+-------+
|summary|  a|  b|      c|
+-------+---+---+-------+
|  count|  3|  3|      3|
|   mean|2.0|3.0|   null|
| stddev|1.0|1.0|   null|
|    min|  1|2.0|string1|
|    max|  3|4.0|string3|
+-------+---+---+-------+
```

`DataFrame.collect()` collects the distributed data to the driver side as the local data in Python. Note that this can throw an out-of-memory error when the dataset is too large to fit in the driver side because it collects all the data from executors to the driver side.

[13]:

```
df.collect()
```

[13]:

```
[Row(a=1, b=2.0, c='string1', d=datetime.date(2000, 1, 1), e=datetime.datetime(2000, 1, 1, 12, 0)),
 Row(a=2, b=3.0, c='string2', d=datetime.date(2000, 2, 1), e=datetime.datetime(2000, 1, 2, 12, 0)),
 Row(a=3, b=4.0, c='string3', d=datetime.date(2000, 3, 1), e=datetime.datetime(2000, 1, 3, 12, 0))]
```

In order to avoid throwing an out-of-memory exception, use `DataFrame.take()` or `DataFrame.tail()`.

[14]:

```
df.take(1)
```

[14]:

```
[Row(a=1, b=2.0, c='string1', d=datetime.date(2000, 1, 1), e=datetime.datetime(2000, 1, 1, 12, 0))]
```

PySpark DataFrame also provides the conversion back to a pandas DataFrame to leverage pandas API. Note that `toPandas` also collects all data into the driver side that can easily cause an out-of-memory-error when the data is too large to fit into the driver side.

[15]:

```
df.toPandas()
```

[15]:

	a	b	c	d	e
0	1	2.0	string1	2000-01-01	2000-01-01 12:00:00
1	2	3.0	string2	2000-02-01	2000-01-02 12:00:00

	a	b	c	d	e
2	3	4.0	string3	2000-03-01	2000-01-03 12:00:00

Selecting and Accessing Data

PySpark DataFrame is lazily evaluated and simply selecting a column does not trigger the computation but it returns a `Column` instance.

[16]:
```
df.a
```
[16]:
```
Column<b'a'>
```
In fact, most of column-wise operations return `Column`s.

[17]:
```
from pyspark.sql import Column
from pyspark.sql.functions import upper

type(df.c) == type(upper(df.c)) == type(df.c.isNull())
```
[17]:
```
True
```
These `Column`s can be used to select the columns from a DataFrame. For example, `DataFrame.select()` takes the `Column` instances that returns another DataFrame.

[18]:
```
df.select(df.c).show()
```
```
+-------+
|      c|
+-------+
|string1|
|string2|
|string3|
+-------+
```

Assign new `Column` instance.

[19]:
```
df.withColumn('upper_c', upper(df.c)).show()
```
```
+---+---+-------+----------+-------------------+-------+
|  a|  b|      c|         d|                  e|upper_c|
+---+---+-------+----------+-------------------+-------+
|  1|2.0|string1|2000-01-01|2000-01-01 12:00:00|STRING1|
|  2|3.0|string2|2000-02-01|2000-01-02 12:00:00|STRING2|
|  3|4.0|string3|2000-03-01|2000-01-03 12:00:00|STRING3|
+---+---+-------+----------+-------------------+-------+
```

To select a subset of rows, use `DataFrame.filter()`.

[20]:
```
df.filter(df.a == 1).show()
```

```
+---+---+-------+----------+-------------------+
|  a|  b|      c|         d|                  e|
+---+---+-------+----------+-------------------+
|  1|2.0|string1|2000-01-01|2000-01-01 12:00:00|
+---+---+-------+----------+-------------------+
```

Applying a Function

PySpark supports various UDFs and APIs to allow users to execute Python native functions. See also the latest Pandas UDFs and Pandas Function APIs. For instance, the example below allows users to directly use the APIs in a pandas Series within Python native function.

[21]:

```
import pandas as pd
from pyspark.sql.functions import pandas_udf

@pandas_udf('long')
def pandas_plus_one(series: pd.Series) -> pd.Series:
    # Simply plus one by using pandas Series.
    return series + 1

df.select(pandas_plus_one(df.a)).show()
```

```
+------------------+
|pandas_plus_one(a)|
+------------------+
|                 2|
|                 3|
|                 4|
+------------------+
```

Another example is `DataFrame.mapInPandas` which allows users directly use the APIs in a pandas DataFrame without any restrictions such as the result length.

[22]:

```
def pandas_filter_func(iterator):
    for pandas_df in iterator:
        yield pandas_df[pandas_df.a == 1]

df.mapInPandas(pandas_filter_func, schema=df.schema).show()
```

```
+---+---+-------+----------+-------------------+
|  a|  b|      c|         d|                  e|
+---+---+-------+----------+-------------------+
|  1|2.0|string1|2000-01-01|2000-01-01 12:00:00|
+---+---+-------+----------+-------------------+
```

Grouping Data

PySpark DataFrame also provides a way of handling grouped data by using the common approach, split-apply-combine strategy. It groups the data by a certain condition applies a function to each group and then combines them back to the DataFrame.

[23]:

```
df = spark.createDataFrame([
```

```
    ['red', 'banana', 1, 10], ['blue', 'banana', 2, 20], ['red', 'carrot', 3,
30],
    ['blue', 'grape', 4, 40], ['red', 'carrot', 5, 50], ['black', 'carrot',
6, 60],
    ['red', 'banana', 7, 70], ['red', 'grape', 8, 80]], schema=['color',
'fruit', 'v1', 'v2'])
df.show()
```

```
+-----+------+---+---+
|color| fruit| v1| v2|
+-----+------+---+---+
|  red|banana|  1| 10|
| blue|banana|  2| 20|
|  red|carrot|  3| 30|
| blue| grape|  4| 40|
|  red|carrot|  5| 50|
|black|carrot|  6| 60|
|  red|banana|  7| 70|
|  red| grape|  8| 80|
+-----+------+---+---+
```

Grouping and then applying the `avg()` function to the resulting groups.

[24]:

```
df.groupby('color').avg().show()
```

```
+-----+-------+-------+
|color|avg(v1)|avg(v2)|
+-----+-------+-------+
|  red|    4.8|   48.0|
|black|    6.0|   60.0|
| blue|    3.0|   30.0|
+-----+-------+-------+
```

You can also apply a Python native function against each group by using pandas API.

[25]:

```
def plus_mean(pandas_df):
    return pandas_df.assign(v1=pandas_df.v1 - pandas_df.v1.mean())

df.groupby('color').applyInPandas(plus_mean, schema=df.schema).show()
```

```
+-----+------+---+---+
|color| fruit| v1| v2|
+-----+------+---+---+
|  red|banana| -3| 10|
|  red|carrot| -1| 30|
|  red|carrot|  0| 50|
|  red|banana|  2| 70|
|  red| grape|  3| 80|
|black|carrot|  0| 60|
| blue|banana| -1| 20|
| blue| grape|  1| 40|
+-----+------+---+---+
```

Co-grouping and applying a function.

```
[26]:
df1 = spark.createDataFrame(
    [(20000101, 1, 1.0), (20000101, 2, 2.0), (20000102, 1, 3.0), (20000102, 
2, 4.0)],
    ('time', 'id', 'v1'))

df2 = spark.createDataFrame(
    [(20000101, 1, 'x'), (20000101, 2, 'y')],
    ('time', 'id', 'v2'))

def merge_ordered(l, r):
    return pd.merge_ordered(l, r)

df1.groupby('id').cogroup(df2.groupby('id')).applyInPandas(
    merge_ordered, schema='time int, id int, v1 double, v2 string').show()
```

```
+--------+---+---+---+
|    time| id| v1| v2|
+--------+---+---+---+
|20000101|  1|1.0|  x|
|20000102|  1|3.0|  x|
|20000101|  2|2.0|  y|
|20000102|  2|4.0|  y|
+--------+---+---+---+
```

Getting Data In/Out

CSV is straightforward and easy to use. Parquet and ORC are efficient and compact file formats to read and write faster.

There are many other data sources available in PySpark such as JDBC, text, binaryFile, Avro, etc. See also the latest Spark SQL, DataFrames and Datasets Guide in Apache Spark documentation.

CSV

```
[27]:
df.write.csv('foo.csv', header=True)
spark.read.csv('foo.csv', header=True).show()
```

```
+-----+------+---+---+
|color| fruit| v1| v2|
+-----+------+---+---+
|  red|banana|  1| 10|
| blue|banana|  2| 20|
|  red|carrot|  3| 30|
| blue| grape|  4| 40|
|  red|carrot|  5| 50|
|black|carrot|  6| 60|
|  red|banana|  7| 70|
|  red| grape|  8| 80|
+-----+------+---+---+
```

Parquet

[28]:
```
df.write.parquet('bar.parquet')
spark.read.parquet('bar.parquet').show()
```
```
+-----+------+---+---+
|color| fruit| v1| v2|
+-----+------+---+---+
|  red|banana|  1| 10|
| blue|banana|  2| 20|
|  red|carrot|  3| 30|
| blue| grape|  4| 40|
|  red|carrot|  5| 50|
|black|carrot|  6| 60|
|  red|banana|  7| 70|
|  red| grape|  8| 80|
+-----+------+---+---+
```

ORC

[29]:
```
df.write.orc('zoo.orc')
spark.read.orc('zoo.orc').show()
```
```
+-----+------+---+---+
|color| fruit| v1| v2|
+-----+------+---+---+
|  red|banana|  1| 10|
| blue|banana|  2| 20|
|  red|carrot|  3| 30|
| blue| grape|  4| 40|
|  red|carrot|  5| 50|
|black|carrot|  6| 60|
|  red|banana|  7| 70|
|  red| grape|  8| 80|
+-----+------+---+---+
```

Working with SQL

DataFrame and Spark SQL share the same execution engine so they can be interchangeably used seamlessly. For example, you can register the DataFrame as a table and run a SQL easily as below:

[30]:
```
df.createOrReplaceTempView("tableA")
spark.sql("SELECT count(*) from tableA").show()
```
```
+--------+
|count(1)|
+--------+
|       8|
+--------+
```

In addition, UDFs can be registered and invoked in SQL out of the box:

[31]:
```
@pandas_udf("integer")
```

```
def add_one(s: pd.Series) -> pd.Series:
    return s + 1

spark.udf.register("add_one", add_one)
spark.sql("SELECT add_one(v1) FROM tableA").show()
```

```
+-----------+
|add_one(v1)|
+-----------+
|          2|
|          3|
|          4|
|          5|
|          6|
|          7|
|          8|
|          9|
+-----------+
```

These SQL expressions can directly be mixed and used as PySpark columns.

[32]:

```
from pyspark.sql.functions import expr

df.selectExpr('add_one(v1)').show()
df.select(expr('count(*)') > 0).show()
```

```
+-----------+
|add_one(v1)|
+-----------+
|          2|
|          3|
|          4|
|          5|
|          6|
|          7|
|          8|
|          9|
+-----------+

+--------------+
|(count(1) > 0)|
+--------------+
|          true|
+--------------+
```

Spark Connect

Spark Connect introduced a decoupled client-server architecture for Spark that allows remote connectivity to Spark clusters using the DataFrame API.

This notebook walks through a simple step-by-step example of how to use Spark Connect to build any type of application that needs to leverage the power of Spark when working with data.

Spark Connect includes both client and server components and we will show you how to set up and use both.

Launch Spark server with Spark Connect

To launch Spark with support for Spark Connect sessions, run the `start-connect-server.sh` script.

[1]:

```
!$HOME/sbin/start-connect-server.sh --packages org.apache.spark:spark-connect_2.12:$SPARK_VERSION
```

Connect to Spark Connect server

Now that the Spark server is running, we can connect to it remotely using Spark Connect. We do this by creating a remote Spark session on the client where our application runs. Before we can do that, we need to make sure to stop the existing regular Spark session because it cannot coexist with the remote Spark Connect session we are about to create.

[2]:

```
from pyspark.sql import SparkSession

SparkSession.builder.master("local[*]").getOrCreate().stop()
```

The command we used above to launch the server configured Spark to run as `localhost:15002`. So now we can create a remote Spark session on the client using the following command.

[3]:

```
spark = SparkSession.builder.remote("sc://localhost:15002").getOrCreate()
```

Create DataFrame

Once the remote Spark session is created successfully, it can be used the same way as a regular Spark session. Therefore, you can create a DataFrame with the following command.

[4]:

```
from datetime import datetime, date
from pyspark.sql import Row

df = spark.createDataFrame([
    Row(a=1, b=2., c='string1', d=date(2000, 1, 1), e=datetime(2000, 1, 1, 12, 0)),
    Row(a=2, b=3., c='string2', d=date(2000, 2, 1), e=datetime(2000, 1, 2, 12, 0)),
```

```
    Row(a=4, b=5., c='string3', d=date(2000, 3, 1), e=datetime(2000, 1, 3,
12, 0))
])
df.show()
+---+---+-------+----------+-------------------+
|  a|  b|      c|         d|                  e|
+---+---+-------+----------+-------------------+
|  1|2.0|string1|2000-01-01|2000-01-01 12:00:00|
|  2|3.0|string2|2000-02-01|2000-01-02 12:00:00|
|  4|5.0|string3|2000-03-01|2000-01-03 12:00:00|
```

OPTIONAL : [Pandas API on Spark]

Build a PySpark Application

Here is an example for how to start a PySpark application. Feel free to skip to the next section, “Testing your PySpark Application,” if you already have an application you’re ready to test.

First, start your Spark Session.

[3]:

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

# Create a SparkSession
spark = SparkSession.builder.appName("Testing PySpark Example").getOrCreate()
```

Next, create a DataFrame.

[5]:

```python
sample_data = [{"name": "John    D.", "age": 30},
  {"name": "Alice   G.", "age": 25},
  {"name": "Bob  T.", "age": 35},
  {"name": "Eve   A.", "age": 28}]

df = spark.createDataFrame(sample_data)
```

Now, let’s define and apply a transformation function to our DataFrame.

[7]:

```python
from pyspark.sql.functions import col, regexp_replace

# Remove additional spaces in name
def remove_extra_spaces(df, column_name):
    # Remove extra spaces from the specified column
    df_transformed = df.withColumn(column_name,
regexp_replace(col(column_name), "\\s+", " "))
```

```
    return df_transformed

transformed_df = remove_extra_spaces(df, "name")

transformed_df.show()
```

```
+---+--------+
|age|    name|
+---+--------+
| 30| John D.|
| 25|Alice G.|
| 35|  Bob T.|
| 28|  Eve A.|
+---+--------+
```

Testing your PySpark Application

Now let's test our PySpark transformation function.

One option is to simply eyeball the resulting DataFrame. However, this can be impractical for large DataFrame or input sizes.

A better way is to write tests. Here are some examples of how we can test our code. The examples below apply for Spark 3.5 and above versions.

Note that these examples are not exhaustive, as there are many other test framework alternatives which you can use instead of `unittest` or `pytest`. The built-in PySpark testing util functions are standalone, meaning they can be compatible with any test framework or CI test pipeline.

Option 1: Using Only PySpark Built-in Test Utility Functions

For simple ad-hoc validation cases, PySpark testing utils like `assertDataFrameEqual` and `assertSchemaEqual` can be used in a standalone context. You could easily test PySpark code in a notebook session. For example, say you want to assert equality between two DataFrames:

[10]:

```
import pyspark.testing
from pyspark.testing.utils import assertDataFrameEqual

# Example 1
df1 = spark.createDataFrame(data=[("1", 1000), ("2", 3000)], schema=["id",
"amount"])
df2 = spark.createDataFrame(data=[("1", 1000), ("2", 3000)], schema=["id",
"amount"])
assertDataFrameEqual(df1, df2)  # pass, DataFrames are identical
```

[11]:

```
# Example 2
df1 = spark.createDataFrame(data=[("1", 0.1), ("2", 3.23)], schema=["id",
"amount"])
df2 = spark.createDataFrame(data=[("1", 0.109), ("2", 3.23)], schema=["id",
"amount"])
```

```
assertDataFrameEqual(df1, df2, rtol=1e-1)  # pass, DataFrames are approx
equal by rtol
```

You can also simply compare two DataFrame schemas:

[13]:

```
from pyspark.testing.utils import assertSchemaEqual
from pyspark.sql.types import StructType, StructField, ArrayType, DoubleType

s1 = StructType([StructField("names", ArrayType(DoubleType(), True), True)])
s2 = StructType([StructField("names", ArrayType(DoubleType(), True), True)])

assertSchemaEqual(s1, s2)  # pass, schemas are identical
```

Option 2: Using Unit Test

For more complex testing scenarios, you may want to use a testing framework.

One of the most popular testing framework options is unit tests. Let's walk through how you can use the built-in Python `unittest` library to write PySpark tests. For more information about the `unittest` library, see here: https://docs.python.org/3/library/unittest.html.

First, you will need a Spark session. You can use the `@classmethod` decorator from the `unittest` package to take care of setting up and tearing down a Spark session.

[15]:

```
import unittest

class PySparkTestCase(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls.spark = SparkSession.builder.appName("Testing PySpark
Example").getOrCreate()


    @classmethod
    def tearDownClass(cls):
        cls.spark.stop()
```

Now let's write a `unittest` class.

[17]:

```
from pyspark.testing.utils import assertDataFrameEqual

class TestTranformation(PySparkTestCase):
    def test_single_space(self):
        sample_data = [{"name": "John    D.", "age": 30},
                       {"name": "Alice   G.", "age": 25},
                       {"name": "Bob  T.", "age": 35},
                       {"name": "Eve   A.", "age": 28}]

        # Create a Spark DataFrame
```

```
        original_df = spark.createDataFrame(sample_data)

        # Apply the transformation function from before
        transformed_df = remove_extra_spaces(original_df, "name")

        expected_data = [{"name": "John D.", "age": 30},
        {"name": "Alice G.", "age": 25},
        {"name": "Bob T.", "age": 35},
        {"name": "Eve A.", "age": 28}]

        expected_df = spark.createDataFrame(expected_data)

        assertDataFrameEqual(transformed_df, expected_df)
```

When run, `unittest` will pick up all functions with a name beginning with "test."

Option 3: Using Pytest

We can also write our tests with `pytest`, which is one of the most popular Python testing frameworks. For more information about `pytest`, see the docs here: https://docs.pytest.org/en/7.1.x/contents.html.

Using a `pytest` fixture allows us to share a spark session across tests, tearing it down when the tests are complete.

[20]:

```
import pytest

@pytest.fixture
def spark_fixture():
    spark = SparkSession.builder.appName("Testing PySpark
Example").getOrCreate()
    yield spark
```

We can then define our tests like this:

[22]:

```
import pytest
from pyspark.testing.utils import assertDataFrameEqual

def test_single_space(spark_fixture):
    sample_data = [{"name": "John    D.", "age": 30},
                   {"name": "Alice   G.", "age": 25},
                   {"name": "Bob  T.", "age": 35},
                   {"name": "Eve   A.", "age": 28}]

    # Create a Spark DataFrame
    original_df = spark.createDataFrame(sample_data)

    # Apply the transformation function from before
    transformed_df = remove_extra_spaces(original_df, "name")
```

```
    expected_data = [{"name": "John D.", "age": 30},
    {"name": "Alice G.", "age": 25},
    {"name": "Bob T.", "age": 35},
    {"name": "Eve A.", "age": 28}]

    expected_df = spark.createDataFrame(expected_data)

    assertDataFrameEqual(transformed_df, expected_df)
```

When you run your test file with the `pytest` command, it will pick up all functions that have their name beginning with "test."

Putting It All Together!

Let's see all the steps together, in a Unit Test example.

[25]:

```
# pkg/etl.py
import unittest

from pyspark.sql import SparkSession
from pyspark.sql.functions import col
from pyspark.sql.functions import regexp_replace
from pyspark.testing.utils import assertDataFrameEqual

# Create a SparkSession
spark = SparkSession.builder.appName("Sample PySpark ETL").getOrCreate()

sample_data = [{"name": "John    D.", "age": 30},
  {"name": "Alice   G.", "age": 25},
  {"name": "Bob  T.", "age": 35},
  {"name": "Eve   A.", "age": 28}]

df = spark.createDataFrame(sample_data)

# Define DataFrame transformation function
def remove_extra_spaces(df, column_name):
    # Remove extra spaces from the specified column using regexp_replace
    df_transformed = df.withColumn(column_name,
regexp_replace(col(column_name), "\\s+", " "))

    return df_transformed
```

[26]:

```
# pkg/test_etl.py
import unittest

from pyspark.sql import SparkSession

# Define unit test base class
class PySparkTestCase(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
```

```python
        cls.spark = SparkSession.builder.appName("Sample PySpark
ETL").getOrCreate()

    @classmethod
    def tearDownClass(cls):
        cls.spark.stop()

# Define unit test
class TestTranformation(PySparkTestCase):
    def test_single_space(self):
        sample_data = [{"name": "John    D.", "age": 30},
                       {"name": "Alice   G.", "age": 25},
                       {"name": "Bob  T.", "age": 35},
                       {"name": "Eve   A.", "age": 28}]

        # Create a Spark DataFrame
        original_df = spark.createDataFrame(sample_data)

        # Apply the transformation function from before
        transformed_df = remove_extra_spaces(original_df, "name")

        expected_data = [{"name": "John D.", "age": 30},
        {"name": "Alice G.", "age": 25},
        {"name": "Bob T.", "age": 35},
        {"name": "Eve A.", "age": 28}]

        expected_df = spark.createDataFrame(expected_data)

        assertDataFrameEqual(transformed_df, expected_df)
```

[27]:

```python
unittest.main(argv=[''], verbosity=0, exit=False)
```

```
Ran 1 test in 1.734s

OK
```

[27]:

```
<unittest.main.TestProgram at 0x174539db0>
```