Testing PySpark

This guide is a reference for writing robust tests for PySpark code.

To view the docs for PySpark test utils, see here.

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]:
```
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]:
```
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]:
```
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()
```

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:

Search the docs ...

Installation
Quickstart: DataFrame
Quickstart: Spark Connect
Quickstart: Pandas API on Spark
Testing PySpark

[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
```

Testing your PySpark
Utility Functions
Option 2: Using Unit Test

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.

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.

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]:
```python
# 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]:
```python
# pkg/test_etl.py
import unittest

from pyspark.sql import SparkSession

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

[27]:
```
<unittest.main.TestProgram at 0x174539db0>
```

Previous
Quickstart: Pandas API on Spark

Next
User Guides