Unit Testing in Databricks

What is it, How to Implement, and Examples

Abhishek Agrawal
Azure Data Engineer

Q1. What is Unit Testing?

Unit testing is the practice of testing individual units of code (usually functions or methods) to ensure they work as expected. In the context of Databricks, unit testing focuses on testing specific parts of your data engineering pipeline, such as PySpark transformations and business logic.

Key Points:

- Ensures that individual functions perform as expected.
- Helps catch errors early in the development cycle.
- Improves code reliability and maintainability

Q2. Why Unit Testing is Important in Databricks?

- **Validation of Functions:** Ensures each transformation or function behaves correctly before integration into larger workflows.

- **Consistency:** Helps guarantee that changes made to the code do not break existing functionality.

- **Automation:** Can be automated to run during continuous integration (CI) processes for seamless testing in Databricks notebooks.

Q3. How to Implement Unit Testing in Databricks?

To implement unit testing in Databricks, follow these steps:

- **Create Test Functions:** Use a testing framework like unittest (Python's built-in testing module) to write test cases.
- **Mock Data:** Use sample datasets or mock data to simulate different conditions in your functions.
- **Run Tests:** Use Databricks notebooks or a CI/CD pipeline to run tests.
- **Test Frameworks:** Databricks supports Python’s unittest, pytest, and nose testing frameworks.

Example of Workflow:

- Create tests in separate files or cells within a Databricks notebook.
- Run tests in a CI/CD pipeline connected to your Databricks workspace.

Example - Unit Testing with unittest

Here’s an example of how to create unit tests for a PySpark transformation using **unittest.**

Test Case Example:

```python
import unittest
from pyspark.sql import SparkSession
from pyspark.sql.functions import lit

# Sample function to be tested
def add_column(df, col_name, value):
    # Add a new column to the DataFrame with a constant value
    return df.withColumn(col_name, lit(value))

class TestDataFrameFunctions(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        # Create a Spark session and a sample DataFrame for testing
        cls.spark = SparkSession.builder.appName("UnitTesting").getOrCreate()
        cls.df = cls.spark.createDataFrame([(1, 'John'), (2, 'Jane')], ["id", "name"])

    def test_add_column(self):
        # Test the add_column function
        df_with_new_col = add_column(self.df, "age", 25)
        self.assertIn("age", df_with_new_col.columns)  # Check if the new column is added
        self.assertEqual(df_with_new_col.select("age").distinct().collect()[0][0], 25)
# Check if the new column has the correct value

    @classmethod
    def tearDownClass(cls):
        # Stop the Spark session after tests are complete
        cls.spark.stop()

if __name__ == '__main__':
    unittest.main()
```

Explanation:

- **setUpClass**: Creates a sample DataFrame before tests are run.
- **test_add_column:** Validates that the new column is added correctly.
- **unittest.main():** Runs the tests.

Running Tests in Databricks

Title: Running Unit Tests in Databricks

Content:

- In Notebooks: You can run unit tests directly in Databricks notebooks by creating test cells and running them manually.

- In CI/CD Pipelines: For automation, integrate your tests into a CI/CD pipeline using Azure DevOps, Jenkins, or GitHub Actions to run tests automatically when code changes are made.

Example:
To run tests in a Databricks notebook, simply:

- Create a separate test cell with your test functions.
- Execute the cell to see results.
- Use the **Databricks CLI** or **Databricks REST API** to automate test runs as part of your workflow.

Example - Unit Testing with pytest

You can also use pytest, a more flexible testing framework, for unit testing.

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

# Sample function to be tested
def add_column(df, col_name, value):
    return df.withColumn(col_name, col(value))

@pytest.fixture(scope="session")
def spark():
    return SparkSession.builder.appName("PySpark Unit Test").getOrCreate()

def test_add_column(spark):
    df = spark.createDataFrame([(1, 'John'), (2, 'Jane')], ["id", "name"])
    df_with_new_col = add_column(df, "age", 25)
    assert "age" in df_with_new_col.columns
```

Explanation:

- **pytest.fixture:** Creates a Spark session for the test.
- **test_add_column:** Validates that the column is added.

Running Tests:
You can run pytest directly from a Databricks notebook using %sh magic commands:

```
%sh
pytest path/to/test_file.py
```

Best Practices for Unit Testing in Databricks

- **Isolate Tests:** Ensure that each test is independent of others and does not affect the global state.
- **Use Mock Data:** For testing, use small, isolated datasets to simulate various scenarios.
- **Automate Tests:** Integrate tests into a CI/CD pipeline for automatic validation.
- **Test Coverage:** Aim for high test coverage to ensure robustness.
- **Error Handling:** Test edge cases (e.g., missing values, empty datasets).

Conclusion

Unit testing in Databricks ensures that individual data transformations and functions work as expected. By incorporating testing frameworks like **unittest or pytest**, you can automate validation, catch errors early, and maintain the integrity of your data engineering pipelines.

Key Takeaways:

- Unit tests validate transformations and business logic.
- Frameworks like **unittest and pytest** are commonly used in Databricks.
- Automate testing in CI/CD pipelines for continuous validation.

Follow for more content like this

Abhishek Agrawal

Azure Data Engineer