pyspark-tdd-template

Release 0.0.1

Aug 17, 2020

Contents:

CHAPTER 1

Introduction

There are a few good Blog about modularising, packaging and structuring the data pipelines for Spark jobs. However, the testing part is often neglected or covered from very top level. Data application testing is different and a pipeline change in the logic is always prone to breaking the logic somewhere else.

This project discusses on a template for data pipeline project using Apache Spark and its Python(*PySpark*) APIs with special focus on data-pipeline testing. But, before we deep dive, We need to touch upon how we can structure our code to use of the approach outlined here. This project covers the following topics:

- Structuring ETL codes into testable modules.
- Setting up configurations and test data(Testbed).
- Boilerplate pytest style testcases for PySpark jobs.
- Packaging and submitting jobs in the cluster.

1.1 Initial Pyspark code

Let's start with a poorly constructed Pyspark pipeline. We will apply the structure in it one step at a time. We will go over the codes so that at the end, all the pieces will make sense how this approach can help us to build a TDD data-pipeline.

Let's consider, we have a pipeline that consume files containing pageviews data and merge it into a final table.

```
"""
Incremental file: input/page_views
        email,pages
        james@example.com,home
        james@example.com,about
        patricia@example.com,home
Final Table::
        +----------------+---------+------------+-----------+
        |email           |page_view|created_date|last_active|
```

(continues on next page)

(continued from previous page)

```
        +-----------------+---------+------------+-----------+
        |james@example.com|10       |2020-01-01  |2020-07-04 |
        |mary@example.com |100      |2020-02-04  |2020-02-04 |
        |john@example.com |1        |2020-03-04  |2020-06-04 |
        +-----------------+---------+------------+-----------+
"""

if __name__ == '__main__':
    spark: SparkSession = SparkSession.builder.enableHiveSupport().getOrCreate()

    page_views = StructType(
        [
            StructField('email', StringType(), True),
            StructField('pages', StringType(), True)
        ]
    )

    inc_df: DataFrame = spark.read.csv(path='/user/stabsumalam/pyspark-tdd-template/
↪input/page_views',
                                       header=True,
                                       schema=page_views)
    inc_df.show()
    prev_df: DataFrame = spark.read.table(tableName='stabsumalam_db.user_pageviews')
    prev_df.show()
    inc_df: DataFrame = (inc_df.groupBy('email').count().
                         select(['email',
                                 col('count').alias('page_view'),
                                 current_date().alias('last_active')
                                 ])
                         )

    df_transformed: DataFrame = (inc_df.join(prev_df, inc_df.email == prev_df.email,
↪'full').
                                 select([coalesce(prev_df.email, inc_df.email).alias(
↪'email'),
                                         (coalesce(prev_df.page_view, lit(0)) +␣
↪coalesce(inc_df.page_view,
                                                                                  ␣
↪   lit(0))).alias(
                                             'page_view'),
                                         coalesce(prev_df.created_date, inc_df.last_
↪active).cast('date').alias(
                                             'created_date'),
                                         coalesce(inc_df.last_active, prev_df.last_
↪active).cast('date').alias(
                                             'last_active')
                                         ])
                                 )

    df_transformed.write.save(path='/user/stabsumalam/pyspark-tdd-template/output/
↪user_pageviews', mode='overwrite')

    spark.stop()
```

The application can be submitted on spark

Let's now look into modularising the application.

1.2 Handling static configurations

If we look closely to the above code, the file paths and other static configurations are tightly coupled with the code. For local execution we want to execute the code in isolation and we will avoid the side effects that can occur from I/O.

Let's decouple the static configurations as a JSON file *configs/config.json*.

```
{
  "page_views": "/user/stabsumalam/pyspark-tdd-template/input/page_views",
  "user_pageviews": "stabsumalam_db.user_pageviews",
  "output_path" : "/user/stabsumalam/pyspark-tdd-template/output/user_pageviews"
}
```

1.3 Handling spark environments

It is not practical to test and debug Spark jobs by sending them to a cluster using spark-submit and examining stack traces for clues on what went wrong. Fortunately we have Pypi Pyspark locally on pipenv

Our pipeline should only focus on the business transformations. Let's take out the auxiliary heavy lifting to a separate module. This module can be reused for all other pipelines that follow a common structure as suggested in this project.

`dependencies.job_submitter` takes care of the following

- Handles the creation of spark environment.
- Passes static job configuration parameters from *configs/config.json* to the job.
- Parses command line arguments to accept dynamic inputs and pass it to the job.
- Dynamically loads the requested job module and runs it.

The job itself has to expose a *run* method.

```
def create_spark_session(job_name: str):
    """Create spark session to run the job

    :param job_name: job name
    :type job_name: str
    :return: spark and logger
    :rtype: Tuple[SparkSession,Log4j]
    """
    spark: SparkSession = SparkSession.builder.appName(job_name).enableHiveSupport().
↪getOrCreate()
    app_id: str = spark.conf.get('spark.app.id')
    log4j = spark._jvm.org.apache.log4j
    message_prefix = '<' + job_name + ' ' + app_id + '>'
    logger = log4j.LogManager.getLogger(message_prefix)
    return spark, logger


def load_config_file(file_name: str) -> Dict:
    """
    Reads the configs/config.json file and parse as a dictionary

    :param file_name: name of the config file
    :return: config dictionary
    """
```

(continues on next page)

(continued from previous page)

```
    try:
        with open(f'{file_name}') as f:
            conf: Dict = json.load(f)
        return conf

    except FileNotFoundError:
        raise FileNotFoundError(f'{file_name} Not found')


def parse_job_args(job_args: str) -> Dict:
    """
    Reads the additional job_args and parse as a dictionary

    :param job_args: extra job_args i.e. k1=v1 k2=v2
    :return: config dictionary
    """
    return {a.split('=')[0]: a.split('=')[1] for a in job_args}


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Job submitter',
                                     usage='--job job_name, --conf-file config_file_
↪name, --job-args k1=v1 k2=v2')
    parser.add_argument('--job', help='job name', dest='job_name', required=True)
    parser.add_argument('--conf-file', help='Config file path', required=False)
    parser.add_argument('--job-args',
                        help='Additional job arguments, It would be made part of␣
↪config dict',
                        required=False,
                        nargs='*')
    args = parser.parse_args()
    job_name = args.job_name
    spark, logger = create_spark_session(job_name)
    config_file = args.conf_file if args.conf_file else 'configs/config.json'
    config_dict: Dict = load_config_file(config_file)
    if args.job_args:
        job_args = parse_job_args(args.job_args)
        config_dict.update(job_args)
    logger.warn(f'calling job {args.job_name}  with {config_dict}')
    job = importlib.import_module(f'jobs.{job_name}')
    job.run(spark=spark, config=config_dict, logger=logger)
    spark.stop()
```

1.4 Modularize the code

Regardless of the complexity of a data-pipeline, this often reduces to defining a series of Extract, Transform and Load (ETL) jobs.

So, 1st step to test the application is to modularize to address the below.

- Segregate the logic into testable modules.
- Separating out the side effects of reading and writing the data.

Below is our pipeline structure:

- `jobs.pipeline.extract()` - deals with reading the input data and return the DataFrames.

- `jobs.pipeline.transform()` - deals with defining the business logic and produce the final DataFrame.
- `jobs.pipeline.load()` - deals with saving the final data into the final destination.
- `jobs.pipeline.run()` - acts as the entry point for the pipeline and collaborate between different parts of the pipeline.
- We have taken out the schema for the DataFrames in *ddl/schema.py*

There is a really good blog by Dr. Alex Ioannides and Eran Campf about structuring ETL projects. Here We have a single module pipeline here with just singleton Extract, Transform and Load methods. Our overall project structure would look like.

```
root/
 |-- configs/
 |   |-- config.json
 |-- dependencies/
 |   |-- job_submitter.py
 |-- ddl/
 |       |-- schema.py
 |-- jobs/
 |   |-- pipeline.py
 |-- tests/
 |   |-- test_data/
 |   |-- | -- employees/
 |   |-- | -- employees_report/
 |   |-- conftest.py
 |   |-- test_bed.json
 |   |-- test_pipeline.py
 |   Makefile
 |   Pipfile
 |   Pipfile.lock
```

Our final code looks like:

```
def extract(spark: SparkSession, config: Dict, logger) -> Tuple[DataFrame, DataFrame]:
    """Read incremental file and historical data and return as DataFrames

    :param spark: Spark session object.
    :type spark: SparkSession
    :param config: job configuration
    :type config: Dict
    :param logger: Py4j Logger
    :rtype logger: Py4j.Logger
    :return: Spark DataFrames.
    :rtype: DataFrame
    """
    inc_df: DataFrame = spark.read.load(path=config['page_views'], format='csv',␣
↪header=True, schema=schema.page_views)
    prev_df: DataFrame = spark.read.table(tableName=config['user_pageviews'])

    return inc_df, prev_df


def transform(inc_df: DataFrame, prev_df: DataFrame, config: Dict, logger) ->␣
↪DataFrame:
    """Transform the data for final loading.
```

(continues on next page)

(continued from previous page)

```
    :param inc_df: Incremental DataFrame.
    :type inc_df: DataFrame
    :param prev_df: Final DataFrame.
    :type prev_df: DataFrame
    :param config: job configuration
    :type config: Dict
    :param logger: Py4j Logger
    :rtype logger: Py4j.Logger
    :return: Transformed DataFrame.
    :rtype: DataFrame
    """

    # calculating the metrics
    inc_df: DataFrame = (inc_df.groupBy('email').count().
                         select(['email',
                                 col('count').alias('page_view'),
                                 lit(config['process_date']).alias('last_active')
                                 ])
                         )

    # merging the data with historical records
    df_transformed: DataFrame = (inc_df.join(prev_df, inc_df.email == prev_df.email,
↪'full').
                                 select([coalesce(prev_df.email, inc_df.email).alias(
↪'email'),
                                         (coalesce(prev_df.page_view, lit(0)) +␣
↪coalesce(inc_df.page_view,
                                                                                ␣
↪   lit(0))).alias('page_view'),
                                         coalesce(prev_df.created_date, inc_df.last_
↪active).cast('date').alias(
                                             'created_date'),
                                         coalesce(inc_df.last_active, prev_df.last_
↪active).cast('date').alias(
                                             'last_active')
                                         ])
                                 )

    return df_transformed


def load(df: DataFrame, config: Dict, logger) -> bool:
    """Write data in final destination

    :param df: DataFrame to save.
    :type df: DataFrame
    :param config: job configuration
    :type config: Dict
    :param logger: Py4j Logger
    :rtype logger: Py4j.Logger
    :return: True
    :rtype: bool
    """
    df.write.save(path=config['output_path'], mode='overwrite')
    return True
```

(continues on next page)

(continued from previous page)

```
def run(spark: SparkSession, config: Dict, logger) -> bool:
    """
    Entry point to the pipeline

    :param spark: SparkSession object
    :type spark: SparkSession
    :param config: job configurations and command lines
    :param logger: Log4j Logger
    :type logger: Log4j.Logger
    :type config: Dict
    :return: True
    :rtype: bool
    """

    logger.warn('pipeline is starting')

    # execute the pipeline
    inc_data, prev_data = extract(spark=spark, config=config, logger=logger)
    transformed_data = transform(inc_df=inc_data, prev_df=prev_data, config=config,␣
↪logger=logger)
    load(df=transformed_data, config=config, logger=logger)

    logger.warn('pipeline is complete')
    return True
```

1.5 Testing setup

Given that we have structured our ETL jobs in testable modules. We can feed it a small slice of 'real-world' production data that has been persisted locally(*tests/test_data*) and check it against expected results. We will be using pytest style tests for our pipeline, under the hood we will also leverage few features (i.e. mock) form unittest

Let's look into the different functionality of our `tests.conftest`

The 1st function of it is to start a SparkSession locally for testing.

```
    def setUp(self):
        """Start Spark, read configs, create the Dataframes and mocks
        """
        self.spark, self.logger = job_submitter.create_spark_session('test_pipeline')
        self.config: Dict = job_submitter.load_config_file(self.config_file)
        self.setup_testbed()
        self.setup_mocks()

    def tearDown(self):
        """Stop Spark
        """
        self.teardown_testbed()
```

We have an utility method `tests.conftest.SparkETLTests.setup_testbed()` that reads the *Testbed* configurations to create the Dataframes in order to test out transform function.

```

    def setup_testbed(self):
        """Creates the Dataframes and tables from the test files as mapped in tests/
↪testbed.json, \
```

(continues on next page)

(continued from previous page)

```
        store those in instance variable named dataframes. \
        It also enriches the test specific job configurations as per the test_bed.json

        tests/test_data/page_views.csv
        email,pages
        james@example.com,home
        james@example.com,about
        patricia@example.com,home

        ddl/schema.py
        page_views = StructType(
        [StructField('email', StringType(), True),
        StructField('pages', StringType(), True)])

        testbed.json
        {
        "data": {
        "page_views": { "file": "tests/test_data/page_views.csv" , "schema": "page_
↪views"}
        }
        }
        """

        try:
            with open('tests/test_bed.json') as f:
                test_bed_conf: Dict = json.load(f)
                data_dict: Dict = test_bed_conf.get('data')
                self.logger.info('loading test data from testbed')
                for df, meta in data_dict.items():
                    dataframe: DataFrame = self.spark.read.load(meta.get('file'),
                                                                schema=getattr(schema,
↪ meta.get('schema'), None),
                                                                **self.file_options)
                    self.dataframes[df] = dataframe
                    if len(df.split('.')) > 1:
                        self.spark.sql(f'create database if not exists {df.split(".
↪")[0]}')
                    dataframe.write.saveAsTable(df, format='hive', mode='overwrite')
                    self.logger.info(f'loaded{df} from {meta.get("file")}')

                conf: Dict = test_bed_conf.get('config')
                self.config.update(conf)
                self.logger.info(f'loaded test config {self.config}')

        except FileNotFoundError:
            self.logger.info('No test data to cook')
```

Let's now have a look into our testing code for the Transform method.

```
def test_pipeline_transform(testbed: SparkETLTests):
    """Test pipeline.transform method using small chunks of input data and expected␣
↪output data\
    to make sure the function is behaving as expected.
    .. seealso:: :class:`SparkETLTests`

    """
    # getting the input dataframes
```

(continues on next page)

(continued from previous page)

```
    inc_df: DataFrame = testbed.dataframes['page_views']
    prev_df: DataFrame = testbed.dataframes['stabsumalam_db.user_pageviews']
    # getting the expected dataframe
    expected_data: DataFrame = testbed.dataframes['exp_user_pageviews']
    # actual data
    transformed_data: DataFrame = pipeline.transform(inc_df=inc_df, prev_df=prev_df,␣
↪config=testbed.config, logger=testbed.logger)
    # comparing the actual and expected data
    testbed.comapare_dataframes(df1=transformed_data, df2=expected_data)
```

As you can see we have made available the a pytest fixture named *testbed*. This object stores the DataFrames and configs for testing in it's member variables. We are passing the DataFrames created out of the test files and matching the output DataFrame using another helper function `tests.conftest.SparkETLTests.comapare_dataframes()`

```
    @classmethod
    def comapare_dataframes(cls, df1: DataFrame, df2: DataFrame, excluded_keys:␣
↪Union[List, str, None] = []) -> bool:
        """
        Compares 2 DataFrames for exact match\
        internally it use pandas.testing.assert_frame_equal


        :param df1: processed data
        :type df1: DataFrame
        :param df2: gold standard expected data
        :type df2: DataFrame
        :return: True
        :param excluded_keys: columns to be excluded from comparision, optional
        :type excluded_keys: Union[List, str, None]
        :rtype: Boolean
        :raises: AssertionError Dataframe mismatch
        """
        excluded_keys = excluded_keys if type(excluded_keys) == list else [excluded_
↪keys]
        df1 = df1.drop(*excluded_keys)
        df2 = df2.drop(*excluded_keys)
        sort_columns = [cols[0] for cols in df1.dtypes]
        df1_sorted = df1.toPandas().sort_values(by=sort_columns, ignore_index=True)
        df2_sorted = df2.toPandas().sort_values(by=sort_columns, ignore_index=True)
        assert_frame_equal(df1_sorted, df2_sorted)
        return True
```

Since the I/O operations are already been separated out we can introspect their calling behaviour using mocks. These mocks are setup in `tests.conftest.SparkETLTests.setup_mocks()`

```
    def setup_mocks(self):
        """Mocking spark and dataframes to introspect the calling behaviour for␣
↪unittesting
        """
        mock_read = create_autospec(DataFrameReader)
        mock_write = create_autospec(DataFrameWriter)
        type(self.mock_spark).read = PropertyMock(return_value=mock_read)
        type(self.mock_df).write = PropertyMock(return_value=mock_write)
```

And the code is tested using below block

```
def test_pipeline_extract(testbed: SparkETLTests):
    """Test pipeline.extract method using the mocked spark session and introspect the␣
↪calling pattern\
    to make sure spark methods were called with intended arguments
    .. seealso:: :class:`SparkETLTests`

    """
    # calling the extract method with mocked spark and test config
    pipeline.extract(spark=testbed.mock_spark, config=testbed.config, logger=testbed.
↪config)
    # introspecting the spark method call
    testbed.mock_spark.read.load.assert_called_once_with(path='/user/stabsumalam/
↪pyspark-tdd-template/input/page_views', format='csv', header=True, schema=schema.
↪page_views)
    testbed.mock_spark.read.table.assert_called_once_with(tableName='stabsumalam_db.
↪user_pageviews')
    testbed.mock_spark.reset_mock()


def test_pipeline_load(testbed: SparkETLTests):
    """Test pipeline.load method using the mocked spark session and introspect the␣
↪calling pattern\
    to make sure spark methods were called with intended arguments
    .. seealso:: :class:`SparkETLTests`

    """
    # calling the extract method with mocked spark and test config
    pipeline.load(df=testbed.mock_df, config=testbed.config, logger=testbed.config)
    # introspecting the spark method call
    testbed.mock_df.write.save.assert_called_once_with(path='/user/stabsumalam/
↪pyspark-tdd-template/output/user_pageviews', mode='overwrite')
```

I have used the generic read and write module of spark for these mocks to work.

Now, let's look into the integration testing, We are now able to test out pipeline by mocking the return value of the I/O operations.

```
def test_run_integration(testbed: SparkETLTests):
    """Test pipeline.run method to make sure the integration is working fine\
    It avoids reading and writing operations by mocking the load and extract method
    .. seealso:: :class:`SparkETLTests`

    """
    with patch('jobs.pipeline.load') as mock_load:
        with patch('jobs.pipeline.extract') as mock_extract:
            mock_load.return_value = True
            mock_extract.return_value = (testbed.dataframes['page_views'], testbed.
↪dataframes['stabsumalam_db.user_pageviews'])
            status = pipeline.run(spark=testbed.spark, config=testbed.config,␣
↪logger=testbed.logger)
            testbed.assertTrue(status)
```

The idea is to use immutable test files for performing the whole validation. Methods can be connected in terms of input and expected output, across different upstream and downstream modules. A proper regression can be leveraged by using this approach of immutable test data and plugged into a CICD deployment.

1.6 Running the example locally

We use pipenv for managing project dependencies and Python environments (i.e. virtual environments). All development and production dependencies are described in the *Pipfile*

```
pip install pipenv
```

Additionally, you can have pyenv to have the desired python enviroment.

To execute the example unit test for this project run

```
pipenv run python -m unittest tests/test_*.py
```

1.7 Deploying into production

The project has a build-in Makefile utility to create zipped dependency and configs and bundle them together

```
make clean
make build
```

Now you can run the pipeline using below command

```
$SPARK_HOME/bin/spark-submit \
--py-files packages.zip \
--files configs/config.json \
dependencies/job_submitter.py --job pipeline --conf-file configs/config.json
```

CHAPTER 2

Jobs package

2.1 jobs.pipeline module

CHAPTER 3

Tests package

3.1 tests.conftest module

3.2 tests.test_pipeline module

CHAPTER 4

Dependencies package

4.1 dependencies.job_submitter module

CHAPTER 5

Indices and tables

- genindex
- modindex
- search