BIG DATA ASSIGNMENT 3

Building an ETL Pipeline with Apache Airflow

In this assignment we are going to build and ETL pipeline using Apache Airflow that processes and prepares data for a ML model. We will use the UCI Machine Learning Repository's "Online Retail" dataset. There one can find transactions occurring between 01/12/2010 and 09/12/2011 for a UK-based and registered non-store online retail.

PIPELINE

Now, we will explain the pipeline followed in this assignment, by briefly commenting each of the functions used throughout this project. The methodology can be summarized in the following steps:

1. Downloading the dataset.
2. Cleaning the data.
3. Transforming the data.
4. Loading the data to a MongoDB collection.
5. Sending an email notifying that the process was executed successfully.

Let us now begin explaining the pipeline.

1. Importing libraries

These libraries will be used in different parts of the pipeline

```
import pandas as pd
from pymongo import MongoClient
from ucimlrepo import fetch_ucirepo
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.operators.email_operator import EmailOperator
from airflow.models import Variable
from datetime import datetime, timedelta
```

2. *Downloading the dataset*.

This is carried out by the function called dataset_downloading, which downloads the data from a URL previously provided, and saves the file in a temporary folder in csv format.

To do that, we use XCom, a feature that allows one to exchange messages or small amounts of data. This provides a simple way for tasks within a workflow to communicate with one another.

```
def dataset_downloading(**kwargs):
    '''
    Function to download the dataset
    '''
    # Fetch dataset from UC Irvine Machine Learning Repository
    online_retail = fetch_ucirepo(id=352)
    df = online_retail.data.original
    #print(df.shape)

    # Save dataframe to a CSV file
    file_path = '/tmp/online_retail.csv'
    df.to_csv(file_path, index=False)

    # Push the file path to XCom
    kwargs['ti'].xcom_push(key='file_path', value=file_path)
```

3. *Cleaning the data*.

In this case, the function clean_the_data is responsible, as one could expect, for cleaning the data. First, reads the data from the previous step and converts the 'InvoiceDate' column to a datetime variable. Then, finds and removes all duplicated and null rows, and finally returns the cleaned data back to the temporary folder.

```
def clean_the_data(**kwargs):
    '''
    Function to clean the data
    '''
    # Pull the file path from XCom
    ti = kwargs['ti']
    file_path = ti.xcom_pull(key='file_path', task_ids='dataset_downloading')

    # Read the dataframe from the CSV file
    df = pd.read_csv(file_path)
    #print(df.shape)

    # Clean the data
    df['InvoiceDate'] = pd.to_datetime(df['InvoiceDate'])
    df.dropna(inplace=True)
    df.drop_duplicates(inplace=True)
    df.reset_index(drop=True, inplace=True)

    # Save the cleaned dataframe back to the CSV file
    df.to_csv(file_path, index=False)

    # Push the cleaned file path back to XCom
    ti.xcom_push(key='file_path', value=file_path)
```

4. *Data transformation*

Here, we add an extra column to the dataframe, called ‘TotalPrice’, that can be defined as the product of the ‘Quantity’ and ‘UnitPrice’ columns. This process is done in the transform_the_data function.

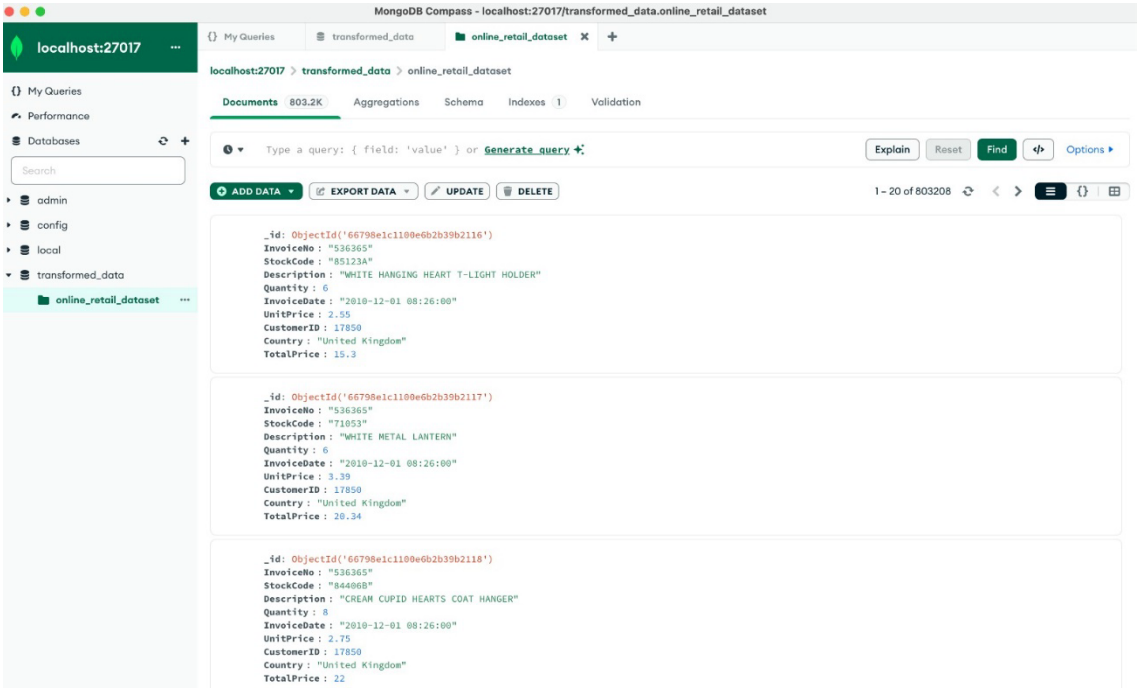
```python
def transform_the_data(**kwargs):
    '''
    Function to transform the data
    '''
    # Pull the file path from XCom
    ti = kwargs['ti']
    file_path = ti.xcom_pull(key='file_path', task_ids='clean_the_data')

    # Read the dataframe from the CSV file
    df = pd.read_csv(file_path)
    #print(df.shape)

    # Add a new column 'TotalPrice'
    df['TotalPrice'] = df['Quantity'] * df['UnitPrice']

    # Save the transformed dataframe back to the CSV file
    file_path = '/tmp/online_retail.csv'
    df.to_csv(file_path, index=False)

    # Push the transformed file path back to XCom
    ti.xcom_push(key='file_path', value=file_path)
```

5. *Loading the data to a MongoDB collection.*

This is carried out by the mongodb_loading function, which, as one can see in the title above, loads the transformed data to the MongoDB collection titled 'retail'. It is worth highlighting that the data is stored as a dictionary.

```python
def mongodb_loading(**kwargs):
    '''
    Function to load data into MongoDB
    '''
    # pull the file path from XCom
    ti = kwargs['ti']
    file_path = ti.xcom_pull(key='file_path', task_ids='transform_the_data')

    # read the dataframe from the csv file
    df = pd.read_csv(file_path)
    #print(df.shape)
    retail_data = df.to_dict(orient='records')
    #print(retail_data)
    client = MongoClient('mongodb://mymongo:27017')

    # specify the database to use
    db = client['transformed_data']

    # specify the collection to use
    collection = db['online_retail_dataset']

    # insert data into the collection
    result = collection.insert_many(retail_data)
    #print(result)
```

In order to confirm that the transformed data was correctly loaded to a NoSQL database, let us share a screenshot of the MondoDB collection.



6. *Sending an email notifying that the process was executed successfully.*

As it is indicated, this task was built to send an email notifying that the pipeline was executed successfully. Below we can see the Airflow variable "emails" that we have to create.

```
# fetch the emails that will recieve the notification
emails = Variable.get("emails")
email_list = [value.strip() for value in emails.split(',')]
```

```
    # Extra Exercise

    summary_email = EmailOperator(
        task_id='send_summary_email',
        to=email_list,
        subject='DAG {{ dag.dag_id }}: Summary',
        html_content="""
        <h3>Summary for DAG {{ dag.dag_id }}</h3>
        <p>The DAG {{ dag.dag_id }} has completed its run.</p>
        <p>Run details:</p>
        <ul>
            <li>Execution Date: {{ ds }}</li>
            <li>Start Date: {{ execution_date }}</li>
            <li>End Date: {{ next_execution_date }}</li>
        </ul>
        """,
        dag=dag
    )
```

Lastly, the screenshot below shows the format of the email.

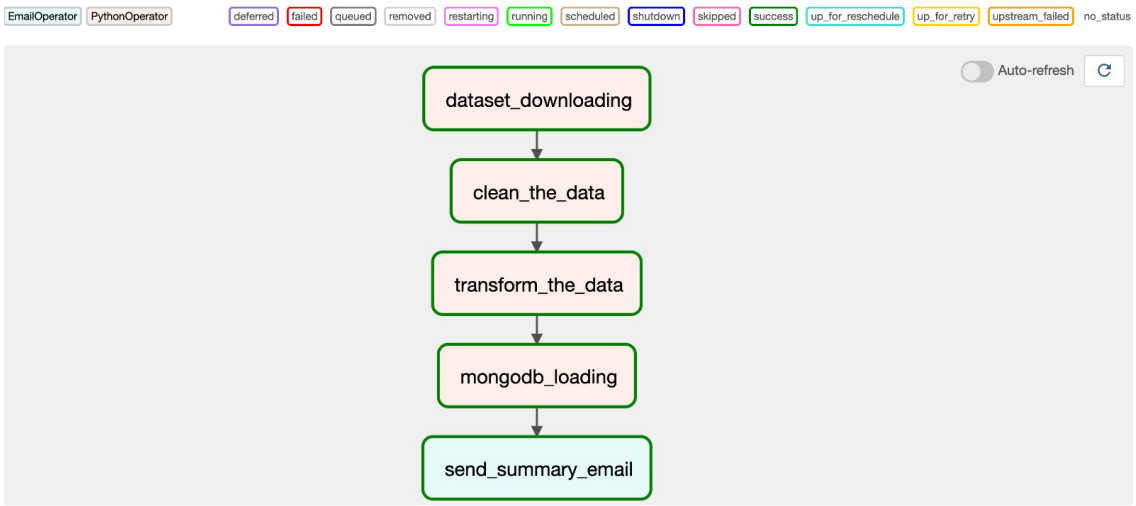
DAG STRUCTURE AND SCHEDULING

Once we have explained the pipeline to follow, we can now define the task flow in this pipeline. Note that the schedule_interval argument sets a timedelta of 1 day, that is, the pipeline executes with a frequency of 1 day.

```
# defaults
default_args = {
    'owner': 'David',
    'depends_on_past': False,
    'start_date': datetime(2024, 6, 21),
    'email': email_list,
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=1)
}

dag = DAG('Assignment3', default_args=default_args, description="Big Data Assignment 3 by David Íñiguez and Jaume Sánchez",
schedule_interval=timedelta(days=1))
```

```
# Task defining using operators
download_task = PythonOperator(
    task_id='dataset_downloading',
    python_callable=dataset_downloading,
    dag=dag)

cleaning_task = PythonOperator(
    task_id='clean_the_data',
    python_callable=clean_the_data,
    dag=dag)

transformation_task = PythonOperator(
    task_id='transform_the_data',
    python_callable=transform_the_data,
    dag=dag)

nosql_task = PythonOperator(
    task_id='mongodb_loading',
    python_callable=mongodb_loading,
    dag=dag)
```

The DAG schema generated by airflow is the following:



Matching:

```
#DAG
download_task >> cleaning_task >> transformation_task >> nosql_task >> summary_email
```

Right below we have the duration of each of the tasks defined.