Data Pipeline - Indicium

[Flashlight] Internship Challenge DE

Felipe Victor Marques de Sousa

Documentation

Contents

1 Problem introduction **1**
1.1 Requirements 3

2 Solution **4**
2.1 Frameworks 4
2.2 Build instructions 5
2.3 pipeline.py 5
2.4 ler_postgres.py 7
2.5 ler_csv.py 10
2.6 conexao.py 10
2.7 saida.py 10
2.8 consulta.py 13
2.9 relacionamentos.py 13

3 Github Repositorie **13**

1 Problem introduction

We are going to provide 2 data sources, a Postgres database and a CSV file. The CSV file represents details of orders from a ecommerce system.

The database provided is a sample database provided by microsoft for education purposes called northwind, the only difference is that the order_table does not exists in this database you are beeing provided with.This order_table is represented by the CSV file we provide.

Schema of the original Northwind Database:

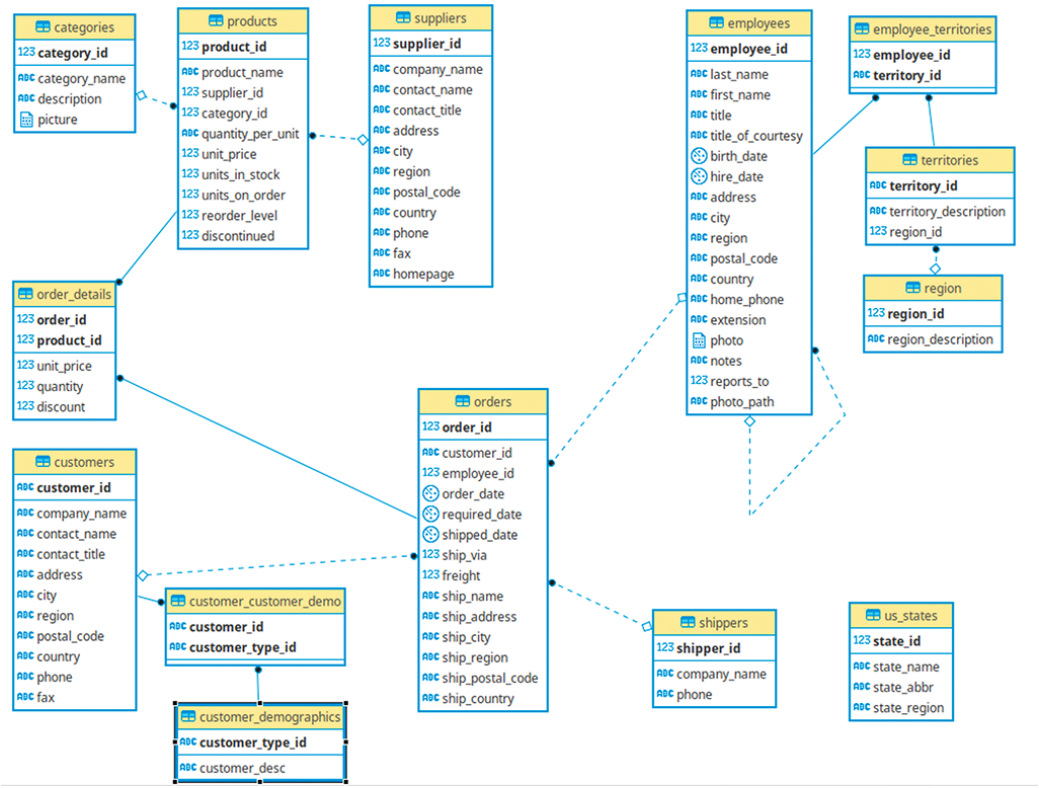

Figure 1: Reproduction of the diagram

Your mission is to build a pipeline that extracts the data everyday from both sources and write the data first to local disk, and second to a database of your choice. For this challenge, the CSV file and the database will be static, but in any real world project, both data sources would be changing

constantly.

Its important that all writing steps are isolated from each other, you shoud be able to run any step without executing the others.

For the first step, where you write data to local disk, you should write one file for each table and one file for the input CSV file. This pipeline will run everyday, so there should be a separation in the file paths you will create for each source(CSV or Postgres), table and execution day combination, e.g.:

```
/data/postgres/{table}/2021-01-01/file.format
/data/postgres/{table}/2021-01-02/file.format
/data/csv/2021-01-02/file.format
```

Listing 1: Out format

you are free to chose the naming and the format of the file you are going to save.

At step 2, you should load the data from the local filesystem to the final database that you chosed.

The final goal is to be able to run a query that shows the orders and its details. The Orders are placed in a table called orders at the postgres Northwind database. The details are placed at the csv file provided, and each line has an order_id field pointing the orders table.

How you are going to build this query will heavily depend on which database you choose and how you will load the data this database.

The pipeline will look something like this:

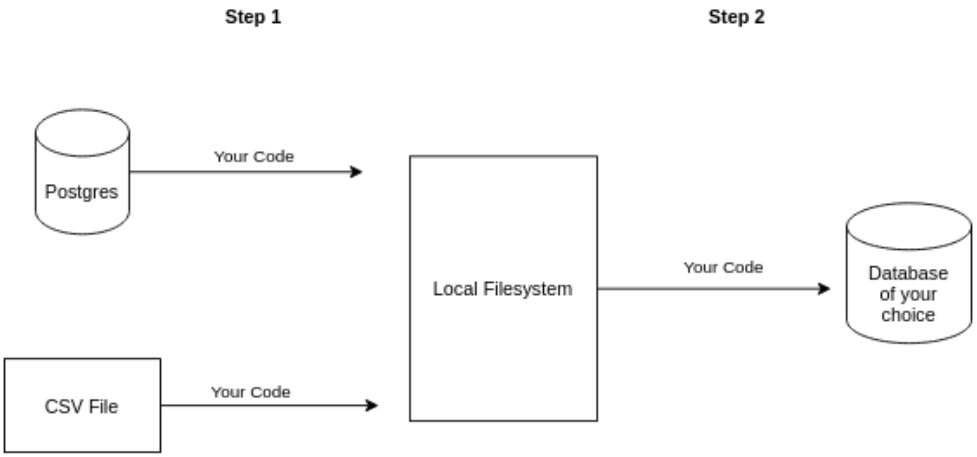

Figure 2: Reproduction of the diagram

1.1 Requirements

- All tasks should be idempotent, you should be able the whole pipeline for a day and the result should be always the same
- Step 2 depends on both tasks of step 1, so you should not be able to run step 2 for a day if the tasks from step 1 did not succeed
- You should extract all the tables from the source database, it does not matter that you will not use most of them for the final step.
- You should be able to tell where the pipeline failed clearly, so you know from which step you should rerun the pipeline
- You have to provide clear instructions on how to run the whole pipeline. The easier the better.
- You have to provide a csv or json file with the result of the final query at the final database.
- You dont have to actually schedule the pipeline, but you should assume that it will run for different days.
- Your pipeline should be prepared to run for past days, meaning you should be able to pass an argument to the pipeline with a day from the past, and it should reprocess the data for that day. Since the data

for this challenge is static, the only difference for each day of execution will be the output paths.

2 Solution

2.1 Frameworks

The problem was solved using the Python language, which makes it easier to manipulate data. The solution brings together a number of frameworks, such as Docker, which brought the environment from the challenge with the PostgreSQL database - and the Visual Studio Code, IDE where the solution was developed.

Some libraries were also used;

- asyncio - The asyncio library provides tools that ensure the waiting execution of specific snippets of code. With this library, it was possible to meet the requirement that each step of the pipeline be successfully executed before the other.
- datetime - Already the datetime library, allows to store information of the current date and also of the previous date, indispensable requirements of the application.
- sqlite3 - The sqlite3 library allows you to work with a SQLite database.
- json - Generating JSON files and writing new files has become much easier using the json library.
- os - The "os" library allows you to handle folders and files within the operating system.
- pandas - The "pandas" library allows you to create dataframes. This made it a little easier to transition files between CSV and distribution to folders in the operating system.
- psycopg2 - Psycopg is the most popular PostgreSQL database adapter for the Python programming language. It was required to handle the initial database.

2.2 Build instructions

In first time, you must install the libraries listed above. Below is a list of required commands if you are not using Docker;

```
pip install asyncio (maybe not necessary install because has in your python version)
pip install db-sqlite3 (maybe not necessary install because has in your python version)
pip install os_sys (maybe not necessary install because has in your python version)

pip install psycopg2
```

Listing 2: libraries install

In secod time, after install libraries, your need open the **pipeline.py** and run this code. Its a principal class on this.

- Download the data files and play in a folder within the repository, called "data". These files are the "northwind.sql" and the "order_details.csv"
- Use "docker-compose.yml" to mount the Postgre database before compiling.
- Type "python pipeline.py" to compile via terminal.

```
python pipeline.py
```

Listing 3: run the code

2.3 pipeline.py

The pipeline.py is a skeleton of solution. It contains all the main function calls for the operation of the pipeline. In addition, the rule of execution of the problem is also embedded.

Step 1: The first step of the code reads the Postgre database and stores the files in a JSON file. The choice of JSON file type is why it makes it easier to treat the data later.

Because JSON organizes data in a simplistic way, reading it later becomes a less labor-intensive task, and for this particular problem, it would not be

inappropriate to use it.

In addition, the first step also reads the CSV file, making it a JSON extension, just as it was done with the Postgre database.

```
    PG = ler_postgres.PG(data_atual, conn)
    PG_anterior = ler_postgres.PG(data_anterior, conn)
    nomestabelas = await PG.read_PG()
    await PG_anterior.read_PG()

    CSV = ler_csv.CSV(data_atual)
    CSV_anterior = ler_csv.CSV(data_anterior)
    await CSV.read_csv()
    await CSV_anterior.read_csv()
```

Listing 4: Functions call on step 1

Step 2 - On the step 2, has a union with CSV Json and Postgre Json. This union generates a single database in SQLite. SQLite was chosen because it's a simple database system to work with and doesn't require external installations to work well in python. The use of it meets the requested problem and did not demonstrate operational complications or inconsistencies when dealing with the data.

```
    uniao = saida.Saida(data_atual, nomestabelas)
    uniao.banco()

    relacionamentos.relacionamentos('Wellington Importadora')
```

Listing 5: Functions call on step 2

2.4 ler_postgres.py

In the PG class within ler_postgres.py, there is the read_PG function. This function reads the data from the Postgre database and transforms it into JSON files.

In the foreground, a SQL connection is made. Next, the name of all the tables present in the database is stored. The name of the tables also takes the name of the .json files.

```
    cursor = self.conn.cursor()
        consulta_sql = "SELECT table_name FROM information_
            schema.tables WHERE table_schema = 'public';"

        if consulta_sql == None:
            print('Consulta SQL falhou! Revise os dados e tente
                novamente!')
        else:
            cursor.execute(consulta_sql)
            nomes_tabelas = [registro[0] for registro in cursor.
                fetchall()]
```

Listing 6: SQL connection and tables names

In the foreground, a SQL connection is made. Next, the name of all the tables present in the database is stored. The name of the tables also takes the name of the .json files.

```
nomes_colunas, tipos_colunas = data_consulta.get_colunas()
if nomes_colunas == [] or tipos_colunas == []:
    print('A consulta de nomes ou tipos de colunas falhou!
        Revise os dados e tente novamente!')
    return
```

Listing 7: SQL connection and tables names

A loop runs through all the corresponding tables and stores the data in the "data" variable.

Next, the path of the folder where this data will be saved is stored, using "table" and the corresponding data for the name and body of the json file.

A conversion of the data of the "data", "memoryview" and "Timestamp" types to strings was required, since the JSON file would not receive this data as it came from the originating database.

```
    dados_serializaveis = []
    for linha in dados:
        linha_serializavel = [str(col) if isinstance(col,
            datetime.date) else col for col in linha]
        linha_serializavel = [list(col) if isinstance(col,
            memoryview) else col for col in linha_serializavel]
        linha_serializavel = [list(col) if isinstance(col, pd._
            libs.tslibs.timestamps.Timestamp) else col for col in
             linha_serializavel]
        dados_serializaveis.append(linha_serializavel)
```

Listing 8: Data type conversion

Next, the name of the columns of each table was captured. This information was useful to create another .json file that has this information, so that it also facilitates the operation of the data when passing it to the SQLite database.

```
# Obter os nomes das colunas
    nomes_colunas, tipos_colunas = data_consulta.get_colunas()
    if nomes_colunas == [] or tipos_colunas == []:
        print('A consulta de nomes ou tipos de colunas falhou!
            Revise os dados e tente novamente!')
        return
```

Listing 9: Columns names

Then there are a series of commands that save the JSON files. There are three; one for the table data, another for storing the column name, and a last file that deals with the types of the columns present in the table. For example, if the type is an integer, a string, or an image, among others.

After this definition, a dictionary was also created to better handle this division between .json files.

```
relacionamentos = {
    "tabela": tabela,
    "colunas": nomes_colunas,
    "registros": dados_serializaveis
}
```

Listing 10: Separation dictionary

Then, after new commands to finish generating the .json files, the names of the tables are returned, which will be used when joining the data in step 2.

2.5 ler_csv.py

The "read_csv" class begins by enabling the construction of folders to save the .json files that will receive the data from the CSV file.

The difference here is due to the fact that the data was first sent to a dataframe.

They could be taken directly to JSON. However, by convention, so that they can be better handled, it was decided that they would first be allocated in a dataframe.

In this data set, because they are static, no treatment and no learning application were necessary, but if necessary, the path would already be facilitated with the intermediate of the dataframe.

```
1 df_data = pd.read_csv('data/order_details.csv')
```

Listing 11: Dataframe

By the end of the class, the steps are similar to the class read_PG. The focus is on saving the data to a .json file. For this, a dictionary was also used to help separate the data types and column names.

2.6 conexao.py

The "conexao.py" class is a very simple. It is here to facilitate the connection call to the Postgre database. It has the credentials made available and performs the connection.

2.7 saida.py

The "bank" class present in the saida.py is where the entire structure of step 2 is. This is where all the data from the .json files is read and brought into a single database, built using SQLite.

A loop runs through all the tables present in the Postgre database. This database was not accessed again, since the data is already saved in "table". An initializer was used in the class to retrieve this data.

In this for loop are also the calls to the .json files that were saved.

```
for tabela in self.tabelas:
    caminho_arquivo_json = f'data/postgres/{tabela}/{str(self.
        data)}/{tabela}.json'
    caminho_arquivo_colunas = f'data/postgres/{tabela}/{str(self
        .data)}/{tabela}_colunas.json'
    caminho_arquivo_tipos = f'data/postgres/{tabela}/{str(self.
        data)}/{tabela}_tipos.json'

    with open(caminho_arquivo_json, 'r') as f:
        dados = json.load(f)
        if dados:
            tabela_nome = dados["tabela"]
            registros = dados["registros"]

            with open(caminho_arquivo_colunas, 'r') as colunas_
                file:
                colunas = json.load(colunas_file)

            with open(caminho_arquivo_tipos, 'r') as tipos_file:
                tipos = json.load(tipos_file)

            self.criar_tabela(cursor, tabela_nome, colunas,
                tipos)
            self.inserir_dados(cursor, tabela_nome, registros,
                colunas)
```

Listing 12: Opening .json files and calling function to create SQLite table and insert into it

In the create table function, which is the first call after opening the .json files, this is where SQLite actually comes into play.

The function is short and has only what is necessary to create the table if it does not exist. It receives the data, table names, column names, and column type to create the database.

The "inserir_dados" function is a bit more complex. In addition to the table names, data, and column names, this function also receives the records so that the data can be inserted into the SQLite table in an organized way.

```
def inserir_dados(self, cursor, tabela, registros, colunas):
    for registro in registros:
        valores = [str(value) if value is not None else '' for
            value in registro]
        placeholders = ', '.join(['?' for _ in valores])

        consulta_verificar = f'SELECT COUNT(*) FROM {tabela}
            WHERE {colunas[0]} = ?'
        cursor.execute(consulta_verificar, (valores[0],))
        if cursor.fetchone()[0] == 0:
            consulta_insert = f'INSERT INTO {tabela} VALUES ({
                placeholders})'
            cursor.execute(consulta_insert, valores)
```

Listing 13: Opening .json files and calling function to create SQLite table and insert into it

A query is also made to avoid saving duplicate data. That is; Since the pipeline runs every day, then, one cannot record what has already been recorded the day before.

There is also an almost identical but separate data entry function to enter the data from the "order_details" table. There is a slight difference when it comes to inserting, because the list type is different.

```
def inserir_dados_orderdt(self, cursor, tabela, registros,
    colunas):
    for registro in registros:
        valores = [str(value) if value is not None else '' for
            value in registro.values()]
        placeholders = ', '.join(['?' for _ in valores])

        consulta_verificar = f'SELECT COUNT(*) FROM {tabela}
            WHERE {colunas[0]} = ?'
        cursor.execute(consulta_verificar, (valores[0],))
        if cursor.fetchone()[0] == 0:
            consulta_insert = f'INSERT INTO {tabela} VALUES ({
                placeholders})'
            cursor.execute(consulta_insert, valores)
```

Listing 14: inserir_dados_orderdt function

A function has also been developed to set the column types according to the types read from the .json file. An analysis was done to identify the most common types of the database and with that, set the type in the SQLite database.

```
def tipo_sqlite(self, tipo):
    if "integer" in tipo:
        return 'INTEGER'
    elif "character" in tipo or "text" in tipo:
        return 'TEXT'
    else:
        return 'BLOB'
```

Listing 15: inserir_dados_orderdt function

2.8 consulta.py

The "Consulta" class within consulta.py, is specific to perform data queries to generate the lists of "nomes_colunas", and "tipos_colunas".

2.9 relacionamentos.py

Finally, the relacionamentos.py class is a class for generating the relationships between tables. Because the data is static, I only listed the order_details, orders, and products tables. The relationship between these tables and the correct obtaining of the output data proves the efficiency and proper functioning of the pipeline.

In addition, this class also prints the result of the query to the terminal, as well as writing a . JSON with the result.

3 Github Repositorie

Link to the repository of the developed code. (https://github.com/phvictorr/Pipeline-dados)