ON PREMISE SOLUTION

Data Pipeline Task Documentation

Contents

1. Introduction

This task was about working with data from CSV files. First, I loaded the files into Python. Then, I checked the data for problems, cleaned it, added new information, combined the data, and saved it into a SQL Server database. Below are the steps I followed.

2. Setting Up the Environment

I used Python and needed some libraries:

- **os** to work with files and folders.
- **pandas** to work with data in tables.
- **sqlalchemy** to save data into SQL Server.

Then I saved the datasets in folder within the jupyter notebook named LufthansaTask. I loaded all the CSV files from a folder. Each file became a table called a DataFrame. I stored all these tables in a dictionary so I could use them easily.

```python
import os
import pandas as pd

# folder where CSV files are located
folder_path = 'LufthansaTask'

# List all files in the folder
csv_files = [f for f in os.listdir(folder_path) if f.endswith('.csv')]

# Initialize a dictionary to store DataFrames, where each file is a key
data_dict = {}

# Loop through each CSV file
for file in csv_files:
    file_path = os.path.join(folder_path, file)

    # Read the CSV file into a DataFrame
    data = pd.read_csv(file_path)

    # Store the DataFrame in the dictionary with the file name as the key
    data_dict[file] = data
    print(file_path)
```

```
LufthansaTask\olist_customers_dataset.csv
LufthansaTask\olist_geolocation_dataset.csv
LufthansaTask\olist_orders_dataset.csv
LufthansaTask\olist_order_items_dataset.csv
LufthansaTask\olist_order_payments_dataset.csv
LufthansaTask\olist_order_reviews_dataset.csv
LufthansaTask\olist_products_dataset.csv
LufthansaTask\olist_sellers_dataset.csv
LufthansaTask\product_category_name_translation.csv
```

3. Check Data Quality

I created a function not to be repetitive to check the data quality for each dataset. I checked types of data in the columns if there were some columns to change the data type. I also checked for missing values in the tables, duplicated rows and some statistics like average or counts.

```
# function to check data quality
def check_data_quality(data_dict):
    for file_name, df in data_dict.items():
        print(f"\n========== Analyzing {file_name} ==========\n")

        # display data types
        print("Data Types:")
        print(df.dtypes)

        # check for missing values
        print("\nMissing Values:")
        print(df.isnull().sum())

        # check for duplicate rows
        print("\nDuplicate Rows:", df.duplicated().sum())

        # display general statistics
        print("\nSummary Statistics:")
        print(df.describe(include='all'))  # Include categorical data as well

        print("\n================================================================

# Run the data quality check
check_data_quality(data_dict)
```

4. Data processing

4.1 Geolocation Data

I cleaned the geolocation data. I grouped rows that had the same zip code. For each group, I took the middle value for latitude and longitude. I also chose the most common city and state.

```python
# Load the geolocation dataset
geolocation = data_dict['olist_geolocation_dataset.csv']

# aggregate duplicate zip codes by taking median lat/lng and most frequent city/state
geolocation = geolocation.groupby('geolocation_zip_code_prefix').agg({
    'geolocation_lat': 'median',
    'geolocation_lng': 'median',
    'geolocation_city': lambda x: x.mode()[0],   # simpler and clearer
    'geolocation_state': lambda x: x.mode()[0]
}).reset_index()
```

4.2 Orders Data

I removed rows with missing values. The number of missing values is small, and they don't represent significant portions of the data. Then, I converted time columns into a special format called datetime. This format makes it easier to work with dates and times.

```python
# drop NA values from orders dataset
orders = 'olist_orders_dataset.csv'
orders = data_dict[orders].dropna()

print(orders.shape)
```

```
(96461, 8)
```

```python
# list of columns to convert to datetime
timestamp_columns = [
    'order_purchase_timestamp',
    'order_approved_at',
    'order_delivered_carrier_date',
    'order_delivered_customer_date',
    'order_estimated_delivery_date'
]

# convert the timestamp columns to datetime format
for col in timestamp_columns:
    orders[col] = pd.to_datetime(orders[col])
```

4.3 Order Items

In this table, I converted the shipping_limit_date column into the datetime format.

```python
order_items = data_dict['olist_order_items_dataset.csv']

# convert 'shipping_limit_date' to datetime
order_items['shipping_limit_date'] = pd.to_datetime(order_items['shipping_limit_date'])
```

4.4 Order Reviews Data

I converted the review_answer_timestamp column to datetime format. I also filled empty comments with the text "No Comment."

```
order_reviews = data_dict['olist_order_reviews_dataset.csv']

# convert 'shipping_limit_date' to datetime
order_reviews['review_answer_timestamp'] = pd.to_datetime(order_reviews['review_answer_timestamp'])
```

```
# fill missing values in review_comment_title and review_comment_message with a comment
order_reviews['review_comment_title'] = order_reviews['review_comment_title'].fillna('No Comment')
order_reviews['review_comment_message'] = order_reviews['review_comment_message'].fillna('No Comment')
```

4.5 Products Data

I removed all rows that had missing values. The number of missing values is small, and they don't represent significant portions of the data.

```
# drop NA values from products dataset
products = 'olist_products_dataset.csv'
products = data_dict[products].dropna()
```

5. Calculated Columns and Merges

Before calculating any new columns, I made sure to merge the customer and seller data with geolocation information. This ensured that both datasets had accurate location details, including cities and states.

```
customers = data_dict['olist_customers_dataset.csv']
sellers = data_dict['olist_sellers_dataset.csv']

# merge customers with geolocation
customers = customers.merge(geolocation, left_on="customer_zip_code_prefix", right_on="geolocation_zi
).drop(columns=["geolocation_zip_code_prefix"])

# merge sellers with geolocation
sellers = sellers.merge(geolocation, left_on="seller_zip_code_prefix", right_on="geolocation_zip_code_
).drop(columns=["geolocation_zip_code_prefix"])
```

5.1 Total Price

I calculated the total price by adding the product price and the freight value. This gave a complete cost for each order item.

```
# Total Price: Sum of product price and freight value.
order_items["total_price"] = order_items["price"] + order_items["freight_value"]
```

5.2 Delivery Time

First, I merged the order_items dataset with the orders dataset to have access to delivery and purchase dates. I calculated the number of days between the two dates.

```
# Delivery Time: Difference between the delivery date and the order purchase date.
# merge order_items and orders
order_items_orders = order_items.merge(orders, on="order_id", how="left")

order_items_orders["delivery_time"] = (order_items_orders["order_delivered_customer_date"] - order_ite
```

5.3 Payment Count

I grouped the order_payments dataset by order_id and summed the payment_installments. Then, I merged the payment data with the order_items_orders dataset.

```
# Payment Count: Sum of payment installments for each order.
payment_count = order_payments.groupby("order_id", as_index=False).agg(payment_count=("payment_instal

# 2. Merge order_items_orders with payment_count
order_items_orders_payments = order_items_orders.merge(payment_count, on="order_id", how="left")
```

5.4 Profit Margin

To estimate profit, I calculated the difference between the product price and the freight value. This provided a rough measure of profitability for each product.

```
# create the 'profit_margin' column
order_items_orders_payments["profit_margin"] = order_items_orders_payments["price"] - order_items_orders_payments["freight_value"]
```

6. Window Functions Over Partitions and Merges

Window functions help calculate values that depend on groups of data, like totals or averages. I used these to add columns for total sales and average delivery time, grouped by specific categories.

6.1 Total Sales per Customer

I calculated a running total of the product prices for each customer. This helps track how much each customer has spent over their orders.

First, I merged the order_items_orders_payments dataset with the customers dataset using the customer_id column. This added customer details to the data. Then, I used groupby with customer_id to group orders by customers. Finally, I used cumsum() to calculate a running total of the total_price column for each customer.

```
# merge order_items_orders_payments with customers
order_items_orders_payments_customers = order_items_orders_payments.merge(customers, on="customer_id", how="left")
order_items_orders_payments_customers["total_sales_per_customer"] = order_items_orders_payments_customers.groupby("customer_id")["total_price"].cumsum()
```

6.2 Average Delivery Time per Product Category

I calculated a rolling average of delivery time for each product category. This shows the average time it takes to deliver products within each category.

I merged the order_items_orders_payments_customers dataset with the products dataset using the product_id column. This added product details to the data. Then, I used groupby with product_category_name to group delivery times by product categories. Finally, I used rolling() to calculate a rolling average of the delivery times for a window of 5 rows.

```
# Merge order_items_orders_payments_customers with products on 'product_id'
order_items_orders_payments_customers_products = order_items_orders_payments_customers.merge(products, on="product_id", how="left")
order_items_orders_payments_customers_products["avg_delivery_time_per_product_category"] = order_items_orders_payments_customers_products.groupby("prod
```

7. Saving Processed Data to SQL Server

To save the data into SQL Server, I first created several new tables (dataframes).

Fact Table This table includes calculated columns like Total Price, Delivery Time, Payment Count, Profit Margin, and more. It brings together all the processed data about orders, items, and customers.

```
# Fact Table: Includes calculated columns like Total Price, Delivery Time, etc.
fact_table = order_items_orders_payments_customers_products[[
    "order_id", "order_item_id", "product_id", "seller_id", "customer_id", "total_price",
    "delivery_time", "payment_count", "profit_margin", "total_sales_per_customer",
    "avg_delivery_time_per_product_category"
]]
```

Dimension Tables Dimension tables hold specific details about customers, products, sellers, and dates. Here's how I created them:

- **Customer Dimension Table:** Contains customer details like zip code, city, and state.
- **Product Dimension Table:** Contains product details like name, description length, and size measurements.
- **Seller Dimension Table:** Contains seller details along with their geolocation.
- **Date Dimension Table:** Contains important dates like order purchase and delivery dates.

```python
# Customer Dimension Table
customer_dimension = order_items_orders_payments_customers_products[[
    "customer_id", "customer_zip_code_prefix", "customer_city", "customer_state"
]].drop_duplicates()

# Product Dimension Table
product_dimension = order_items_orders_payments_customers_products[[
    "product_id", "product_category_name", "product_name_lenght",
    "product_description_lenght", "product_photos_qty", "product_weight_g",
    "product_length_cm", "product_height_cm", "product_width_cm"
]].drop_duplicates()

# Seller Dimension Table
seller_dimension = order_items_orders_payments_sellers_products[[
    "seller_id", "seller_zip_code_prefix", "seller_city", "seller_state"
]].drop_duplicates()

# Date Dimension Table
date_dimension = order_items_orders_payments_customers_products[[
    "order_id", "order_purchase_timestamp", "order_delivered_customer_date"
]].drop_duplicates()
```

7.1 Connecting to SQL Server

To save the tables, I first connected Python to SQL Server. I used SQLAlchemy and defined a connection string. This string includes the server name, database name, and authentication method.

```python
# connection string using Windows Authentication
connection_string = (
    r'DRIVER={ODBC Driver 17 for SQL Server};'
    r'SERVER=LAPTOP-9MIQ9SU9\SQLEXPRESS;'  # my server name
    r'DATABASE=LufthansaTask;'  # the database
    r'Trusted_Connection=yes;'
)

# create connection URL
connection_url = (
    'mssql+pyodbc:///?odbc_connect='
    + urllib.parse.quote_plus(connection_string)
)

# create the SQLAlchemy engine
engine = create_engine(connection_url, isolation_level="AUTOCOMMIT")
```

7.2 Saving Tables

After connecting to the database, I saved each table using the to_sql function.

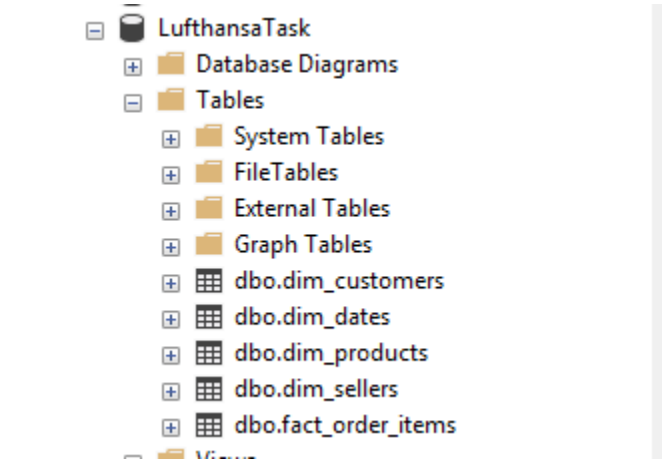
```python
# Save the Fact Table
fact_table.to_sql("fact_order_items", con=engine, if_exists="replace", index=False)

# Save the Customer Dimension Table
customer_dimension.to_sql("dim_customers", con=engine, if_exists="replace", index=False)

# Save the Product Dimension Table
product_dimension.to_sql("dim_products", con=engine, if_exists="replace", index=False)

# Save the Seller Dimension Table
seller_dimension.to_sql("dim_sellers", con=engine, if_exists="replace", index=False)

# Save the Date Dimension Table
date_dimension.to_sql("dim_dates", con=engine, if_exists="replace", index=False)
```



8. SQL Server Validation and Reporting

After uploading the processed data into SQL Server, I wrote queries to check and validate the data. These queries also provide meaningful insights, such as total sales, average delivery time, and order distribution by states.

8.1 Query 1: Total Sales per Product Category

This query calculates the total sales for each product category by summing up the total_price column. The data comes from the fact table joined with the product dimension table.

```sql
---------- Query total sales per pro
SELECT
    p.product_category_name,
    SUM(f.total_price) AS total_sales
FROM
    fact_order_items f
INNER JOIN
    dim_products p
ON
    f.product_id = p.product_id
GROUP BY
    p.product_category_name
ORDER BY
    total_sales DESC;
```

esults | Messages

product_category_name	total_sales
beleza_saude	1409175.41000001
relogios_presentes	1257778.06000001

8.2 Average Delivery Time per Seller

This query calculates the average delivery time for each seller by grouping data in the fact table.

```sql
SELECT
    seller_id,
    AVG(delivery_time) AS avg_delivery_time
FROM
    fact_order_items
GROUP BY
    seller_id
ORDER BY
    avg_delivery_time desc;
```

esults | Messages

seller_id	avg_delivery_time
df683dfda87bf71ac3fc63063fba369d	189
8e670472e453ba34a379331513d6aab1	86
8d92f3ea807b89465643c219455e7369	70

8.3 Number of Orders from Each State

This query counts the number of orders made by customers from each state. The data comes from the fact table joined with the customer dimension table.

```
----------------Query the number of ord
SELECT
    c.customer_state,
    COUNT(f.order_id) AS number_of_orders
FROM
    fact_order_items f
INNER JOIN
    dim_customers c
ON
    f.customer_id = c.customer_id
GROUP BY
    c.customer_state
ORDER BY
    number_of_orders DESC;
```

sults Messages

customer_state	number_of_orders
SP	45706
RJ	13875
MG	12694