ETL Pipeline Project Documentation

1. Introduction

This project focuses on building an ETL (Extract, Transform, Load) pipeline using Python and SQL Server. The pipeline processes data from the Olist Brazilian E-Commerce Public Dataset, including orders, customers, products, reviews, sellers, and payments. The project covers data cleaning, transformation, calculated columns creation, window functions, and loading the data into a SQL Server database.

2. Data Sets

The project uses the following datasets:

Orders: Information about orders (status, dates).
Customers: Customer geographic data.
Products: Product details.
Sellers: Information about sellers.
Order Items: Each item's price and connection to products/orders.
Payments: Payment methods for orders.
Reviews: Customer reviews and ratings.

3. Technology Stack

Python: Used for data transformation and cleaning with Pandas.
MySQL Server: Used to store the transformed data.
SQLAlchemy: To connect and save data to SQL Server.
Jupyter Notebooks: For running the ETL process.

4. Workflow

4.0: Importing the data

Using pandas **read_csv()** to read and import the dataframes from the csv files one by one.

4.1: Cleaning the datasets

Order_reviews

- Since the review_comment_title is not mandatory for analytical purposes and since more than 80% of the column was missing, it would be best to drop it.
- As for review_comment_message it had more data than review_comment_title but still containing lots of null values. The null values are replaced with ‘No comment’.
- Review_creation_date and review_answer_timestamp are set to date type for further manipulations and calculations (if needed in the future)

Orders

- **Order_approved_at** time was set the same as **order_purchase_timestamp** since the confirmation of the order is done when the client pays and purchase it.
- **Order_delivered_carrier_date** and **order_delivered_customer_date** are null for orders that have not yet been delivered thus the null values are filled with 'Not Delivered' and later they are converter to NaT (not a time) if further calculations and aggregations must be done regarding dates
- **Order_purchase_timestamp** , **order_approved_at**, **order_delivered_carrier_date**, **order_delivered_customer_date**, **order_estimated_delivery_date** are converted to date for further aggregations.

Products

- Fixed columns **product_name_lenght** and **product_description_lenght** to **product_name_length** and **product_description_length** respectively.
- Placed the missing **product_category_name** to unknown if null and the null values regarding the **dimensions** of the product to zero if null and not to "unknown" to mitigate having multiple data types on the same column.

Removing duplicates and validate nulls

Using the **function remove_duplicates_and_check_nulls()** to remove the duplicates values while keeping the first one specifically from the unique id columns for each dataframe.

4.2: Creating calculated columns

- **Total Price: Sum of product price and freight value.**
  df_order_items["total_price"] = df_order_items["price"] + df_order_items["freight_value"]
- **Delivery Time: Difference between the delivery date and the order purchase date.**
  df_orders["delivery_time"] = (df_orders["order_delivered_customer_date"] - df_orders["order_purchase_timestamp"]).dt.days
- **Payment Count: Sum of payment installments for each order.**
  df_order_payments['sum_of_payment_installments'] = df_order_payments.groupby('order_id')['payment_installments'].transform('sum')
- **Profit Margin: Subtract freight value from product price to calculate a rough profit estimate.**
  df_order_items['profit_margin'] = df_order_items['price'] - df_order_items['freight_value']

4.3: Using Window Functions Over Partitions (Pandas)

- **Total Sales per Customer: A running total of product price for each customer partitioned by Customer ID.**
  It calculates the cumulative sum of the price for each customer by grouping the data based on customer_id, which means it accumulates the total sales amount for each customer across their orders in the cumulative_sales column. df_merged_sales_df['cumulative_sales'] = df_merged_sales_df.groupby('customer_id')['price'].cumsum()

- **Average Delivery Time per Product Category: A rolling average of delivery time partitioned by product category.**
  df_merged_delivery['rolling_avg_delivery_time'] = (df_merged_delivery.groupby('product_category_name')['delivery_time'].rolling(window=3, min_periods=1).mean().reset_index(level=0, drop=True))
  The line calculates the rolling average delivery time for each product category in the df_merged_delivery DataFrame by applying a window of 3 periods, ensuring at least one observation is needed for the calculation, and then resets the index to align the results with the original DataFrame structure.

4.4: Saving Processed Data to SQL Server (Fact & Dimension Tables)

- Fact Table: Order Items with calculated columns (Total Price, Delivery Time, etc.).

orders = orders[['order_id', 'customer_id', 'order_status', 'order_purchase_timestamp','order_approved_at', 'order_delivered_carrier_date', 'order_delivered_customer_date', 'order_estimated_delivery_date', 'total_price', 'delivery_time','sum_of_payment_installments', 'profit_margin', 'cumulative_sales', 'product_category_name', 'rolling_avg_delivery_time']]

- Dimension Tables: customers,products,sellers,date_dimension

NOTE: Before beginning to load the data I cleaned the dataframes from duplicates using remove_duplicate_and_check_nulls() function again

- Using sqlachlemy function create_engine to connect to the database.
- Created a schema for each table since MySQL required the primary keys to be varchar instead of text, I created a predefined column type schema for all the tables not only the primary keys.
- After the tables are uploaded to the sql server using to_sql functions, using execute() function from sqlalchemy, I altered the tables to create the primary keys

4.5: Validation SQL Queries

- Created a python script to count the number of columns in the sql tables and the pandas dataframes to check if both of them have similar number of rows. This would be more useful if there were a larger number of rows.
- Using information_schema, I checked the columns and datatimes of the tables

SQL Exercises:

- Query total sales per product category from the fact table.

```
SELECT p.product_category_name, SUM(o.total_price) AS total_sales
FROM orders o
JOIN products p
ON o.product_id = p.product_id
GROUP BY
    p.product_category_name
ORDER BY
    total_sales DESC;
```

- Query the average delivery time per seller from the fact table.
  ```
  SELECT o.seller_id, AVG(TIMESTAMPDIFF(DAY, o.order_purchase_timestamp,
  o.order_delivered_customer_date)) AS avg_delivery_time
  FROM orders o
  WHERE o.order_delivered_customer_date IS NOT NULL
  GROUP BY o.seller_id
  ORDER BY avg_delivery_time ASC;
  ```

- Query the number of orders from each state from the customer dimension.
  ```
  SELECT c.customer_state, COUNT(o.order_id) AS num_orders
  FROM customers c
  JOIN orders o
  ON c.customer_id = o.customer_id
  GROUP BY c.customer_state
  ORDER BY num_orders DESC;
  ```