Metyis Data Engineer Assignment

This project simulates a retail company's ETL process for ingesting, cleansing, and storing monthly sales CSV files into a structured, analytics-ready data lake folder.
The pipeline is designed for **Windows compatibility**, given Hadoop native library constraints, but follows scalable PySpark best practices suitable for ADLS or any cloud storage.

Files Execution

1. **verify_env.py**: checks your environment to make sure everything needed for a PySpark data pipeline is properly installed and configured. In particular it verifies Pyspark installation and functionality, Java installation and accessibility, required folder structure (Sales_Data and cleansed) and presence of csv files in the data folder.
   Every check was confirmed.
2. **data_exploration.py**: This script is designed to explore and understand data stored in CSV files before building a PySpark data pipeline. It starts by using Pandas to load all the CSV files, providing a quick overview of the dataset's size, structure, and quality. It checks for missing values, duplicate rows, and inspects important columns like order dates to spot potential formatting issues.
   Main findings where 545 empty rows and more than 800 duplicated rows. Once these were removed, the data types for each column was confirmed.
3. **Data_pipeline.py**: defines a complete PySpark data pipeline designed to work smoothly on Windows systems. It sets up a Spark session to avoid Windows-related errors using Spark and Hadoop config tweaks.
   Aensure data consistency. The data is cleansed through the following operations:
   - Removes rows with missing fields
   - Filters rows where ID is not a valid value (can't be turned into an integer)
   - It converts each column in its natural format (price column to float, quantity to int, order date to datetime)

   After cleaning, the pipeline removes duplicate records using a **window function** that keeps the most recent entry for each combination of order ID and product.

   Finally, the cleaned and deduplicated data is saved as a single Parquet file in a target directory.
   Throughout the process, detailed logging captures progress and errors, ensuring transparency and easier troubleshooting.

Final file format

Parquet was the best choice because it is a columnar storage format that significantly improves query performance and reduces storage space through efficient compression. It works seamlessly with Spark, allowing faster data processing and better handling of large sales datasets. Additionally, Parquet supports partitioning by date, which helps organize the data for easier and quicker access. Overall, Parquet ensures the output is efficient, scalable, and well-suited for future analysis.

Data partitioning

Partitioning the data by order_year and order_month could be the ideal choice. time-based partitioning aligns naturally with the typical ways sales data is analyzed allowing queries to efficiently target specific time ranges without scanning the entire dataset. It also helps keep partitions manageable in size, improving read and write performance.

To implement this partitioning strategy, the key steps would be to:

- **Add partition columns**: Ensure the dataset includes columns for order_year and order_month, derived from the order date timestamp.
- **Write Data Partitioned**: When saving the final dataset, write the Parquet files with partitioning on these columns. In Spark, this can be done using
  .write.partitionBy("order_year", "order_month")
  which organizes data into directory structures reflecting yearn and month.

It would be ideal to monitor the number of files and partition sizes to avoid having too many small partitions or very large ones. This kind of strategy allows to improve performance.

Potential challenges include managing a very large number of partitions if the dataset spans many years or months with sparse data, which could cause overhead. Also, partition pruning only works if queries filter on the partition columns, so it’s important that downstream analyses use these filters to benefit from partitioning. Lastly, writing partitioned data can be more complex when updates or deletes are involved, requiring careful management of partition directories.