**Solution explanation**

Objective: The goal of the task was to deduplicate product records from a dataset. This means ensuring that duplicate entries, representing the same product (or very similar products), are combined into a single record. The challenge involved handling potential duplicates based on various attributes of the products, such as the product identifier, name, brand, size, and color.

Step 1: Understanding the Problem and Dataset

*Initial Thoughts:* The dataset presented a list of products, and our objective was to identify duplicates based on an identifying factor. I began by exploring the dataset to understand the structure of the data and the available columns.

*Key Column:* The dataset contains a product\_identifier column, which serves as a unique identifier for each product. This column was the primary basis for deduplication.

*Additional Attributes:* In the absence of a product identifier (i.e., if the column was missing), I decided to consider other attributes such as product\_name, brand, size, color, and materials to identify duplicates. These columns represent key product characteristics that should be similar for the same product.

Step 2: Analyzing the Approach

I followed these guidelines for the approach:

* Primary Deduplication: If product\_identifier was available, it would serve as the most reliable indicator of product uniqueness. This allowed me to group records by this identifier.
* Backup Plan: If the product\_identifier column was not present, I chose to group by a set of key attributes: product\_name, brand, size, color, and materials.

This would still allow for effective deduplication, although it might not be as precise as the product identifier approach.

* Handling of Multiple Values: For columns that had multiple values for the same product (e.g., multiple colors or materials for the same product), I needed a way to merge these values into a single record. I opted to use the merge\_values function, which concatenates distinct, non-null values into a single string.

This way, all variations of a product (in terms of size, color, etc.) are preserved but represented in a compact manner.

Step 3: Code Implementation

import pandas as pd

# Reading the dataset

df = pd.read\_parquet(r"C:\path\_to\_file\veridion\_product\_deduplication\_challenge.snappy.parquet", engine="pyarrow")

# Function to merge duplicate values in a seriesdef merge\_values(series):

return ', '.join(series.dropna().astype(str).unique())

# Checking if 'product\_identifier' existsif 'product\_identifier' in df.columns:

deduplicated\_df = df.groupby('product\_identifier').agg(merge\_values).reset\_index()else:

deduplicated\_df = df.groupby(['product\_name', 'brand', 'size', 'color', 'materials']).agg(merge\_values).reset\_index()

# Saving the result

output\_path = "deduplicated\_products.parquet"

deduplicated\_df.to\_parquet(output\_path, index=False)

print(f"File successfully deduplicated and saved as {output\_path}")

Step 4: Results & Impact

* Deduplication Process: After executing the code, the number of rows decreased from 21,946 to 1,806, which indicates a significant reduction of duplicate records. This demonstrates that the deduplication process was effective, consolidating data where necessary.
* Merge Strategy: By using the merge\_values function, all unique data from the grouped columns were preserved, allowing for a concise and readable representation of the deduplicated data. This ensures that the final dataset maintains all relevant details without redundant rows.
* Scalability Consideration: Although the dataset was relatively small (less than 22,000 rows), the method I implemented is scalable. If the dataset size were to grow, I could consider implementing parallel processing or optimizing memory usage. The use of PyArrow for reading and writing Parquet files is already an efficient method for handling large data files.

Step 5: Decision-Making & Reflections

* Use of Grouping: The decision to group by product\_identifier first was logical because it’s the most accurate way to deduplicate products. If it wasn’t available, falling back to grouping by product characteristics made sense, though it could result in some false positives in edge cases (e.g., similar products but slightly different in color or size).
* Function Choice: The merge\_values function was a suitable way to handle multiple entries in columns that had non-null values. However, an alternative could have been to use first() or last() to select a specific value when multiple options exist, depending on business rules.
* Scalability: The solution would easily scale for larger datasets as long as sufficient memory and processing power are available. Using tools like Dask or Apache Spark could improve performance for billions of rows, but for this dataset, the method was optimal.

Step 6: Future Considerations

* Handling More Complex Duplicates: In future, if the dataset becomes more complex (e.g., with even more columns or more ambiguity), I could refine the deduplication strategy by considering fuzzy matching or using machine learning models to identify product duplicates based on similarity, rather than exact matches.
* Documentation & Commenting: The code could benefit from more comments to explain why certain decisions were made (e.g., why merge\_values was chosen). This would help improve the readability and clarity of the approach.

Conclusion: This approach successfully deduplicated the dataset based on available product identifiers and attributes. The deduplication process preserved the uniqueness of each product while consolidating multiple attributes in a readable format. The solution is scalable and can be further optimized for larger datasets in the future. The decision-making process was aligned with best practices, and the problem was solved efficiently using Python and Pandas.