2890. Reshape Data Melt

Easy

Leetcode Link

Problem Description

The given LeetCode problem presents a `DataFrame` named `report` which contains sales data for different products across four quarters (quarter_1, quarter_2, quarter_3, quarter_4). Each row of this `DataFrame` represents a product and the sales figures for each of the four quarters are in separate columns. The task is to reshape this data such that the resulting `DataFrame` has a row for each product and quarter combination. Essentially, it involves converting the wide format of the `DataFrame` (where quarters are spread across columns) into a long format (where quarter data is stacked into single column with corresponding sales figures).

The expected output is a `DataFrame` with three columns: 'product', 'quarter', and 'sales'. Each row should contain a product name, a specific quarter, and the sales for that product in that quarter.

Intuition

The intuition behind the solution is that we want to "melt" the wide `DataFrame` into a long `DataFrame`. In pandas, the `melt` function is used for just such a transformation. It takes the following parameters:

- `id_vars`: The column(s) of the old `DataFrame` to preserve as identifier variables. In this case, it's the 'product' column, as we want to keep that fixed for each entry.
- `var_name`: The name to give the variable column that will hold the names of the former columns. We will name it 'quarter' since it will contain the names of the quarter columns.
- `value_name`: The name to give the value column that will contain the values from the former quarter columns. We'll call this 'sales' to indicate that these values represent the sales amounts.

By applying the `melt` method to the `report` dataframe, it will take each entry from the quarter-specific columns and place it into its own row, associated with the appropriate product and tagged with the corresponding quarter, achieving the desired reshaping of the data.

Solution Approach

The solution makes use of the `melt` function from the pandas library. This function is designed to transform a DataFrame from a wide format to a long format, which is exactly what is required in the problem. The `melt` function can be seen as a way to 'unpivot' or 'reshape' the data.

Here's a step-by-step walkthrough of the `meltTable` function shown in the reference solution:

- We start by passing the `report` DataFrame to the `meltTable` function.
- Within the function, we call `pd.melt` on the `report` DataFrame.

The `pd.melt` function is called with the following arguments:

- `id_vars=['product']`: This specifies that the 'product' column should stay as is and not be unpivoted. This column is used as the identifier variable.
- `var_name='quarter'`: This argument tells pandas to name the new column that holds the 'variables', which were originally the column names (quarter_1, quarter_2, quarter_3, quarter_4), as 'quarter'.
- `value_name='sales'`: This specifies that the new column that holds the values from the variable columns should be named 'sales'.

The `melt` function processes the DataFrame `report` by keeping the 'product' column fixed and 'melting' the quarter columns. For each product, it creates a new row for each quarter column, filling in the 'quarter' column with the quarter column name (e.g., 'quarter_1') and the 'sales' column with the corresponding sales value.

As a result, what was previously structured as one row per product with multiple columns for each quarter becomes multiple rows for each product, with each row representing a different quarter.

By using pandas and its `melt` function, the solution effectively harnesses the power of an established data manipulation tool to accomplish the task in a concise and efficient manner without the need for writing complex data reshaping code from scratch.

Example Walkthrough

Let's say we have the following small 'report' DataFrame as an example:

product	quarter_1	quarter_2	quarter_3	quarter_4
ProductA	150	200	250	300
ProductB	100	150	200	250

We want to reshape this data to create a 'long' format DataFrame, where each product and quarter combination gets its own row. Here's how we apply the solution approach:

1. We pass this 'report' DataFrame to our `meltTable` function.
2. Inside `meltTable`, we use `pd.melt` and specify three key parameters:
    - `id_vars=['product']` ensures that the 'product' column is preserved in the transformation.
    - `var_name='quarter'` creates a new column named 'quarter', which will contain the names of the original columns that represented each quarter.
    - `value_name='sales'` specifies that the values from those quarter columns should be placed in a new column called 'sales'.

After calling `pd.melt` with these parameters, we get the following DataFrame:

product	quarter	sales
ProductA	quarter_1	150
ProductA	quarter_2	200
ProductA	quarter_3	250
ProductA	quarter_4	300
ProductB	quarter_1	100
ProductB	quarter_2	150
ProductB	quarter_3	200
ProductB	quarter_4	250

The resulting DataFrame has the three columns: 'product', 'quarter', and 'sales', with each row containing a specific combination of product and quarter with the corresponding sales figure. This transformation enables a more detailed analysis of sales data by quarter for each product.

Python Solution

```python
import pandas as pd  # Import the pandas library with alias 'pd'

def melt_table(report: pd.DataFrame) -> pd.DataFrame:
    # Function to transform the input DataFrame into a format where each row
    # represents a single observation for a specific quarter and product.

    # `pd.melt()` Convert the given DataFrame from wide format to long format.
    # `id_vars`: Column(s) to use as identifier variables.
    # `var_name`: Name of the new column created after melting that will hold the variable names.
    # `value_name`: Name of the new column created that will contain the values.
    melted_report = pd.melt(report, id_vars=['product'], var_name='quarter', value_name='sales')
    return melted_report  # Return the melted DataFrame

# Usage example (not part of the required code rewrite, for illustration purposes):
# Assuming 'report' is a DataFrame structured with 'product' as one of the columns
# and other columns represent sales data for each quarter.
# Example structure of 'report' before melting:
# product  Q1  Q2  Q3  Q4
# A        10  15  20  25
# B         5  10  15  20
# After calling melt_table(report), the result will be:
# product quarter sales
# A       Q1      10
# A       Q2      15
# A       Q3      20
# A       Q4      25
# B       Q1      5
# B       Q2      10
# B       Q3      15
# B       Q4      20
```

Java Solution

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

class SalesReport {

    // A method to transform a report into a melted format where each row represents a single observation
    public static List<Map<String, String>> meltTable(List<Map<String, String>> report) {
        List<Map<String, String>> meltedReport = new ArrayList<>();

        // Loop over each row (each map is a row with the product and sales data)
        for (Map<String, String> row : report) {
            String product = row.get("product");
            // Loop over each entry in the map, which represents the columns in the original table
            for (Map.Entry<String, String> entry : row.entrySet()) {
                if (!entry.getKey().equals("product")) { // Ignore the product column for melting
                    // Create a new map for the melted row
                    Map<String, String> meltedRow = new HashMap<>();
                    meltedRow.put("product", product);
                    meltedRow.put("quarter", entry.getKey()); // The column name becomes the quarter
                    meltedRow.put("sales", entry.getValue()); // The value remains the sale amount
                    meltedReport.add(meltedRow);
                }
            }
        }
        return meltedReport; // Return the melted DataFrame
    }

    // Example usage
    public static void main(String[] args) {
        List<Map<String, String>> report = new ArrayList<>();
        Map<String, String> row1 = new HashMap<>();
        row1.put("product", "A");
        row1.put("Q1", "10");
        row1.put("Q2", "15");
        row1.put("Q3", "20");
        row1.put("Q4", "25");
        report.add(row1);

        Map<String, String> row2 = new HashMap<>();
        row2.put("product", "B");
        row2.put("Q1", "5");
        row2.put("Q2", "10");
        row2.put("Q3", "15");
        row2.put("Q4", "20");
        report.add(row2);

        List<Map<String, String>> meltedReport = meltTable(report);

        // Print melted report
        for (Map<String, String> meltedRow : meltedReport) {
            System.out.println(meltedRow);
        }
    }
}
```

C++ Solution

```cpp
#include <pandas/pandas.h> // Include the pandas C++ library (note: a C++ pandas-like library doesn't exist, but assuming it for the

class ReportTransformer {
public:
    // Transforms the input DataFrame into a format where each row represents a single observation for a specific quarter and product
    pandas::DataFrame meltTable(const pandas::DataFrame& report) const {
        // `melt`: Convert the given DataFrame from wide format to long format.
        // `idVars`: Vector of column names to use as identifier variables.
        // `varName`: Name of the new column created after melting that will hold the variable names.
        // `valueName`: Name of the new column created that will contain the values.
        pandas::DataFrame meltedReport = report.melt(
            {"product"}, // idVars
            "quarter",   // varName
            "sales"      // valueName
        );

        return meltedReport; // Return the melted DataFrame
    }
};

// Usage example:
// Assuming 'report' is a pandas::DataFrame structured with 'product' as one of the columns
// and other columns represent sales data for each quarter like Q1, Q2, Q3, Q4.
// Example structure of 'report' before melting:
// product  Q1  Q2  Q3  Q4
// A        10  15  20  25
// B         5  10  15  20
// After calling meltTable(report), the result will be:
// A  Q1  10
// A  Q2  15
// A  Q3  20
// A  Q4  25
// B  Q1  5
// B  Q2  10
// B  Q3  15
// B  Q4  20
// ...and so on for each product and quarter.
```

Typescript Solution

```typescript
interface ProductReport {
    product: string;
    [key: string]: string | number; // Represents sales data for each quarter with dynamic keys
}

interface MeltedReport {
    product: string;
    quarter: string;
    sales: number;
}

function meltTable(report: ProductReport[]): MeltedReport[] {
    // Function to transform the input array of objects into a format
    // where each entry represents a single observation for a specific quarter and product.

    let meltedReport: MeltedReport[] = [];

    // Loop over each product report
    report.forEach((productReport) => {
        // Loop over each property in the product report object
        for (const [key, value] of Object.entries(productReport)) {
            // Skip the 'product' key as it's the identifier
            if (key !== 'product') {
                // Create an object for each quarter with sales data and push it into the meltedReport array
                meltedReport.push({
                    product: productReport.product,
                    quarter: key,
                    sales: value as number // Assuming the value is always a number for sales data
                });
            }
        }
    });

    return meltedReport; // Return the transformed data
}

// Usage example:
// Assuming 'report' is an array of objects structured with 'product' as one of the properties
// and other properties represent sales data for each quarter.
// Example structure of 'report' before melting:
// [
//   { product: 'A', Q1: 10, Q2: 15, Q3: 20, Q4: 25 },
//   { product: 'B', Q1: 5, Q2: 10, Q3: 15, Q4: 20 }
// ];
// After calling meltTable(report), the result will be:
// [
//   { product: 'A', quarter: 'Q1', sales: 10 },
//   { product: 'A', quarter: 'Q2', sales: 15 },
//   ...
//   { product: 'B', quarter: 'Q1', sales: 5 },
//   { product: 'B', quarter: 'Q2', sales: 10 },
//   ...
// ]
```

Time and Space Complexity

Time Complexity

The `meltTable` function involves the `pd.melt` operation from pandas. The time complexity of this operation depends on the size of the input DataFrame. If we assume the input DataFrame has n rows (excluding the header) and c columns (including the 'product' column), then the `pd.melt` function would iterate through all $(n * (n - 1))$ elements once, converting them into $(n * (n - 1))$ rows of the melted DataFrame. Thus, the time complexity is $O(n * (n - 1))$, which simplifies to $O(n * n)$.

Space Complexity

Regarding the space complexity, `pd.melt` generates a new DataFrame that has $(n * (n - 1))$ rows and 3 columns (`['product', 'quarter', 'sales']`). Hence, the space required for the new DataFrame is proportional to the number of elements in this new structure, which gives us the space complexity of $O(n * (n - 1) * 3)$. Since we tend to ignore constant factors in Big O notation, this simplifies to $O(n * n)$.