How to Unpivot a PySpark DataFrame with the Melt Function

Authored by

stats writer

February 6, 2026

The process of unpivoting a PySpark DataFrame is fundamentally about reshaping data from a wide format, where variables are spread across multiple columns, into a long format, where those variable names and their corresponding values are stacked into new rows. This transformation is crucial for normalizing datasets and aligning them with the requirements of statistical analysis and visualization tools.

Consider the typical scenario of student grade data. In the wide format, each subject score occupies its own column, tied to a single row ID. The input structure below clearly illustrates this:

Input DataFrame (Wide Format):

id	name	subject_1	subject_2	subject_3
1	John	85	90	95
2	Jane	80	75	85

When we apply the unpivot operation, the subject columns collapse. The resulting structure, the long format, greatly simplifies grouped calculations and comparisons across subjects, making the data highly efficient for further processing:

Unpivoted DataFrame (Long Format):

id	name	subject	score
1	John	subject_1	85
1	John	subject_2	90
1	John	subject_3	95
2	Jane	subject_1	80
2	Jane	subject_2	75
2	Jane	subject_3	85

This transformation allows for significantly easier analysis and visualization of the data by reducing redundancy and increasing the number of observations per key identifier. The following detailed example demonstrates the practical implementation of this technique using PySpark code.

Unpivot a PySpark DataFrame (With Detailed Example)

Prerequisites for PySpark Unpivoting

To effectively unpivot a dataset in PySpark, we typically use the built-in functionality available on the PySpark DataFrame object. While some data manipulation libraries offer a dedicated `melt` or

`unpivot` function, Spark's implementation often relies on creating a temporary pivot structure first, or utilizing specific SQL expressions, depending on the Spark version and complexity. However, modern versions of PySpark offer a dedicated `unpivot` method, simplifying the workflow significantly.

Before diving into the code, it is crucial to ensure you have a running SparkSession initialized, as this provides the entry point to all PySpark functionality. The process demonstrated here assumes that the data has already been aggregated or summarized, resulting in a dataset structure that is currently in the wide format, ready for transformation back to the long format. This methodology is particularly useful when reversing a previous aggregation step, such as converting a Pivot table back into transactional data.

The following sections will walk through a practical example, demonstrating how to first create a pivot table from raw data and then successfully reverse that operation using the powerful `unpivot` function available in the PySpark SQL module. This approach provides a clear, cyclical view of data restructuring.

Step-by-Step Example: Preparing the PySpark DataFrame

For our demonstration, we will begin by constructing a basic PySpark DataFrame that tracks points scored by various basketball players, categorized by team and position. This initial dataset is inherently in the long format, making it easy to perform aggregations later on. We initialize the necessary components and define the schema before displaying the resulting DataFrame.

We leverage the SparkSession to create the DataFrame from a list of lists, ensuring that our data is properly ingested and structured for the subsequent pivot and unpivot operations. Pay close attention to the structure of the input data, which lists individual scoring events.

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = ,
,
,
,
,
,
]

```
#define column names
columns =

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+--------+------+
|team|position|points|
+----+--------+------+
| A| Guard| 11|
| A| Guard| 8|
| A| Forward| 22|
| A| Forward| 22|
| B| Guard| 14|
| B| Guard| 14|
| B| Forward| 13|
| B| Center| 7|
+----+--------+------+
```

This initial setup provides the raw data necessary to proceed with the aggregation and pivoting steps, establishing the groundwork for demonstrating the reverse operation of unpivoting.

Pivoting the Data: Transitioning to the Wide Format

The next logical step is to transform this raw, transactional data into a pivot table, which naturally results in the wide format. We achieve this by grouping the data by the 'team' column, pivoting based on the 'position' column, and calculating the aggregated sum of the 'points' for each unique combination. This summarizes the total points scored by each team across all player positions.

This pivot operation is crucial because it creates the exact structure that necessitates the subsequent unpivot operation. Notice how the distinct values from the 'position' column ('Center', 'Forward', 'Guard') become new columns in the resulting DataFrame, making the dataset wider and shorter, suitable for certain forms of reporting.

```
#create pivot table that shows sum of points by team and position
df_pivot = df.groupBy('team').pivot('position').sum('points')

#view pivoted DataFrame
```

```
df_pivot.show()

+----+------+-------+-----+
|team|Center|Forward|Guard|
+----+------+-------+-----+
| B| 7| 13| 28|
| A| null| 44| 19|
+----+------+-------+-----+
```

The resulting DataFrame, named **df_pivot**, clearly shows the sum of points for each team categorized by position. Notably, Team A does not have a 'Center' listed in the original data, which manifests as a **null** value in the pivoted output. This is a common characteristic of wide-format data and must be addressed during the unpivoting process if we require a clean long format.

Executing the Unpivot Operation in PySpark

To reverse the pivot operation and restore the data to its original, normalized structure (the long format), we utilize the dedicated `unpivot` function. This function requires specific parameters: the columns that should remain fixed (the ID columns), the columns that are to be unpivoted (the measure columns), and the names for the two new resulting columns (the variable column and the value column).

In our example, 'team' is the fixed identifier column. The position columns ('Center', 'Forward', 'Guard') are the measure columns we want to collapse. We define the new variable column as 'position' and the new value column as 'points'. The syntax clearly defines the transformation mapping, ensuring the data is correctly reshaped without loss of context.

```
#unpivot DataFrame
df_unpivot = df_pivot.unpivot(, , 'position', 'points')

#view unpivoted DataFrame
df_unpivot.show()

+----+--------+------+
|team|position|points|
+----+--------+------+
| B| Center| 7|
| B| Forward| 13|
| B| Guard| 28|
| A| Center| null|
| A| Forward| 44|
```

| A| Guard| 19|
+----+--------+------+

The output DataFrame, **df_unpivot**, has successfully been transformed back to a structure closely resembling the initial long format, having only three columns: 'team', 'position', and 'points'. This recovery of structure allows for seamless integration into other analytical pipelines that require normalized data inputs, confirming the effectiveness of the `unpivot` function.

Refining the Output: Handling Null Values

As observed in the pivoting step, the absence of a 'Center' for Team A resulted in a **null** value when unpivoted. While having nulls is sometimes acceptable, it often complicates subsequent aggregations or visualizations. Therefore, a final refinement step involves filtering out these rows where the 'points' value is null, ensuring the dataset is clean and complete for analysis.

We use the `filter` function in PySpark, along with the **isNotNull()** method applied to the 'points' column, to strictly retain only those rows that contain actual point totals. This practice is crucial for maintaining data integrity and accuracy, especially in performance metrics analysis where zero values might be confused with missing data.

#filter out rows where points column is null
df_unpivot.filter(df_unpivot.points.isNotNull()).show()

+----+--------+------+
|team|position|points|
+----+--------+------+
B	Center	7
B	Forward	13
B	Guard	28
A	Forward	44
A	Guard	19
+----+--------+------+

This final, cleaned DataFrame represents the successful completion of the unpivot operation, effectively reversing the pivot transformation and eliminating extraneous null entries.

Note: You can find the complete documentation for the PySpark **unpivot** function by visiting the official Apache Spark API documentation.

Further Learning and Related Topics

Mastering the **pivot** and **unpivot** operations is fundamental to effective data manipulation in PySpark. These techniques are often used in preparation for advanced analytical tasks such as machine learning feature engineering or complex business intelligence reporting. Understanding how to fluidly move between wide format and long format ensures flexibility in handling diverse data requirements.

To deepen your expertise in data wrangling using the PySpark library, consider exploring tutorials related to other common data transformation tasks. These complementary skills will enhance your ability to preprocess and analyze massive datasets efficiently.

The following tutorials explain how to perform other common tasks in PySpark:

How to perform joins between multiple DataFrames.
Techniques for handling missing data using imputation or deletion.
Methods for optimizing performance when dealing with extremely large datasets.