# How to Unpivot a PySpark DataFrame Easily

Authored by

**stats writer**

January 20, 2026

The ability to efficiently restructure and reshape data is paramount in modern data engineering and analysis. When working with large datasets, the tools provided by the PySpark library are indispensable. PySpark, the Python API for Apache Spark, allows users to handle massive data operations distributed across clusters. Central to this process is the DataFrame, a distributed collection of data organized into named columns, conceptually equivalent to a table in a relational database or a DataFrame in R/Python (like Pandas). One of the most common data transformation tasks involves converting data between a wide format (many columns) and a long format (fewer columns, but more rows). This transformation is often referred to as pivoting and unpivoting (or melting). The **unpivot** function is the dedicated method provided by PySpark to perform this reversal, transforming wide data back into its normalized, long format.

Understanding the precise syntax and application of the `unpivot` function is vital for any data professional utilizing Spark. This detailed guide demonstrates the practical implementation of `unpivot` using a basketball statistics example, emphasizing clarity and efficiency in code execution. You can use the **unpivot** function to unpivot a PySpark DataFrame.

## Understanding the PySpark Unpivot Operation

The unpivot function performs the reverse operation of a pivot. A pivot table takes a long format dataset and aggregates it, spreading unique values from one column (the pivoting column) into multiple new columns. Conversely, unpivot consolidates multiple columns back into two new columns: one column containing the original column names (the variable column) and another column containing the values associated with those column names (the value column). This process transforms the data back into its normalized, long format.

The primary goal of using the **unpivot** function is to simplify analysis where the variable of interest is currently spread across many columns. For instance, if a dataset tracks monthly sales figures across 12 separate columns (Jan, Feb, Mar, etc.), unpivoting allows these 12 columns to be collapsed into two: a 'Month' column and a 'Sales Value' column. This structure is typically preferred for plotting time series data, performing group-by operations, or inputting data into many statistical packages. The use of a dedicated function ensures the operation is executed efficiently across the distributed environment managed by Apache Spark.

It is important to note that the PySpark `unpivot` function requires careful specification of three critical components: first, the identifying columns (columns that should remain fixed); second, the value columns (the columns to be melted into rows); and third, the names for the newly created variable and value columns. If these parameters are incorrectly specified, the resulting DataFrame structure will be incorrect or incomplete. The following example shows how to use this syntax in practice, beginning with the creation of our initial dataset.

## Initial Data Setup and PySpark DataFrame Creation

To illustrate the `unpivot` operation effectively, we must first establish a sample dataset. We will create a PySpark DataFrame containing basketball statistics, tracking points scored by players across different teams and positions. This initial dataset is already in a normalized, long format, which will serve as our baseline for comparison after the entire pivot-unpivot cycle.

We start by initializing a **SparkSession**, which is the entry point to programming Spark with the Dataset and DataFrame API. Following the session creation, we define the data structure, including the list of rows (the data) and the column schema (the column names). The sample data includes three fields: `team`, `position`, and `points`. This setup ensures that the data is ready for distributed processing.

Suppose we create the following PySpark DataFrame that contains information about the points scored by various basketball players:

**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|
+----+--------+------+

## Step 1: Pivoting the Data for Aggregation

To demonstrate the reverse operation (unpivot), we must first transform the data into a wide format using the `pivot` operation. This process aggregates the initial data, creating a summary pivot table based on team and position. This wide format is typical of summary reports or data structures generated by SQL aggregation queries, and it is the format that the `unpivot` function is designed to normalize.

We will transform our long-format `df` into a wide format `df_pivot` by performing a grouping operation. We use `groupBy('team')` to define the fixed rows, `pivot('position')` to specify the column whose unique values will become new headers, and `sum('points')` to define the aggregation logic. This operation condenses multiple rows per team into a single row, providing a concise summary of total points per position.

The resulting DataFrame shows the sum of the points values for each team and position. Crucially, where a team lacks data for a specific position (like Team A lacking a Center entry), a null value is introduced in the aggregated table. This is an important detail we must manage during the unpivoting phase.

We can use the following syntax to create a pivot table using **team** as the rows, **position** as the columns and the sum of **points** as the values within the pivot table:

**#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|
+----+------+-------+-----+

## Step 2: Applying the Unpivot Function

The core objective of this process is achieved by utilizing the unpivot function to restore the wide `df_pivot` back to its original long format. This transformation involves melting the column headers ('Center', 'Forward', 'Guard') back into rows, making the data highly normalized and suitable for further statistical computation or database ingestion.

The **unpivot** function requires four main components to execute correctly: the fixed identifier columns (columns that are not melted, like `team`), the columns to be melted (the position columns), the name for the new variable column (which holds the original column names, e.g., `position`), and the name for the new value column (which holds the aggregated data, e.g., `points`). Specifying these parameters precisely is essential for a successful transformation.

By executing this function, we effectively stack the position columns on top of each other, creating new rows and associating the corresponding point totals. In order to unpivot this DataFrame, we can use the **unpivot** function with the following syntax:

**#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|
+----+--------+------+

## Analyzing the Unpivoted Result

The resulting DataFrame, `df_unpivot`, successfully demonstrates the structural transformation.

The data is now back to the original format with three columns: `team`, `position`, and `points`. It is crucial to observe that the row representing Team A's Center position still contains a null value in the `points` column. This null propagates directly from the pivot table, confirming that the unpivot operation accurately preserves the underlying data state.

While preserving the null value is often required for complete data representation, analysts frequently need to filter these out to ensure clean and meaningful calculations downstream. This is particularly true if this data is used for training models or calculating descriptive statistics where nulls can skew results or cause function failure. Therefore, the next logical step in the data pipeline is to cleanse the dataset by removing records where the value column lacks data.

The data integrity of the unpivot operation depends entirely on the aggregation performed during the pivot. Since we summed the points, the resulting values reflect these aggregated totals. If the original pivot had used a different aggregation (like average or count), the unpivoted values would reflect those metrics instead. The key takeaway is that the structure is restored, and the values represent the column-wise aggregates.

## Step 3: Handling Missing Values

To finalize the data cleaning process and ensure the unpivoted DataFrame is ready for use, we must address the row containing the null value. In PySpark, the most straightforward way to remove rows based on missing values in a specific column is by using the **filter** function in conjunction with `isNotNull()`.

The **filter** function allows us to define a conditional statement that dictates which rows are retained. By chaining `.isNotNull()` onto the specified column (`df_unpivot.points`), we instruct PySpark to keep only those rows where the `points` column contains a non-missing (non-null) value. This method is generally preferred over a general `dropna()` call, as it provides precise control over which columns are checked for nulls, ensuring that only the relevant missing data is removed.

Lastly, we can filter out any rows with a null value in the **points** column by using the **filter** function:

**#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 DataFrame has now been unpivoted and there are no rows with a null value in the **points** column. The output represents a clean, ready-to-use dataset that is compliant with typical long-format data requirements.

## Conclusion and Best Practices for Data Reshaping

The **unpivot** function in PySpark is an indispensable component of the data preparation toolkit, enabling powerful and efficient transformations from wide, aggregated formats to normalized, long formats. This capability is foundational for maintaining data compliance with "tidy data" principles, which are crucial for effective machine learning and statistical modeling workflows.

When implementing the unpivot operation, adherence to best practices ensures optimal results and maintainable code:

**Clarity in Naming:** Ensure the new column names (for variable and value) clearly describe the data they represent, enhancing readability for subsequent analysis. For example, using "metric_type" and "metric_value" if the columns represent diverse metrics.
**Parameter Verification:** Always double-check the lists passed as ID columns and value columns to prevent accidental data omission or inclusion. Misidentifying the columns to be melted is the most common error in this process.
**Null Handling:** Explicitly decide whether null values should be retained or filtered out. If filtration is chosen, use the specific `filter(col.isNotNull())` pattern to ensure precision, especially when dealing with potentially sparse aggregated data generated by a pivot table.

Mastering `pivot` and `unpivot` operations grants significant flexibility in preparing vast datasets for diverse analytical tasks within the distributed framework of Apache Spark, ensuring data structure adaptability regardless of the initial input format.

**Note**: You can find the complete documentation for the PySpark **unpivot** function here.

## Related PySpark Tutorials

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

How to use the Window function in PySpark.
A guide to performing Joins in PySpark DataFrames.
Optimizing data reading using PySpark and Parquet files.