How to Generate Crosstabs with PySpark DataFrames

Authored by

stats writer

January 3, 2026

The **crosstab** function in PySpark is an indispensable tool for data analysts seeking to understand the relationships between different attributes within their datasets. This functionality facilitates the creation of a cross-tabulation (or contingency table) involving two columns of a DataFrame. Essentially, the function accepts the names of the two desired columns as parameters and generates a new DataFrame that summarizes the frequency count of unique pairs of values.

Generating a cross-tabulation is a fundamental step in exploratory data analysis (EDA), particularly when working with categorical variables. This output provides an immediate, aggregated view, allowing users to quickly visualize and quantify the association or independence between the attributes being analyzed. This method is far more intuitive than scanning raw data and is highly efficient when processing the massive datasets characteristic of the Spark environment.

Understanding Cross-Tabulation in Data Analysis

At its core, a **cross-tabulation** is a matrix that organizes the frequency distribution of two categorical variables simultaneously. Instead of just counting how many times each value appears in isolation, a crosstab reveals how often specific combinations of values occur. This statistical technique is fundamental for testing hypotheses regarding the relationship between attributes, such as whether a player's position is statistically dependent on their team assignment.

The resulting table structure is straightforward: the unique values from the first chosen column (`col1`) form the row headers, while the unique values from the second chosen column (`col2`) form the column headers. Each cell within the matrix then contains the count of records in the original DataFrame where that specific row value and column value combination appears. This aggregate view drastically simplifies large-scale data exploration.

While standard Structured Query Language (SQL) or native Python libraries can achieve this, utilizing the specialized `crosstab` function within PySpark ensures efficient, distributed processing across potentially massive datasets, leveraging the power of the **Apache Spark** framework.

The PySpark DataFrame.crosstab() Function

The implementation of cross-tabulation in PySpark is highly streamlined. The function is accessed directly via a DataFrame instance, requiring only the names of the two columns intended for analysis. Unlike some other aggregation methods, the `crosstab` function explicitly handles the grouping and counting internally, simplifying the code required for this common task.

The basic syntax for creating a crosstab in PySpark is concise and clear:

df.crosstab(col1='team', col2='position').show()

In this illustrative example, we instruct Spark to analyze the relationship between the **team** column, which will define the rows of the output table, and the **position** column, which will define the columns. The resulting DataFrame will be a tabulation where cell values represent the total frequency count for each unique pairing of team and position found in the input data.

It is important to remember that the output of `crosstab` is itself a new DataFrame. This allows users to chain additional DataFrame operations onto the result, such as filtering, statistical calculations, or joining with other datasets, enabling deeper analysis beyond simple frequency counts. The following sections demonstrate this syntax in a complete, practical scenario.

Practical Example: Setting Up the PySpark Environment

To illustrate the functionality of the `crosstab` operation, we will define a simple PySpark DataFrame containing sample data related to basketball player statistics. This dataset includes three key columns: **team**, **position**, and **points**. Our primary goal is to use PySpark's `crosstab` to determine the distribution of player positions across different teams.

The setup requires initiating a Spark Session and defining both the underlying data structure and the column schema. This preparatory step ensures that our input data is properly formatted and accessible within the Spark ecosystem:

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| Forward| 14|
| B| Forward| 13|
| B| Forward| 7|
| C| Forward| 11|
| C| Guard| 10|
+----+--------+------+
```

The resulting table, displayed above, confirms that we have 10 records detailing players across three teams (A, B, C) and two distinct positions (Guard, Forward). Note that the 'points' column is irrelevant for the cross-tabulation itself, as `crosstab` focuses solely on the counts of the chosen categorical variables.

Generating the Cross-Tabulation

With the input DataFrame successfully defined, we can now proceed to apply the `crosstab` function. We aim to summarize the data by using **team** as the primary grouping variable (rows) and **position** as the secondary grouping variable (columns). This operation provides immediate insights into team composition.

We execute the function using the following clean syntax, specifying `'team'` for `col1` and `'position'` for `col2`:

```
#create crosstab using 'team' and 'position' columns
df.crosstab(col1='team', col2='position').show()

+-------------+-------+-----+
|team_position|Forward|Guard|
+-------------+-------+-----+
| B| 3| 1|
```

```
| C| 1| 1|
| A| 2| 2|
+-------------+-------+-----+
```

The execution yields a new DataFrame, which is the cross-tabulation. Notice the structure: the new DataFrame automatically names the row column `team_position` (a concatenation of the input column names, a convention used by Spark). The remaining columns are named after the unique values found in the `position` column: **Forward** and **Guard**.

The values within the cells represent the joint frequencies. For example, in the row corresponding to Team B, the value 3 under the 'Forward' column indicates that exactly three records in the original data satisfied both criteria simultaneously (Team = B AND Position = Forward). This output is highly valuable for initial statistical inspection.

Interpreting the Results of the Crosstab

The resulting cross-tabulation summarizes the entire population distribution based on the two chosen variables. Detailed analysis of the cell counts allows us to draw specific conclusions about the composition of each team, confirming the counts derived from the original dataset:

There are **3** players who are Forwards on Team B.
There is **1** player who is a Guard on Team B.
There is **1** player who is a Forward on Team C.
There is **1** player who is a Guard on Team C.
There are **2** players who are Forwards on Team A.
There are **2** players who are Guards on Team A.

This organized presentation of joint frequencies is far superior to manually iterating through the source data. By providing clear evidence of distribution, the crosstab is foundational for further statistical testing, such as calculating Chi-squared statistics to formally test the independence of team and position.

Advanced Considerations and Usage Notes

While the `crosstab` function is generally straightforward, users should be aware of several important technical considerations when deploying it in large-scale data processing workflows:

First, the function is optimized for categorical variables with a relatively small number of unique values (low cardinality). If one of the columns used in the crosstab has thousands or millions of unique values, the resulting table will be extremely wide. In distributed computing environments like Spark, generating a very wide table can lead to performance degradation or memory overflow

when the result set is collected or displayed, as wide DataFrames place high pressure on the driver node.

Second, handling null values is important. Cross-tabulation counts only non-null combinations. If either `col1` or `col2` contains nulls, those records are effectively excluded from the joint frequency count, making it crucial to understand the cleanliness of the input data. If counting nulls is necessary, a preliminary step to fill nulls with a placeholder value (e.g., 'Unknown') must be executed before calling `crosstab`.

Third, for statistical completeness, users often require not just the raw counts (as provided by `crosstab`) but also marginal totals, row percentages, or column percentages. The native function only provides the joint frequency counts. To derive percentages, you must apply further PySpark DataFrame operations, typically involving functions like `withColumn`, `sum`, and `groupBy` on the resulting DataFrame to calculate the necessary marginal totals for normalization.

Further Exploration of PySpark Aggregation

The `crosstab` function is just one tool in PySpark's extensive arsenal for data summarization and aggregation. For situations requiring more than simple frequency counts--such as calculating the mean points scored per team and position--users should turn to the more generalized `groupBy()` and `agg()` functions. These provide the flexibility to apply various aggregate functions (e.g., sum, average, maximum) to numerical columns, conditional on the groupings defined by categorical variables.

Understanding when to use the specialized `crosstab` versus the more flexible `groupBy/agg` structure is key to efficient data processing in Spark. Use `crosstab` exclusively for counting the co-occurrence of two categorical attributes. Use `groupBy/agg` when numerical summaries are required alongside the categorical breakdown, or when more than two columns need to be aggregated simultaneously.

Note: You can find the complete documentation for the PySpark **crosstab** function on the official Apache Spark API reference page.

Conclusion

The cross-tabulation feature in PySpark provides an extremely powerful and efficient method for generating contingency tables from large datasets. By utilizing the simple `df.crosstab(col1, col2)` syntax, analysts can quickly quantify the relationships between two different attributes, forming a critical foundation for statistical analysis and modeling. Mastery of this function is essential for effective data manipulation within the Spark ecosystem, ensuring that descriptive statistics are generated accurately and at scale.

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