How to Partition Your PySpark DataFrames by Multiple Columns

Authored by

stats writer

January 2, 2026

The partitionBy() method in PySpark is a cornerstone tool for defining analytical boundaries within large datasets. It allows data to be logically grouped by one or more specified columns. This operation defines separate Window Functions, meaning the data is conceptually split into segments based on the unique values or combinations of the columns provided to the method. When partitioning by multiple columns, we create a composite key that dictates the boundaries for computation. This structural segmentation is crucial because it enables data to be processed in parallel processing, significantly improving the speed and efficiency of complex analyses.

The ability to use multiple columns for partitioning means that data can be segmented based on various criteria simultaneously, such as grouping sales data by 'Region' and 'Product Category.' This granular control is indispensable for advanced data preparation, analysis, and feature engineering tasks, particularly when seeking precise patterns or running cumulative metrics within tightly defined segments. By leveraging multiple partition keys, analysts ensure that calculations are localized and consistent with complex business logic.

Core Syntax: Applying Window.partitionBy() with Iterables

To implement multi-column partitioning dynamically in PySpark, you can define the list of partitioning columns and then utilize the Python unpacking operator. This technique provides flexibility and is highly readable, especially when the number of columns might change. This approach is essential for using **Window.partitionBy()** when the columns are stored in an iterable object like a list or tuple.

from pyspark.sql.window import Window

partition_cols =

w = Window.partitionBy(*partition_cols)

This particular example passes the column names **col1** and **col2** to the **partitionBy** function. The resulting window specification, stored in variable `w`, will define partitions based on the unique combination of values found in these two columns across the DataFrame.

Note that the ***** operator is used to unpack an iterable into a function call. This is a standard Python feature that is crucial for dynamic argument passing in PySpark methods. When **partitionBy()** is called, the list `partition_cols` is expanded, effectively calling `partitionBy('col1', 'col2')`. This capability ensures that we are able to pass each of the elements in **partition_cols** without manually specifying each element individually, making the code cleaner and more maintainable.

The Role of the Asterisk (*) Operator in Function Calls

Understanding the role of the Python * operator is essential for dynamic column handling in PySpark. In the context of function calls, the single asterisk unpacks the list or tuple, treating its elements as positional arguments. Since the **partitionBy()** method is designed to accept an arbitrary number of column names as distinct string arguments, we must use unpacking when these names are stored collectively in a list.

If the * operator were omitted, **PySpark** would attempt to interpret the entire list object as the first and only column name to partition by, leading to incorrect behavior or an error, as it expects string references or `Column` objects. The unpacking operation resolves this by expanding the list into the required sequence of arguments: `'col1'`, `'col2'`.

This technique is particularly valuable when the list of columns is generated programmatically. For instance, if you filter your DataFrame columns based on a metadata schema, you can pass the resulting list directly to **partitionBy()**, ensuring maximal code flexibility and adherence to the dynamic nature of large data workflows.

Practical PySpark Example: Setting Up the Scenario

The following example shows how to use this dynamic syntax in a practical application. We will create a sample dataset and then apply multi-column partitioning to calculate a sequential ID within specific subgroups.

Suppose we have the following **PySpark DataFrame** that contains information about basketball players, including their team, position, and points scored. We start by initializing our Spark environment and defining the data.

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| 21|
| A| Forward| 22|
| A| Forward| 30|
| B| Guard| 14|
| B| Guard| 14|
| B| Forward| 13|
| B| Forward| 7|
+----+--------+------+
```

This initial dataset shows nine players spread across two teams (A and B) and two positions (Guard and Forward). Our objective is to calculate a ranking or sequential ID within each unique combination--for example, ranking only the 'Guard' players on 'Team A' separately from the 'Forward' players on 'Team B'.

Implementing Multi-Column Partitioning and Window Functions

Suppose we would like to add a new column named **id** that contains row numbers for each record in the DataFrame, but these numbers must reset every time the group defined by the **team** and **position** columns changes. This requires us to define a window specification that incorporates both columns.

To do so, we define our partitioning columns in a list and use the * operator to pass each column name to the **partitionBy** function. We then use the `row_number()` Window Function over this defined window to generate the sequential IDs.

```
from pyspark.sql.functions import row_number,lit
from pyspark.sql.window import Window
```

```
#specify columns to partition by
partition_cols =

#specify window: partitioning by the unpacked list, and ordering is required by PySpark
w = Window.partitionBy(*partition_cols).orderBy(lit('A'))

#add column called 'id' that contains row numbers
df = df.withColumn('id', row_number().over(w))

#view updated DataFrame
df.show()

+----+--------+------+---+
|team|position|points| id|
+----+--------+------+---+
|  A| Forward|    21|  1|
|  A| Forward|    22|  2|
|  A| Forward|    30|  3|
|  A|   Guard|    11|  1|
|  A|   Guard|     8|  2|
|  B| Forward|    13|  1|
|  B| Forward|     7|  2|
|  B|   Guard|    14|  1|
|  B|   Guard|    14|  2|
+----+--------+------+---+
```

The resulting DataFrame successfully contains row numbers for each row, grouped by the composite key of the **team** and **position** columns. The ID counter resets whenever a new combination (e.g., transitioning from Team A/Forward to Team A/Guard) is encountered.

Analyzing the Results and Interpreting the Partition Logic

The output clearly demonstrates the effect of multi-column partitioning. The dataset has been logically divided into four distinct partitions, based on the cross-product of the unique values in the two partitioning columns. For instance, the combination 'A' and 'Forward' constitutes a single, isolated window where the row numbering runs from 1 to 3. The subsequent partition 'A' and 'Guard' correctly starts its count again at 1, running to 2.

This strict segmentation is the powerful outcome of using partitionBy() with multiple columns. If we had only partitioned by 'team', the ID for Team A would have run continuously from 1 to 5. By adding 'position' to the partitioning criteria, we refined the boundaries, making the calculated IDs

specific to the player's role within their specific team.

Note #1: In this practical example, we passed two column names to the **partitionBy** function, but the list can include as many column names as required by your analytical needs. The use of the * operator ensures scalability regardless of the list size.

Performance Considerations and Best Practices

When applying multi-column partitioning, performance is a paramount consideration. While partitioning enables crucial parallel processing, the operation inherently requires data shuffling--moving data across the cluster network to ensure that all members of a partition reside on the same worker node before the window function calculation begins. Excessive or inefficient shuffling can degrade performance.

It is best practice to choose partitioning columns that result in a balanced distribution of data across the resulting windows. If the combination of partition columns yields a few extremely large partitions (a data skew issue), the calculation for those partitions will be bottlenecked on a few nodes. Conversely, if the partition keys result in thousands of extremely small partitions (high cardinality), the overhead of managing the sheer number of partitions can negatively impact cluster efficiency.

Note #2: You can find the complete and authoritative documentation for the PySpark **partitionBy** function on the official Apache Spark documentation website, which provides deep technical insights into its implementation and performance characteristics.

Conclusion

Mastering the use of **partitionBy()** with dynamically provided multiple columns is vital for executing complex, granular analyses in PySpark. By leveraging the Python * unpacking operator alongside a list of column names, developers can create highly precise and scalable window specifications. This dynamic approach allows for flexible code that adapts easily to changing data models, ensuring that metrics and calculations are always segmented correctly based on composite key definitions.

The implementation shown illustrates the effective application of this technique to segment data by both **team** and **position**, demonstrating how the resulting row numbers reset accurately at each new partition boundary. This capability is foundational for sequential analysis, calculating cumulative sums, and determining rankings within specific, multi-dimensional groups across massive datasets.

Further Resources

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