PySpark: Select Columns with Alias

Authored by

Mohammed looti

November 11, 2025

Introduction to Column Aliasing in PySpark

Aliasing columns is a fundamental operation when working with large-scale data processing systems like Apache Spark, particularly when utilizing the Python API, PySpark. Renaming a column--or providing an alias--is often necessary for several reasons: improving readability, ensuring compliance with downstream system requirements, or handling conflicts during data joins where source columns might share identical names. The ability to select columns and apply meaningful aliases is critical for generating clean and understandable data outputs.

In PySpark DataFrames, there are two primary approaches to achieving column aliasing, each serving a distinct purpose based on whether you intend to narrow down the result set or simply rename an existing column while keeping the entire dataset intact. Understanding the functional difference between these methods is key to efficient data manipulation in a Spark environment.

The first method involves using the `alias()` function in conjunction with the `select()` transformation. This approach is highly effective when the goal is to extract a specific subset of columns from the DataFrame and apply new names simultaneously. The second method, utilizing the `withColumnRenamed()` transformation, is typically employed when the requirement is to retain all existing columns in the DataFrame but change the header name of one or more specific columns for clarity or standardization. We will explore both techniques using practical examples to illustrate their implementation and resulting outputs.

Setting Up the PySpark Environment and Sample Data

Before diving into the methods for aliasing, we must establish a working PySpark context and create the sample PySpark DataFrame that will be used throughout our examples. This foundational step ensures repeatability and clarity. We begin by initializing the `SparkSession`, which is the entry point for using Spark functionality.

The sample data simulates a small dataset containing basketball statistics, including team identifiers, conference affiliations, points scored, and assists recorded. We define both the raw data rows and the column names explicitly before using the `createDataFrame()` method to construct the DataFrame object, which we name `df`. This structured approach allows us to demonstrate how aliasing functions interact with predefined column names.

The following code block executes the necessary imports, defines the data structure, and displays the resulting DataFrame, providing the baseline context for all subsequent column aliasing transformations.

```
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|conference|points|assists|
+----+----------+------+-------+
| A| East| 11| 4|
| A| East| 8| 9|
| A| East| 10| 3|
| B| West| 6| 12|
| B| West| 6| 4|
| C| East| 5| 2|
+----+----------+------+-------+
```

Technique 1: Selecting a Single Column and Applying an Alias (The `alias()` Method)

The first common approach involves explicitly selecting a column and immediately applying a new name using the **`alias()`** function. This method is fundamentally tied to the **`select()`** transformation, which is used to project a specified set of columns from the existing DataFrame into a new DataFrame.

When **`select()`** is used, only the columns explicitly listed within the function arguments will be present in the resulting DataFrame. Therefore, if we only select a single column (e.g., **`df.team`**) and apply an alias (e.g., **`.alias('team_name')`**), the output DataFrame will contain only that single column, now labeled with its new alias. This is the most efficient way to isolate and rename

specific data elements.

In the example below, we target the `team` column. We call the `alias` function on the column object itself, supplying the desired new name, `team_name`. This demonstrates how to return a highly focused result set where the column header has been customized for clarity.

#select 'team' column and display using aliased name of 'team_name'
df.select(df.team.alias('team_name')).show()

```
+---------+
|team_name|
+---------+
| A|
| A|
| A|
| B|
| B|
| C|
+---------+
```

As observed in the output, the resulting DataFrame contains only the data from the original `team` column, and its header is successfully displayed as `team_name`. This confirms that the `select()` method, combined with `alias()`, is perfect for projection and immediate renaming, particularly when transforming a column into a derived field or preparing a simplified view of the data.

Technique 2: Renaming a Column While Retaining All Data (The `withColumnRenamed()` Method)

The second powerful technique available in PySpark is the use of the `withColumnRenamed()` transformation. Unlike the `select()` method, which acts as a projection operator and discards unselected columns, `withColumnRenamed()` is specifically designed to perform an in-place modification of the DataFrame schema by changing only the name of a specified column while preserving all other columns and their associated data.

This function accepts two string arguments: the original column name and the desired new column name. Because it is a DataFrame-level transformation, it returns a new DataFrame instance that is identical to the original, save for the updated header of the renamed column. This is often the preferred method when preparing a DataFrame for final output or integration where the full context of the data (all columns) must be maintained, but standardization of naming conventions is required.

The following syntax demonstrates how we can rename the `team` column to `team_name` while ensuring that the `conference`, `points`, and `assists` columns remain in the resulting DataFrame, completely unchanged.

#select all columns and display 'team' column using aliased name of 'team_name'
df.withColumnRenamed('team', 'team_name').show()

```
+---------+----------+------+-------+
|team_name|conference|points|assists|
+---------+----------+------+-------+
|        A|      East|    11|      4|
|        A|      East|     8|      9|
|        A|      East|    10|      3|
|        B|      West|     6|     12|
|        B|      West|     6|      4|
|        C|      East|     5|      2|
+---------+----------+------+-------+
```

The output clearly shows that the resulting PySpark DataFrame retains its original four columns, but the header of the first column has been successfully updated to `team_name`. This confirms that `withColumnRenamed()` provides a non-destructive renaming operation ideal for preserving the full data structure.

Choosing the Right Method: Select vs. withColumnRenamed

While both the `alias()` function (used within `select()`) and the `withColumnRenamed()` method achieve the goal of changing a column's name, the choice between them hinges entirely on the desired output structure and the complexity of the transformation being performed.

The `select()` approach, coupled with `alias()`, is powerful because it allows for complex expressions and calculations to be performed while simultaneously defining the output name. For instance, if you were calculating a new column derived from two existing columns (e.g., `df.points + df.assists`) and wanted to name the result `total_stats`, you would use `(df.points + df.assists).alias('total_stats')` within your `select()` statement. However, remember that if you use `select()`, you must explicitly list every single column you wish to keep, which can be verbose for DataFrames with dozens or hundreds of columns.

Conversely, `withColumnRenamed()` is the simpler, cleaner solution for pure schema updates. Its sole purpose is renaming an existing column without affecting any other aspect of the DataFrame. If you have a massive DataFrame and only need to standardize the case or correct a typo in a

single column header, using `withColumnRenamed()` is far more efficient and readable than listing every column in a `select()` statement just to rename one element.

To summarize the use cases: use `select()` with `alias()` when you are projecting a subset of data or creating new derived columns, and use `withColumnRenamed()` when you need to rename a column while maintaining the complete set of existing columns in the output DataFrame.

Conclusion and Further Resources

Mastering column aliasing is essential for any data engineer or analyst working with PySpark. By utilizing the `alias()` function for selective projection and transformation, or the `withColumnRenamed()` method for simple schema adjustments, practitioners can ensure their DataFrames are clear, compliant, and optimized for subsequent processing stages.

The operational difference between these two methods is a key differentiator in PySpark efficiency. Remembering that `select()` implies a projection (keeping only what is listed) while `withColumnRenamed()` implies a schema modification (keeping everything and only changing the name) allows for precise control over the output data structure.

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

Additional Resources

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

How to use window functions to calculate running totals in Spark.

Understanding the difference between `map()` and `mapPartitions()` for RDD transformations.

A comprehensive guide to joining multiple PySpark DataFrames efficiently.