How to Remove Spaces from PySpark Column Names

Authored by

stats writer

February 6, 2026

Achieving optimal data hygiene is fundamental when performing large-scale data processing, especially within the PySpark environment. One of the most common requirements for cleaning structured data involves standardizing column names. Removing spaces from column names in PySpark can be accomplished through powerful built-in functions. While the `withColumnRenamed` function is suitable for individual column updates, a more robust and scalable solution for mass changes utilizes the `select` transformation combined with list comprehension and string manipulation. This technique allows data engineers to efficiently iterate through a DataFrame and ensure all columns adhere to strict naming conventions, significantly improving code readability and preventing runtime errors associated with accessing columns containing special characters or spaces.

PySpark: Efficiently Standardizing and Renaming Columns

1. Introduction: The Necessity of Clean Column Names in PySpark

In the realm of big data analytics, particularly when working with Apache Spark via its Python API, PySpark, adhering to consistent naming conventions for your data schema is not merely a cosmetic choice--it is a critical requirement for maintainability and operational stability. Column names containing spaces, such as "Customer Name" or "Total Revenue", often complicate data manipulation tasks. These spaces necessitate the use of quotation marks or backticks when referencing the column in SQL expressions or certain API calls, increasing the risk of syntax errors and making code harder to read. Standardizing column names, typically by converting spaces to underscores (snake_case) or eliminating them entirely (camelCase or flatcase), drastically simplifies development efforts.

The motivation for column renaming extends beyond just aesthetics; it directly impacts compatibility with downstream systems. Many database systems, BI tools, and serialization formats (like Parquet or Avro) have constraints on identifiers, often disallowing spaces or certain special characters. By cleaning up column names early in the data pipeline, data engineers ensure that the DataFrame remains compatible throughout its entire lifecycle, from ingestion to final reporting. This proactive approach prevents unexpected schema validation failures later on.

While individual column renaming is straightforward, handling a DataFrame with dozens or even hundreds of columns, all requiring space removal, demands an automated, programmatic solution. Relying on manual calls to renaming functions is inefficient and error-prone. The ideal solution leverages PySpark's functional programming capabilities, enabling a concise and highly scalable operation that iterates over all existing column names and applies a consistent transformation rule to each one simultaneously.

2. Initial Methods for Renaming Columns

For situations involving only a few columns, the primary mechanism provided by the PySpark DataFrame API is the **`withColumnRenamed`** function. This function takes two mandatory string arguments: the existing column name and the desired new column name. When dealing with an old name that contains spaces, it is often best practice to enclose that name in backticks (`) if using SQL expressions, but within the Python API, standard string representation usually suffices when passed directly to the function.

The syntax is simple and provides excellent clarity for targeted changes. For example, if you only needed to rename a column called "Sales Region," you would execute `df.withColumnRenamed("Sales Region", "sales_region")`. However, using **`withColumnRenamed`** sequentially for many columns quickly becomes verbose and computationally inefficient, as each call returns a new DataFrame object, potentially increasing overhead.

3. The Power of the `select` Transformation for Mass Renaming

When the goal is to systematically sanitize all column names in a DataFrame, the most efficient and standard approach involves using the **`select`** transformation alongside list comprehension. This technique allows us to generate a comprehensive list of all columns, apply a string manipulation function (like Python's built-in `replace()` method) to rename them, and execute the entire transformation in a single, optimized operation.

This approach requires importing necessary functions from `pyspark.sql`, specifically the **`functions`** module, which is typically aliased as `F`. The core of the solution is iterating through `df.columns` (which returns a list of all current column names), applying the string replacement logic, and then aliasing the original column data using **`F.col(x).alias(new_name)`**. This creates the final list of renamed column expressions necessary for the **`select`** statement.

The general syntax is concise yet powerful, providing a clear illustration of how functional programming principles integrate seamlessly into the PySpark API for advanced schema modifications. Below is the fundamental syntax used to automate the column renaming process, replacing spaces with a chosen character, such as an underscore:

from pyspark.sql import functions as F

#replace all spaces in column names with underscores
df_new = df.select()

4. Practical Example: Setting Up the PySpark DataFrame

To demonstrate this solution practically, let us first establish a sample DataFrame that deliberately incorporates column names containing spaces. This scenario simulates raw data ingestion where column headers might be derived directly from source files (like CSVs or Excel sheets) that permit spaces. We utilize a `SparkSession` to initialize the environment and create a dataset representing basketball team metrics.

In this example, we define three columns: 'team name', 'points scored', and 'total assists'. All three require standardization. The creation process confirms that the DataFrame structure reflects these non-standard names, setting the stage for the subsequent cleansing operations.

Example: How to Remove Spaces from Column Names in PySpark

Suppose we have the following PySpark DataFrame that contains information about 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 name|points scored|total assists|
+---------+-------------+-------------+
| Mavs| 18| 4|

Nets	33	9
Hawks	12	7
Thunder	15	3
Lakers	19	2
Cavs	24	5
Magic	28	7
+---------+-------------+-------------+

As clearly illustrated by the output above, each column name presently contains one or more spaces, which we must address for proper data hygiene and programmatic access. The next step will apply the automated renaming mechanism.

5. Strategy 1: Replacing Spaces with Underscores (Industry Best Practice)

The most widely accepted convention for handling spaces in data processing environments is replacing them with underscores (_). This format, known as snake_case, is highly readable and is compatible with virtually all programming languages and database technologies, including PySpark's internal processing mechanisms. Underscores act as visual separators while ensuring the resulting name remains a single, valid identifier.

We apply the generic `select` transformation previously introduced. By utilizing Python's list comprehension, we iterate through the list of column names, calling `x.replace(' ', '_')` to perform the substitution. This generates a new list of expressions where the original column data is selected but is simultaneously assigned a new, standardized alias.

Executing this code transforms the DataFrame efficiently without necessitating multiple passes or verbose sequential function calls. The resulting `df_new` DataFrame maintains all the original data values but presents a cleaned, standardized schema, which is essential for subsequent aggregation, joining, or machine learning preparation steps.

from pyspark.sql import functions as F

#replace all spaces in column names with underscores
df_new = df.select()

#view new DataFrame
df_new.show()

+-----------+-------------+-------------+
| team_name|points_scored|total_assists|
+-----------+-------------+-------------+

Mavs	18	4
Nets	33	9
Hawks	12	7
Thunder	15	3
Lakers	19	2
Cavs	24	5
Magic	28	7
+----------+-------------+-------------+

Observation of the output confirms that the spaces in the column headers have been successfully replaced by underscores, yielding standardized names like `team_name`, `points_scored`, and `total_assists`.

6. Strategy 2: Removing Spaces Entirely (When Conciseness is Key)

While snake_case is generally preferred, there are scenarios where data engineering teams might opt for simply removing the spaces entirely, resulting in what is often termed flatcase (e.g., `teamname`). This strategy maximizes conciseness but can occasionally reduce readability, especially for very long column names. The procedure for implementation remains identical to Strategy 1, only modifying the replacement string within the `replace()` method.

To achieve complete space removal, we pass an empty string (`''`) as the second argument to the `replace()` function. The list comprehension executes `x.replace(' ', '')`, effectively deleting every space encountered in the column name during the iteration. This results in tightly packed, compound column names.

This method is particularly useful when integrating with systems that impose very strict character limits on identifiers, although it should be used judiciously, prioritizing long-term code clarity over extreme brevity. The following code illustrates this alternative approach:

from pyspark.sql import functions as F

#remove all spaces in column names
df_new = df.select()

#view new DataFrame
df_new.show()

+--------+------------+------------+
|teamname|pointsscored|totalassists|
+--------+------------+------------+

Mavs	18	4
Nets	33	9
Hawks	12	7
Thunder	15	3
Lakers	19	2
Cavs	24	5
Magic	28	7
+--------+-----------+-----------+

Upon reviewing the output, it is evident that the spaces have been completely eradicated, resulting in names like `teamname` and `pointsscored`. This technique provides flexibility depending on organizational naming standards.

7. Advanced Considerations and Best Practices for Column Hygiene

The core of both renaming strategies relies on Python's built-in string **`replace`** function, which is executed within the context of the list comprehension before the PySpark operation is finalized. It is important to remember that this operation is performed lazily by PySpark until an action (like `.show()` or `.write()`) is called, ensuring optimization within the Spark execution engine.

While focusing on spaces is crucial, robust data hygiene often requires addressing other problematic characters. A superior approach for comprehensive column sanitization might involve using Python's regular expression library (`re`) to replace not just spaces, but any non-alphanumeric character (excluding the desired underscore) with a specified separator.

Furthermore, standardizing case (e.g., converting all column names to lowercase) is another critical best practice. This can be easily incorporated into the list comprehension: `x.lower().replace(' ', '_')`. Combining replacement techniques with case standardization ensures maximum compatibility and clarity, promoting consistent code across large data projects.

The functions utilized in these examples--**`F.col()`** and the `.alias()` method--are essential components of the PySpark SQL API, enabling developers to reference column objects and assign them descriptive, clean names within a single, powerful transformation. Data engineers are encouraged to leverage these tools for automated schema management as part of their initial ETL processes.

Further PySpark Documentation and Tutorials

For developers interested in exploring additional string manipulation and column management features within the framework, the official documentation for the PySpark string functions, including

`replace` and related utilities, serves as an invaluable resource.

In addition to column renaming, PySpark offers extensive capabilities for data transformation. We recommend reviewing tutorials on topics such as:

Handling missing values (Imputation).
Performing complex data joins and window functions.
Optimizing DataFrame performance through partitioning and caching.

These foundational skills ensure the creation of high-quality, production-ready data pipelines.