PySpark: Drop Multiple Columns from DataFrame

Authored by

stats writer

November 16, 2025

RECOMMENDED CITATION

stats writer (2025). *PySpark: Drop Multiple Columns from DataFrame*. PSYCHOLOGICAL SCALES. Retrieved from https://scales.arabpsychology.com/?p=92541

The Necessity of PySpark DataFrame Manipulation

In the world of big data processing, the PySpark framework, which leverages the power of Apache Spark, stands out as an indispensable tool for data engineers and data scientists alike. When dealing with massive datasets, effective data preparation is paramount. This often involves cleaning, transforming, and importantly, reducing the dimensionality of the data by removing irrelevant or redundant columns. The process of dropping columns in a DataFrame is a fundamental operation in this preparation phase. Efficiently removing multiple columns simultaneously is crucial for maintaining optimal performance, especially when working with production-scale distributed data environments.

While dropping a single column is straightforward using the built-in `drop()` function provided by the PySpark DataFrame API, handling multiple columns requires slightly more sophisticated techniques. This specialized need arises frequently during feature selection or when certain calculated fields are no longer needed after a transformation step. Selecting the appropriate method--whether listing columns directly or using a dynamic list structure--depends heavily on the context, the number of columns involved, and the maintainability requirements of the code base. Understanding the nuances of these approaches allows developers to write robust and scalable data pipelines.

This guide delves into the two primary, highly efficient methods available in PySpark for removing multiple columns from a DataFrame. We will explore the syntax, provide concrete examples utilizing a sample dataset, and discuss the practical considerations that might lead a user to favor one method over the other. Mastery of these column manipulation techniques ensures cleaner data models and faster execution times for subsequent analytical tasks. The goal is to maximize the readability of your code while ensuring that data engineering best practices are rigorously followed throughout the data cleaning process.

Overview of PySpark Column Dropping Methods

There are two highly effective and common ways to drop multiple columns in a PySpark DataFrame. The choice between these methods usually hinges on whether the list of columns is static or generated dynamically.

Method 1: Direct Column Specification. This involves passing each column name as a separate string argument to the `df.drop()` function. This method is concise and excellent for a small, fixed set of columns.
Method 2: List Unpacking. This technique utilizes a Python list to hold the column names and employs the asterisk (*) operator to unpack that list into the `df.drop()` function. This is preferred for dynamic or large sets of columns.

The following examples first illustrate the core syntax for each technique using placeholder column names (`team` and `points`).

Method 1 Syntax: Drop Multiple Columns by Name

This is the simplest approach, providing column names directly to the function call.

#drop 'team' and 'points' columns
df.drop('team', 'points').show()

Method 2 Syntax: Drop Multiple Columns Based on List Unpacking

This method defines the columns in a list variable and uses the unpacking operator (`*`) before passing it to the `drop()` function.

#define list of columns to drop
drop_cols =

#drop all columns in list
df.drop(*drop_cols).show()

Establishing the Working DataFrame

To demonstrate these practical approaches, we will first establish a sample PySpark DataFrame. This DataFrame simulates a small dataset of sports statistics, containing four columns: `team`, `conference`, `points`, and `assists`. Establishing a clear, reproducible dataset is essential for validating the effectiveness of the column dropping operations. We initialize the Spark environment using `SparkSession.builder.getOrCreate()`, which is the standard entry point for all Spark functionality.

The subsequent steps involve defining the data as a list of rows and defining the header names, which are then passed to the `createDataFrame` method. This transformation converts the native Python data structure into a distributed, schema-aware PySpark object. The initial visualization using `df.show()` confirms the structure before any modification, allowing us to clearly track the changes introduced by the `drop()` operations. Our aim in the following examples is to eliminate the `team` and `points` columns.

The following examples show how to use each method in practice with the following PySpark DataFrame:

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|
+----+----------+------+-------+
```


Example 1: Dropping Columns via Direct Specification

The direct specification method is highly effective due to its simplicity and immediate clarity. When the columns to be removed are known constants, passing them directly to the `drop()` function provides the most concise syntax. This approach avoids the need for intermediate variable declarations, streamlining the code, which is particularly beneficial for small-scale data preparation tasks or interactive exploration sessions. We utilize the DataFrame's built-in transformation capability to return a new DataFrame excluding the specified fields.

In this example, we want to permanently remove `team` and `points` from our analysis. We simply call `df.drop('team', 'points')`. It is essential to remember that since `drop()` is a

transformation, it must be followed by an action, such as `show()`, to execute the command and return the visible result. This method relies entirely on the positional arguments matching the exact string names of the columns in the DataFrame schema.

While straightforward, developers must ensure that column names are spelled correctly and match case, as misspellings will typically be ignored by the function without raising an explicit error, potentially leading to retained columns where removal was intended. This simplicity, however, makes it perfect for quick cleaning where the schema knowledge is high and volatility is low.

Example 1: Drop Multiple Columns by Name

We can use the following syntax to drop the **team** and **points** columns from the DataFrame:

#drop 'team' and 'points' columns
df.drop('team', 'points').show()

+----------+-------+
|conference|assists|
+----------+-------+
East	4
East	9
East	3
West	12
West	4
East	2
+----------+-------+

Notice that the **team** and **points** columns have both been successfully dropped from the DataFrame, resulting in a cleaner schema containing only the `conference` and `assists` fields, just as we specified in the direct positional arguments.

Example 2: Dynamic Dropping Using Python Lists and Unpacking

For scenarios involving dynamic schema changes, configuration-driven pipelines, or when dealing with a large quantity of columns, the list-based approach offers superior flexibility and readability. By first defining the columns to be dropped within a Python list, we separate the configuration step from the execution step. This modularity is a hallmark of robust software engineering and highly recommended for production ETL jobs.

The core feature enabling this technique is the use of the Python unpacking operator (`*`). When a

list is preceded by `*` inside a function call, Python treats the elements of the list as individual arguments passed to that function. Thus, if `drop_cols` contains , then `df.drop(*drop_cols)` is functionally equivalent to `df.drop('team', 'points')`. This abstraction allows the list to be populated dynamically, perhaps by querying metadata or applying filtering logic to the DataFrame's schema.

This method is invaluable when performing advanced data cleaning tasks, such as removing all columns identified as highly correlated or those marked for archival based on a configuration file. It ensures the main DataFrame manipulation logic remains clean, regardless of how many columns are eventually selected for removal. The following code demonstrates the implementation of this powerful and scalable technique.

Example 2: Drop Multiple Columns Based on List

We can use the following syntax to specify a list of column names and then drop all columns in the DataFrame that belong to the list by utilizing the unpacking operator:

#define list of columns to drop
drop_cols =

#drop all columns in list
df.drop(*drop_cols).show()

+----------+-------+
|conference|assists|
+----------+-------+
East	4
East	9
East	3
West	12
West	4
East	2
+----------+-------+

Notice that the resulting DataFrame drops each of the column names that we specified in the list `drop_cols`, yielding the exact same result as the direct specification method, but offering greater flexibility for future adjustments.

Choosing the Right Method for PySpark Data Cleanup

Selecting between the two methods for dropping multiple columns should be an informed decision

based on the complexity, scale, and expected evolution of the data pipeline. While Method 1 provides immediate coding brevity, Method 2 offers long-term maintenance advantages essential for production environments utilizing Apache Spark.

Use the **Direct Specification (Method 1)** when:

You are performing interactive analysis or rapid prototyping.
The number of columns to drop is small (typically 1 to 5).
The list of columns is static and hardcoded within the script.

The benefit here is simplicity, as it avoids creating an extra variable, making the operation immediately transparent on a single line.

Use the **List Unpacking (Method 2)** when:

The list of columns is large, making a single line function call unwieldy.
The columns are generated dynamically (e.g., based on schema introspection or data quality checks).
You need to separate configuration data (the list of columns) from execution logic for better modularity and maintainability.

The initial cost of defining the Python list is quickly outweighed by the gains in code clarity and adaptability, particularly in team development environments or large-scale ETL jobs.