How to Drop Multiple Columns from a PySpark DataFrame Easily

Authored by

stats writer

February 9, 2026

In the realm of big data processing, efficient manipulation of large datasets is paramount. When working with PySpark, removing unnecessary columns from a DataFrame is a fundamental preprocessing step crucial for optimizing memory usage and speeding up subsequent analytical operations. Fortunately, PySpark offers a straightforward and powerful mechanism to accomplish this: the built-in `drop()` transformation.

The drop method is designed to efficiently handle the removal of one or more columns simultaneously. This operation is non-mutating, meaning it does not modify the original DataFrame but rather returns an entirely new DataFrame containing only the remaining columns. Understanding how to leverage this method, particularly when dealing with multiple columns, is essential for any data engineer or data scientist utilizing the PySpark framework for scalable data analysis and ETL processes.

The Power of the PySpark drop() Transformation

The drop method in PySpark is highly flexible, accepting either a single column name as a string or multiple column names specified in various formats. When cleaning data, it is far more common to remove several columns at once, especially those deemed irrelevant, redundant, or sensitive, which is why optimizing this multi-column drop operation is critical for maintaining workflow efficiency.

We will examine two primary, widely-used methods for removing a collection of columns. Both approaches utilize the same core `drop()` function but differ slightly in how the input arguments--the column names--are passed to the function, catering to different scripting needs and readability preferences. These methods ensure that whether you know the columns beforehand or dynamically generate a list of columns to exclude, the process remains streamlined and declarative.

The following common scenarios cover nearly all requirements for column exclusion in large-scale data manipulation:

Direct Specification: Providing the names of columns to drop as separate string arguments to the method.
List Utilization: Defining the columns to drop as a Python list and passing that list using the Python splat operator (`*`).

Setting Up the PySpark Environment and Sample Data

Before demonstrating the two methods for dropping columns, we must first establish a functional DataFrame. This involves initializing a SparkSession, defining the sample data structure, and

creating the DataFrame itself. This setup is crucial for ensuring the subsequent code examples execute correctly and display the expected output.

The following code snippet demonstrates the standard procedure for creating a simple DataFrame named `df`, which we will use throughout this article to illustrate the column dropping mechanics. Our sample DataFrame contains information about teams, conferences, and performance metrics (points and assists). This initial step simulates real-world data loading and is necessary for demonstrating the column 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|
+----+----------+------+-------+
```

As seen in the output, the resulting DataFrame `df` contains four distinct columns: `team`,

`conference`, `points`, and `assists`. Our goal in the subsequent examples will be to remove the `team` and `points` columns, leaving only the contextual `conference` and metric `assists` fields for targeted data analysis.

Method 1: Dropping Multiple Columns by Direct Specification

The most straightforward way to use the drop method when dealing with a small, fixed number of columns is to pass their names directly as sequential string arguments. This syntax is concise and highly readable, particularly when you only need to exclude two or three columns whose names are known at the time of coding. This method capitalizes on the flexibility of Python function arguments.

When you call `df.drop()`, every string argument provided after the DataFrame object refers to a column that PySpark should exclude from the resulting transformation. This technique avoids the intermediate step of creating a separate list object, making the script slightly cleaner for quick tasks. It is important to remember that, as with all DataFrame transformations, this operation returns a new object, which we typically chain with `.show()` for immediate verification or assign to a new variable for subsequent use.

Below is the syntax illustration for this method:

#drop 'team' and 'points' columnsdf.drop('team', 'points').show()

Example 1: Implementation of Direct Column Dropping

Applying the direct specification method to our sample DataFrame `df`, we specifically target the removal of the **team** and **points** columns. The resulting DataFrame, shown below, clearly indicates the success of the operation, as these two columns have been successfully excluded, leaving only the desired features.

#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 how the output confirms that the drop method seamlessly handles multiple arguments, treating each string input as a column identifier to be removed. This method is ideal when the list of columns is short and static, prioritizing immediate clarity over structural flexibility.

Method 2: Utilizing a List of Column Names

When the number of columns to drop grows large, or when the list of columns is generated dynamically (perhaps through a schema inspection or an external configuration file), passing them individually becomes cumbersome and error-prone. In such scenarios, the preferred and more scalable approach is to first define the column names within a Python list and then unpack that list into the `drop()` function using the asterisk (splat) operator (`*`).

This technique separates the definition of the columns to be dropped from the execution of the dropping function, greatly improving code organization and maintainability. By utilizing the splat operator, Python internally expands the contents of the list, passing each element as a distinct, positional argument to the `drop()` method, effectively mimicking the direct specification approach (Method 1) but with superior flexibility required for automated data pipelines.

Here is the syntax illustration for defining and using a drop list:

#define list of columns to drop
drop_cols =

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

Example 2: Implementation Using a Defined List

We define the list `drop_cols` containing the names `'team'` and `'points'`. We then pass this list to the `drop()` transformation, ensuring the asterisk `*` precedes the list name. This is a critical syntactical requirement for unpacking the iterable object into individual arguments expected by the function signature. This method is structurally superior for production environments where lists of columns might frequently change based on external input or complex logic.

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

As demonstrated by the output, the resulting DataFrame is identical to the one produced by Method 1. The list-based approach is functionally equivalent but provides superior utility when dealing with dynamic column removal, such as when cleaning data based on criteria like column data types, null counts, or low variance thresholds.

Why Choose the List Approach? Best Practices for Data Pipelines

While both methods achieve the same end result, the list-based approach (Method 2) is highly recommended for professional PySpark development. This recommendation stems from several key benefits related to code maintenance, flexibility, and dynamic execution in complex, large-scale data pipelines.

Firstly, defining the columns separately enhances readability and modularity. If you are dropping a large number of columns, listing them all within the `drop()` function call makes the line excessively long and difficult to parse. Separating this list allows developers to quickly see which fields are being excluded without cluttering the transformation logic. Secondly, the list structure allows for programmatic generation. For instance, if you need to drop all columns matching a certain regular expression or all columns that are identified as system metadata (e.g., columns ending in `_id` or `_ts`), you can generate a list of names first, and then simply pass that variable to `df.drop()`. This dynamic capability is indispensable in robust ETL workflows where schemas evolve.

The list structure also simplifies debugging and logging. You can easily print or log the contents of the `drop_cols` variable to verify exactly what columns are being targeted for removal before the potentially resource-intensive PySpark transformation occurs. This practice promotes transparency and reliability in data processing scripts, offering a clear checkpoint for validation.

Understanding Immutability and Performance in PySpark

It is critical to reinforce the concept of immutability within the Apache Spark ecosystem, especially when performing transformations like `drop()`. When you execute `df.drop(...)`, the existing DataFrame `df` remains untouched. Instead, Spark computes and returns a completely new DataFrame object with the requested columns excluded. This principle is fundamental to Spark's fault tolerance, distributed processing model, and guarantees that parallel operations do not accidentally corrupt shared data states.

From a performance perspective, dropping columns is generally a highly optimized operation in PySpark. When the `drop()` method is called, Spark updates the internal metadata (the schema) of the new DataFrame, marking the specified columns as absent. The actual removal of the data from memory (or disk) occurs efficiently as part of the execution plan when an action (like `.show()`, `.count()`, or `.write()`) is subsequently triggered. This lazy evaluation ensures that resources are only expended when necessary, optimizing the overall pipeline execution time without unnecessary data shuffling.

A final consideration involves using `df.select()` as an alternative when the number of columns to keep is much smaller than the number of columns to drop. While `drop()` defines exclusions, `select()` defines inclusions. For instance, if a DataFrame has 100 columns and you only need 5, using `df.select('col1', 'col2', ...)` is often cleaner and less prone to errors than listing 95 columns to drop. Developers should choose the method that minimizes the length and complexity of the required column list.

Summary of Column Dropping Techniques

The ability to accurately and efficiently remove multiple columns is a cornerstone of effective data preparation in PySpark. By leveraging the versatility of the `drop()` method, users can choose between two robust implementation styles tailored to their specific use case--be it a quick removal of known columns or a sophisticated, dynamic exclusion driven by programmatic logic.

Key takeaways regarding the two methods reviewed:

Direct Specification (Method 1): Best used for quick, ad-hoc cleaning where the number of columns to drop is small (typically two or three). It is concise and highly readable for simple tasks, avoiding intermediate list definitions.
List Utilization (Method 2): The recommended standard for production-level ETL pipelines, large-scale column removal, or when the column names are generated dynamically. It uses the Python splat operator (`*`) for unpacking the list into arguments, offering superior flexibility, maintainability, and compatibility with dynamic schema operations.

Mastering these techniques ensures that your data preparation phase is optimized, leading to cleaner data sets and faster analytical processing down the line.

Further PySpark Exploration

To further enhance your mastery of PySpark data manipulation, consider exploring tutorials on related tasks that are frequently encountered during the data cleaning and preparation phases. Efficiently handling columns often goes hand-in-hand with renaming, filtering, and joining operations. Developing proficiency in these areas will solidify your foundation in scalable big data processing.

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

How to filter rows based on specific conditions using the `filter()` or `where()` methods.
Best practices for renaming columns in a large DataFrame using the `withColumnRenamed()` transformation.
Techniques for joining multiple DataFrames efficiently using various join types like inner, left, and right joins.