PySpark: Drop Multiple Columns from DataFrame

Authored by

Mohammed looti

November 11, 2025

Understanding Column Management in PySpark

The ability to efficiently manage the schema of a PySpark DataFrame is a foundational skill in modern data engineering and analysis. During the typical ETL (Extract, Transform, Load) process, data often arrives with numerous columns that are either redundant, contain sensitive information, or are simply not relevant to the current analytical task. Removing these extraneous columns is crucial for optimizing performance, reducing memory overhead, and improving the overall readability of the resulting dataset.

When working within the Apache Spark ecosystem, especially using its Python API, PySpark DataFrames are immutable. This means that when you drop columns, you are not modifying the original DataFrame in place; rather, you are creating a new DataFrame that contains the desired subset of columns. Understanding this concept is key to effective data manipulation.

Fortunately, PySpark offers highly flexible and intuitive methods for column removal. For scenarios requiring the removal of multiple columns simultaneously, two robust techniques stand out. The choice between these methods usually depends on the dynamism of the column names--whether they are hardcoded or generated from an external list or condition.

Prerequisites and Setup of the Example DataFrame

Before diving into the specific methods for dropping columns, we must first establish a working environment and create the sample PySpark DataFrame that will be used throughout our examples. All PySpark operations require an active SparkSession, which acts as the entry point to communicate with the Spark cluster.

The following code snippet demonstrates the necessary imports and the creation of a sample DataFrame representing hypothetical sports statistics. This setup ensures that all subsequent code examples are immediately executable and verifiable. We define the data structure, assign meaningful column names, and then display the resulting DataFrame using the `.show()` action.

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 shown above, our initial DataFrame, named `df`, contains four columns: `team`, `conference`, `points`, and `assists`. For the purposes of our demonstration, we will proceed with the common task of dropping the `team` and `points` columns, retaining only the geographical (`conference`) and auxiliary statistical (`assists`) data.

Method 1: Explicitly Dropping Columns by Name (The `drop()` Function)

The most straightforward approach for removing columns in PySpark utilizes the built-in `.drop()` method available on the DataFrame object. This method is highly effective when you know the exact names of the columns you wish to eliminate beforehand. It accepts column names as sequential string arguments.

To drop multiple columns using this method, you simply pass each column name as a separate argument to the `.drop()` function. This technique is often preferred for rapid development or when dealing with a small, fixed set of columns that need pruning.

The core benefit of this syntax is its clarity and conciseness. When reading the code, it is immediately apparent which columns are being targeted for removal.

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

Executing this command produces a new DataFrame where the specified columns have been successfully removed. This is demonstrated below using our sample data:

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

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

As illustrated by the output, only the `conference` and `assists` columns remain, confirming that both `team` and `points` were accurately dropped from the resulting DataFrame schema.

Method 2: Leveraging Lists for Dynamic Column Removal

While Method 1 is excellent for hardcoded column removal, real-world data processing often requires a more flexible approach. In large-scale data manipulation scenarios, the list of columns to drop might be generated dynamically--perhaps identified through schema analysis, filtering based on data types, or read from a configuration file. For these situations, defining a Python list containing all column names and passing that list to the `.drop()` method is the superior strategy.

This technique requires the use of the Python asterisk operator (`*`), also known as the unpacking operator. When placed before the list variable inside the function call, the asterisk unpacks the list elements, treating each element as a separate argument to the `.drop()` function. This mimics the behavior of Method 1 but allows for programmatic definition of the target columns.

Initially, we define the list of columns we intend to remove.

#define list of columns to drop
drop_cols =

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

While the initial code above demonstrates defining the list, the correct application of the list to the `.drop()` function--which is the method required for removal--is shown in the detailed example below. This approach ensures scalability and maintainability, especially when the number of columns to be dropped is large or variable.

#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
+----------+-------+

The resulting PySpark DataFrame is identical to the output from Method 1. The key difference lies in the input mechanism: instead of passing individual strings, we passed a list that was dynamically unpacked by Python, making this method highly adaptable for complex transformation pipelines, such as those found in robust ETL workflows.

Why Effective Column Management is Crucial for Data Efficiency

The practice of removing unnecessary columns is not merely a matter of tidiness; it has significant implications for the performance and resource consumption of Apache Spark jobs. Spark operates on distributed memory, and columns that are not needed still consume space and contribute to the I/O costs associated with reading and writing data across the cluster nodes.

By proactively dropping columns early in the processing chain, particularly in large-scale operations, data scientists and engineers can achieve several key benefits. First, smaller DataFrames require less memory (RAM) for caching and processing, reducing the chance of memory overflow errors and speeding up iterative tasks. Second, when Spark performs shuffle operations--which involve moving data between executors--fewer data points need to be serialized and transmitted across the network, leading to substantial performance gains.

Furthermore, maintaining a clean and concise schema improves data governance and readability. Complex DataFrames with hundreds of columns can obfuscate the true purpose of the data structure. Effective column removal ensures that downstream analysis and machine learning models are trained only on relevant features, minimizing noise and improving model interpretability. Whether using the explicit naming method or the dynamic list method, the goal remains the same: efficient, scalable, and focused data manipulation.

Conclusion and Further Resources

Dropping multiple columns from a PySpark DataFrame can be achieved reliably using the native `.drop()` function. The selection between Method 1 (explicit naming) and Method 2 (list unpacking) should be guided by the complexity and dynamism of the transformation requirements. For static, simple removals, explicit naming is clean and quick. For programmatic, complex transformations, leveraging the list and the unpacking operator offers the necessary flexibility and robustness required in production ETL environments.

Mastering these fundamental DataFrame operations is essential for anyone working with distributed data processing on Apache Spark. We recommend exploring the official documentation for further advanced column manipulation techniques.

Additional Resources

The following resources provide excellent tutorials and documentation that explain how to perform other common tasks in PySpark and expand upon the core concepts discussed here:

Official SparkSession Documentation

Advanced Column Selection and Filtering Techniques

Optimizing PySpark Performance