How to Drop Columns in PySpark DataFrames

Authored by

stats writer

January 20, 2026

The Necessity of Column Exclusion in Data Engineering

The process of data preparation is fundamental to any robust data analysis or machine learning pipeline. When working with large-scale datasets managed by PySpark, it is frequently necessary to curate the data by eliminating variables that are irrelevant, redundant, or contain excessive missing values. This step, often referred to as column exclusion or dropping columns, is essential for optimizing memory usage, streamlining subsequent transformations, and enhancing model performance by reducing noise.

In the DataFrame API, PySpark provides an exceptionally efficient and straightforward method for performing this operation: the built-in `.drop()` method. This transformation is highly optimized within the Apache Spark ecosystem, ensuring that operations on vast datasets are executed rapidly and distributed across the cluster nodes. Understanding how to correctly apply this method--whether for a single field or a group of fields--is a core skill for any data professional utilizing this powerful framework.

This guide will thoroughly detail the application of the `.drop()` method within PySpark, covering both the exclusion of individual columns and the simultaneous removal of multiple columns. We will provide clear syntax examples and practical demonstrations to solidify the understanding of these critical DataFrame manipulation techniques.

Understanding the `drop` Transformation in PySpark

The primary mechanism for removing columns from a DataFrame in PySpark is the PySpark **drop** function, which is available directly on the DataFrame object. It is crucial to remember that DataFrames in Apache Spark are immutable. This means that invoking `df.drop()` does not modify the original DataFrame (`df`); instead, it returns a brand-new DataFrame that contains the results of the transformation (i.e., the original data minus the specified columns).

The `.drop()` method accepts column names as string arguments. Its flexibility allows it to handle various exclusion scenarios seamlessly. When passed a single string, it identifies and removes that specific column. When passed multiple string arguments, it processes all of them simultaneously, resulting in a cleaner and more concise execution plan compared to chaining multiple operations. This method is the preferred standard for column removal due to its readability and optimized performance characteristics within the distributed computing environment.

Before executing any drop operation, developers often inspect the schema of the source DataFrame (using `df.printSchema()`) to confirm the exact spelling and case sensitivity of the column names, as PySpark column references are strictly case-sensitive. Using the correct name ensures the successful exclusion of the desired fields and prevents unexpected errors during the transformation process.

Syntax for Excluding a Single Column

Excluding an individual column is the simplest use case for the PySpark **drop** function. The syntax requires passing the exact name of the column to be removed as a single string argument to the method call. The resulting DataFrame, assigned to a new variable, will possess all the data and structure of the original, save for the specified column.

If the original DataFrame is named `df`, and the column slated for removal is identified as 'points', the operation is written straightforwardly. This process is common when performing early stage data cleaning where a single feature is deemed irrelevant or non-contributory to the analytical goals. Below illustrates the precise syntax used for this single exclusion:

#select all columns except 'points' column
df_new = df.drop('points')

This transformation generates the new DataFrame, `df_new`, which effectively represents the cleansed dataset ready for further processing steps, such as filtering, aggregation, or model training. This method ensures immutability is respected, preserving the integrity of the original source data structure.

Syntax for Excluding Multiple Columns Concurrently

When preparing complex datasets, it is often necessary to remove several columns simultaneously. The PySpark **drop** function is designed to accommodate this requirement by accepting multiple string arguments, each corresponding to a column name to be excluded. This capability makes the code concise and improves performance by consolidating multiple removal operations into a single logical step within the Apache Spark execution plan.

To exclude two or more columns, simply list their names, separated by commas, within the parentheses of the `.drop()` method. For example, if we need to eliminate both the 'conference' and 'points' columns from the original DataFrame `df`, the execution syntax is structured as follows:

#select all columns except 'conference' and 'points' columns
df_new = df.drop('conference', 'points')

It is important to ensure that every column name provided exists within the source DataFrame. While PySpark is robust, attempting to drop a non-existent column might lead to unexpected behavior or raise an analysis exception, depending on the Spark version and configuration. Always verify column names for accuracy, especially when dealing with large schemas where typographical errors are common.

Setting Up the Illustrative PySpark Environment

To effectively demonstrate the column exclusion techniques, we must first establish a working DataFrame. This involves initializing a **SparkSession**--the primary entry point for using Apache Spark functionality--and then creating the sample dataset. Our sample data simulates a simple sports record dataset containing team statistics, which is ideal for showcasing how specific fields can be selectively removed during data processing.

The dataset includes fields for 'team', 'conference', 'points', and 'assists'. We define both the data rows and the column names explicitly before using the `spark.createDataFrame()` method to materialize the distributed structure. This foundational step ensures that all subsequent code examples operate on a standardized, known starting point, allowing us to clearly observe the effects of the `.drop()` transformations.

The following code snippet details the creation, population, and initial display of the PySpark DataFrame used throughout the subsequent examples. This setup is typical for localized development and testing environments where data manipulation logic is being verified prior to deployment on a larger cluster:

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
+----+----------+------+-------+

This starting DataFrame, `df`, provides a visual baseline. It contains four columns that we will systematically target for removal in the following practical demonstrations, confirming the efficacy and correctness of the exclusion methods.

Example 1: Dropping a Single Column in PySpark

Practical Demonstration 1: Dropping a Single Column

In this first practical application, we focus on removing only one variable from our existing DataFrame, `df`. Assuming the 'points' column is deemed unnecessary for a particular analysis--perhaps because a derived metric like 'score_ratio' is preferred--we utilize the single-argument syntax of the `.drop()` method. This action generates a new DataFrame structure that excludes all references to the 'points' data while retaining the 'team', 'conference', and 'assists' fields.

This technique is fundamental in feature engineering workflows where initial data exploration identifies extraneous features that do not contribute positively to the desired outcome. The process is computationally light and highly scalable, making it suitable for even the largest datasets managed by Apache Spark. We assign the result to `df_new` to maintain the conventional separation between the source and transformed data structures, aligning with the principles of immutability.

The code below executes the transformation and then immediately displays the schema and content of the resulting DataFrame, allowing for a clear verification that the exclusion was successful. Observe how the output structure confirms the absence of the previously existing 'points' column:

#select all columns except 'points' column
df_new = df.drop('points')

#view new DataFrame
df_new.show()

```
+----+----------+-------+
|team|conference|assists|
+----+----------+-------+
| A| East| 4|
| A| East| 9|
| A| East| 3|
| B| West| 12|
| B| West| 4|
| C| East| 2|
+----+----------+-------+
```

Upon reviewing the output, it is evident that the new DataFrame, `df_new`, successfully retains the 'team', 'conference', and 'assists' columns but excludes the **points** column, confirming the correct implementation of the single-column drop function syntax.

Example 2: Dropping Multiple Columns in PySpark

Practical Demonstration 2: Dropping Multiple Columns

Building upon the previous example, this demonstration addresses the common requirement of excluding several columns simultaneously. For instance, if both 'conference' (perhaps deemed too granular) and 'points' (already excluded in a previous hypothetical scenario) are required to be removed before final analysis, we simply list them in the `.drop()` method. This multiple-argument approach is highly efficient because it allows PySpark to optimize the column projection in one go, rather than performing sequential drops that might incur unnecessary overhead.

The ability to pass multiple string arguments is one of the key features of the drop function, simplifying data pipeline code significantly. Instead of chaining `df.drop('col1').drop('col2')`, which is less idiomatic and potentially less efficient, we use the preferred `df.drop('col1', 'col2')` syntax. This ensures the intent is clear and the execution is streamlined across the distributed cluster.

Here is the implementation targeting the exclusion of both the **conference** and **points** columns. The remaining columns in the resulting DataFrame, `df_new`, will be 'team' and 'assists'. We use the same sample data setup established earlier to ensure consistency and clarity in the transformation:

```
#select all columns except 'conference' and 'points' columns
df_new = df.drop('conference', 'points')

#view new DataFrame
```

```
df_new.show()

+----+-------+
|team|assists|
+----+-------+
| A| 4|
| A| 9|
| A| 3|
| B| 12|
| B| 4|
| C| 2|
+----+-------++
```

The resulting output confirms that the DataFrame now consists solely of the 'team' and 'assists' columns. Both the **conference** and **points** fields were successfully excluded in a single, atomic operation, demonstrating the efficiency and utility of passing multiple arguments to the `.drop()` method. This flexibility is vital when dealing with wide datasets containing hundreds of features.

Considerations and Best Practices for DataFrame Manipulation

While the `.drop()` function offers a straightforward way to exclude columns, data engineers should maintain several best practices when manipulating DataFrames in PySpark. Foremost among these is the strict adherence to column naming conventions and case sensitivity, as mismatches will result in the column not being dropped without necessarily raising a critical error, leading to unexpected data outcomes downstream. Always verify the schema using `df.printSchema()` before and after major transformations.

Furthermore, understanding that DataFrames are immutable is key. Always assign the result of the `.drop()` transformation to a new variable (e.g., `df_new`) or reassign it to the original variable (e.g., `df = df.drop(...)`) if the original structure is no longer needed. Failure to assign the result will mean the transformation is executed but the original DataFrame remains unchanged, potentially causing confusion and logic errors in complex pipelines.

For specialized scenarios, especially when dealing with nested structures or complex data types like structs, the PySpark **drop** function may be combined with other methods like `.select()` to achieve highly customized column selection and exclusion logic, though `.drop()` remains the standard for simple column removal. For comprehensive details on all parameters and advanced usage, consulting the official documentation is highly recommended.

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