Learning PySpark: Excluding Columns from DataFrames with Examples

Authored by

Mohammed looti

November 11, 2025

Introduction to Excluding Columns in PySpark DataFrames

When working with large datasets, optimizing performance and focusing on relevant features is critical. In the context of big data processing using PySpark, selectively removing unnecessary columns from a DataFrame is a fundamental data preparation step. Excluding columns helps reduce memory footprint, speeds up subsequent transformations, and streamlines the data analysis pipeline by focusing only on variables essential for the task at hand. The primary and most straightforward mechanism for achieving column exclusion in PySpark leverages the built-in **drop()** function, which operates directly on the DataFrame schema.

The **drop()** function is exceptionally versatile and designed to handle both single and multiple column exclusions efficiently within the distributed computing environment of Apache Spark. Unlike some operations that require explicit selection of all desired columns, **drop()** allows developers to specify only the columns they wish to discard, making the code cleaner and more maintainable, especially when dealing with DataFrames containing dozens or hundreds of features. Understanding how to apply this function correctly is vital for any data engineer or data scientist utilizing PySpark for enterprise-level data manipulation.

We will explore two primary approaches for utilizing this powerful function, covering scenarios ranging from removing a single extraneous column to excluding a batch of irrelevant features simultaneously. These methods are foundational components of effective data cleaning and preparation workflows in Spark.

You can use the following methods to exclude specific columns in a PySpark DataFrame:

Method 1: Exclude One Column

This approach involves passing the name of the column you wish to remove as a single string argument to the **drop()** method. PySpark DataFrames are immutable, meaning this operation does not modify the original DataFrame (`df`) but instead returns a new DataFrame (`df_new`) that contains all columns except the one specified.

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

Method 2: Exclude Multiple Columns

To exclude several columns in a single, efficient operation, the **drop()** function can accept multiple string arguments. This is often the preferred method when dealing with feature engineering processes where multiple temporary or redundant fields need to be removed before final analysis.

Simply list the columns to be dropped, separated by commas, directly within the function call.

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

Setting Up the PySpark Environment and Sample Data

Before diving into the column exclusion examples, it is essential to establish a working PySpark environment and define the sample data structure we will be manipulating. All PySpark operations require an active SparkSession, which serves as the entry point to the underlying Spark functionality. Once the session is initialized, we define our raw data and the corresponding column names, allowing us to construct a robust sample DataFrame for demonstration.

The following setup code initializes the Spark context and creates a simple DataFrame containing simulated sports team statistics. This DataFrame, named `df`, includes four columns: `team`, `conference`, `points`, and `assists`. This initial structure provides a clear starting point from which we can demonstrate the selective removal of columns using the **drop()** function in the subsequent examples. We recommend running this initialization block first to ensure all examples execute correctly.

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: Removing a Single Column Using the drop() Function

In many scenarios, you may determine that only one specific column is redundant, calculated elsewhere, or simply not required for the downstream analysis. For instance, if our focus is purely on team distribution and assists, the `points` column might be considered irrelevant. Using the **drop()** function with a single string argument provides the simplest way to execute this targeted exclusion. This operation is highly optimized within Spark, as it primarily involves modifying the DataFrame's metadata (schema) rather than performing intensive data shuffling.

By assigning the result of the `df.drop('points')` operation to a new variable, `df_new`, we ensure that the original `df` remains untouched, maintaining the principle of immutability crucial to reliable distributed processing. This practice allows for easy debugging and comparison between the original dataset and the transformed result.

We can use the following syntax to select all columns in the DataFrame, excluding the **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
+----+----------+-------+

Notice that all columns in the resulting DataFrame are selected except for the **points** column. The output clearly demonstrates the successful structural change, confirming that the function performed the intended removal.

Example 2: Excluding Multiple Columns Efficiently

When the list of columns to be excluded grows, passing multiple arguments to the **drop()** function becomes the most efficient approach. This is particularly useful during the cleaning phase of a project where numerous columns (e.g., legacy IDs, highly sparse features) are identified for removal. Instead of chaining multiple **drop()** calls, which can sometimes be less readable, the PySpark implementation allows for the direct listing of all column names within a single invocation.

In this example, we aim to remove both the `conference` and `points` columns simultaneously. This leaves us with a simplified DataFrame focusing only on the `team` identifier and the `assists` statistic. This level of targeted exclusion is powerful for creating specialized subsets of data tailored to specific analytical models or reporting requirements.

We can use the following syntax to select all columns in the DataFrame, excluding the **conference** and **points** column:

#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

```
+----+-------++
```

Notice that all columns in the resulting DataFrame are selected except for the **conference** and **points** columns. The streamlined output, containing only `team` and `assists`, confirms the successful execution of the multi-column exclusion command. This technique scales seamlessly, regardless of whether you are dropping two columns or twenty.

Understanding the PySpark drop() Function

The drop function in PySpark is a method of the DataFrame class designed specifically for schema manipulation. It is critical to understand that this function is optimized for distributed computation. When PySpark executes **drop()**, it primarily updates the internal structural definition of the dataset without necessarily triggering a full data read or expensive shuffling operation across the cluster nodes, provided the operation is not part of a larger chain that requires immediate computation.

The flexibility of the **drop()** function extends beyond simply listing column names. It can also accept a column object derived from the DataFrame itself. However, for dropping columns by name, the string argument method (as demonstrated in the examples) is the most common and readable convention. When integrating this technique into complex data pipelines, developers should always verify the resulting schema using `df_new.printSchema()` to ensure that all targeted columns were successfully removed and that the remaining data types are as expected.

Using **drop()** is generally preferred over manually selecting all desired columns (e.g., using `df.select(...)`) when the number of columns to retain is large but the number of columns to remove is small. This preference simplifies the code and reduces the potential for errors caused by missing a column in a long selection list.

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

Summary and Best Practices

Effectively managing the column set within a DataFrame is a cornerstone of efficient PySpark development. We have demonstrated that the **drop()** function provides a powerful, concise, and highly efficient mechanism for excluding both single and multiple columns. By utilizing this function, practitioners can significantly enhance the performance and clarity of their ETL (Extract, Transform, Load) processes. Always remember that the result of a **drop()** operation is a new DataFrame, preserving the integrity of the original data source.

To maintain robust code quality and ensure maximum efficiency in a distributed computing environment, consider the following best practices when performing column exclusions:

Verify Schema: After any significant structural change, such as dropping columns, always use `df.printSchema()` to confirm that the expected columns were removed and that no unintended data loss occurred.

Immutability Awareness: Ensure you assign the result of the `.drop()` method to a new DataFrame variable (e.g., `df_cleaned = df.drop(...)`) to leverage Spark's immutable architecture correctly.

Efficiency of List Input: While the examples show variadic arguments (comma-separated strings), for scenarios involving a dynamic or very large list of columns to exclude, it is possible and often recommended to unpack a Python list of column names directly into the **drop()** function call for streamlined implementation.

Additional Resources

For developers seeking deeper understanding of advanced PySpark operations and optimizations, consulting the official Apache Spark documentation remains the authoritative source. Mastering functions like **drop()** is just the beginning of leveraging the full power of Spark for large-scale data processing.