How to Remove the First Column from a PySpark DataFrame

Authored by

stats writer

January 1, 2026

Data manipulation is a fundamental task in data engineering and analysis, especially when working with large datasets. The PySpark DataFrame, the primary data structure in PySpark, provides robust methods for efficiently transforming and cleaning data across distributed clusters. One of the most common requirements is removing columns that are irrelevant, redundant, or serve as temporary artifacts. While dropping columns by name is straightforward, identifying and dropping the very first column--often an arbitrary index column or a timestamp--requires specific understanding of how PySpark structures its column metadata.

To successfully drop the first column in a PySpark DataFrame, we leverage the built-in drop() method. This method is highly optimized for distributed operations and is the standard way to modify the schema of a DataFrame by removal. Crucially, the drop() method is immutable; it does not modify the original DataFrame but instead returns a brand new DataFrame instance containing the requested changes. This characteristic aligns with the functional programming paradigm often used in Spark, ensuring that transformations are predictable and safe, making debugging and complex data pipelines much easier to manage.

The Necessity of Data Manipulation: Why Drop Columns?

The process of identifying and eliminating unnecessary columns is a critical aspect of Data Cleaning and preparation. In real-world scenarios, datasets often contain extraneous features introduced during extraction, transformation, or loading (ETL) processes. For instance, datasets loaded from CSV or databases might inadvertently include an arbitrary row index or an inherited primary key that holds no analytical value for the subsequent modeling or reporting phase. Keeping these non-essential columns not only increases memory consumption across the cluster but can also slow down computational operations, as Spark must process and shuffle data associated with these columns unnecessarily.

Furthermore, removing columns is essential for effective Feature Selection. Analysts often need to prune the feature space to focus on variables that contribute most significantly to the predictive power of a machine learning model, or simply to streamline output reports. If the first column is identified as a unique ID that provides no predictive signal, or if it is a duplicate of information present elsewhere, it should be dropped. This proactive management of the DataFrame structure is a hallmark of efficient data processing in the big data environment provided by PySpark.

Fundamentals of the drop() Method in PySpark

The drop() method is the core function for column removal in PySpark DataFrames. It accepts one or more column names as string arguments. When dropping a single column, you pass the name directly. When dropping multiple columns, you can pass multiple string arguments or a list of strings representing the column names you wish to exclude from the resulting DataFrame. Since

PySpark DataFrames are schema-aware, the method automatically handles the necessary internal data rearrangement and schema update efficiently.

However, the challenge arises when we specifically target the "first column" without knowing its name beforehand. Unlike RDDs or traditional Python lists, DataFrames are columnar stores and do not inherently rely on a fixed numerical Index Position for access in the same way. While the columns have an inherent order defined by the schema, accessing them requires referencing the schema's column list. To drop the first column by its sequential order, we must first retrieve the list of column names using the .columns attribute and then use standard Python indexing to isolate the name of the column at Index Position 0. This column name is then fed into the drop() method.

Setting Up the Environment and Sample Data Creation

Before demonstrating the two primary methods for dropping the initial column, it is necessary to establish a working PySpark environment and define a representative DataFrame. All PySpark operations begin with initializing a SparkSession, which serves as the entry point to Spark functionality. We will create a small, manageable dataset detailing team performance metrics to illustrate the column dropping process clearly. The following code snippet defines the session, the data, the column headers, and constructs the initial DataFrame (df).

This setup ensures that we have a reproducible starting point where the order of columns is explicitly defined: 'team', 'conference', 'points', and 'assists'. In this scenario, 'team' is unambiguously the first column, which we intend to remove using two different approaches.

The initial DataFrame used for our examples is created as follows:

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

Method 1: Dropping the First Column Using Index Position

This method is highly versatile, especially when the column name might change based on dynamic input data or schema evolution, but the requirement is strictly to drop the column residing at the initial position (index 0). To achieve this, we first access the list of column names associated with the DataFrame using df.columns. This attribute returns a standard Python list of strings, where the order corresponds to the schema structure. We can then use standard Python list indexing () to retrieve the name of the first column.

Once the string name of the column is successfully isolated, we pass this derived name directly to the drop() method. The syntax is concise and powerful: df.drop(df.columns). This expression resolves the column name dynamically before executing the distributed drop operation. This technique is preferred when building automated scripts where explicit column names may not be hardcoded, offering robustness against minor schema alterations that preserve relative column order.

Example 1: Drop First Column in PySpark by Index Position

We execute the index-based drop operation and display the resulting DataFrame (df_new). As expected, the column at Index Position 0, which is the 'team' column, is successfully removed, leaving the remaining analytical metrics intact. This demonstrates the dynamic nature of accessing schema metadata in PySpark to perform structural transformations based on order rather than static naming conventions.

```
#create new DataFrame that drops first column by index position
df_new = df.drop(df.columns)
```

```
#view new DataFrame
df_new.show()

+----------+------+-------+
|conference|points|assists|
+----------+------+-------+
| East| 11| 4|
| East| 8| 9|
| East| 10| 3|
| West| 6| 12|
| West| 6| 4|
| East| 5| 2|
+----------+------+-------+
```

Crucially, notice that only the first column, which was the **team** column, has been dropped from the DataFrame. The schema has been successfully altered to include only 'conference', 'points', and 'assists'.

Method 2: Dropping the First Column Using the Column Name (Best Practice)

While dropping by index position is useful for dynamic scenarios, the most explicit and often preferred method for Data Cleaning and transformation in PySpark is dropping columns directly by their explicit name. If the first column is known (e.g., it is always named 'team' or 'id'), using the name guarantees that the correct column is dropped, regardless of any potential preceding columns that might have been added or removed in prior, upstream steps. Relying on an explicit name enhances code readability and reduces the risk associated with relying on fixed positional Index Position, which can be unstable in complex ETL pipelines.

This approach involves directly passing the string name of the first column--in our case, 'team'--to the drop() method. The syntax is simply df.drop('team'). This is often considered a best practice in environments where data governance dictates consistent column naming, as it makes the transformation operation self-documenting: anyone reviewing the code immediately knows which feature is being eliminated.

Example 2: Drop First Column in PySpark by Name

In this example, we demonstrate the direct name-based approach. We confirm that this method yields an identical result to the index-based approach, provided we correctly identify the name of the first column. This comparison highlights that while the outcome is the same, the methodology differs significantly in terms of robustness and reliance on schema position versus explicit naming.

```
#create new DataFrame that drops first column by name
df_new = df.drop('team')

#view new DataFrame
df_new.show()

+----------+------+-------+
|conference|points|assists|
+----------+------+-------+
| East| 11| 4|
| East| 8| 9|
| East| 10| 3|
| West| 6| 12|
| West| 6| 4|
| East| 5| 2|
+----------+------+-------+
```

As observed, only the intended first column (the **team** column) has been successfully dropped from the resultant DataFrame, confirming the efficiency and accuracy of the name-based removal process in PySpark DataFrame operations.

Comparing the Methods and Final Considerations

When deciding between dropping the first column by Index Position (df.columns) or by explicit Name, data practitioners should consider the context of their data pipeline. If the input data is rigidly structured and the first column is always guaranteed to be the one needing removal (perhaps an auto-generated row ID), the index-based approach offers generalized flexibility. However, relying on fixed positions can introduce fragility if the upstream ETL process occasionally shifts column order, leading to unintended data loss or corruption of subsequent columns.

Conversely, dropping by explicit name provides high stability and clarity. Even if a new column is inserted before 'team' in a future data update, the command df.drop('team') will still target and remove the correct column, irrespective of its new positional index. For robust, long-term production pipelines, naming the column explicitly is generally the safer and more maintainable strategy for Data Cleaning and schema management in PySpark. Regardless of the method chosen, both utilize the highly optimized drop() method, ensuring distributed efficiency.

Mastering these fundamental manipulation techniques is crucial for anyone working with distributed datasets, allowing for precise control over data structure and leading to leaner, faster, and more focused analytical results.