How to Drop the First Column in a PySpark DataFrame

Authored by

stats writer

February 6, 2026

Managing and transforming large-scale datasets efficiently is a core challenge in modern data engineering. Within the PySpark ecosystem, the primary structure for data manipulation is the **DataFrame**. A common requirement when preparing data for analysis or modeling is the removal of unnecessary or redundant columns, especially the initial column which might sometimes serve as an index or identifier.

Fortunately, PySpark provides a powerful and intuitive method for this operation: the `.drop()` function. It is essential to understand that when you use this function on a DataFrame, you are not modifying the original structure in place. Due to the immutable nature of Spark DataFrames, this operation returns an entirely new DataFrame containing only the desired remaining columns.

This comprehensive guide will detail two highly effective techniques for removing the first column: accessing it via its **index position** or targeting it specifically by its **column name**. Both methods leverage the built-in optimization capabilities of the Spark engine, ensuring efficient data processing across distributed environments.

Dropping the First Column in a PySpark DataFrame

Understanding PySpark DataFrames and Immutability

Before diving into the specific syntax for column manipulation, it is crucial to establish a foundational understanding of how PySpark manages its data structures. A **DataFrame**, in the context of Apache Spark, is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but designed for massively parallel processing (MPP) across clusters.

A key architectural characteristic of Spark DataFrames is their **immutability**. This means that once a DataFrame is created, it cannot be changed. Any operation that appears to modify the structure--such as filtering, adding a column, or dropping a column--does not alter the original object. Instead, these operations trigger the creation of a new DataFrame reflecting the desired changes.

This immutable design is vital for fault tolerance and distributed computing, as it allows Spark to optimize execution plans (Directed Acyclic Graph or DAG) without worrying about intermediate state changes. When we discuss "dropping" the first column, we are technically generating a derived DataFrame that excludes the specified column from its schema. Always remember to assign the result of the `.drop()` operation to a new variable (e.g., `df_new`) or overwrite the original variable if the original data is no longer needed.

The `drop()` Function: Syntax and Usage

The primary tool for removing columns in PySpark is the `DataFrame.drop()` transformation. This

function accepts column identifiers as arguments and efficiently generates the resulting DataFrame. It is capable of handling single columns, multiple columns, or even structured columns defined using Column objects.

To drop the first column specifically, practitioners typically rely on one of two fundamental strategies: identifying the column by its specific string name, or using Pythonic slicing to access the column name list by its zero-based index. Selecting the appropriate method depends heavily on the robustness required by your ETL pipeline. If the column order is guaranteed to be fixed, index dropping might be quicker to implement, but dropping by name offers far greater stability against schema changes.

We outline the essential syntax for both approaches below. Note that `df` represents the original **PySpark DataFrame** variable name:

Method 1: Drop First Column by Index Position

This technique dynamically extracts the name of the column residing at the first position (index `0`) and passes that name to the `.drop()` function. This is highly effective if you are certain the column you wish to remove will always be the very first one in the structure:

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

Method 2: Drop First Column by Name

The preferred and generally safer method involves directly specifying the column name as a string argument to the `.drop()` function. This ensures that the intended column is removed regardless of where it appears in the DataFrame's schema order. If the first column is named `col1`, the syntax is straightforward:

```
# Create new DataFrame that drops first column by name
df_new = df.drop('col1')
```

Setting Up the PySpark Environment and Sample DataFrame

To provide clear, practical demonstrations, we first need to initialize a **SparkSession**--the entry point for programming Spark with the Dataset and DataFrame API--and define a sample dataset. Our sample data structure contains information about hypothetical athletic performance metrics, allowing us to easily identify the first column for removal.

The data is structured with four columns: `team`, `conference`, `points`, and `assists`. The `team`

column is the first column in the schema, which we will target in our subsequent examples. We use the `spark.createDataFrame` method, passing in the raw list data and the explicitly defined column names, which is a standard procedure for bootstrapping test data within the PySpark environment.

Executing the following code snippet sets up the necessary objects and displays the initial state of our DataFrame (`df`). This initial view confirms the order of the columns before any transformation is applied:

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

# Define data for the DataFrame
data = ,
,
,
,
,
]

Define column names (Note: 'team' is the first column, index 0)
columns =

# Create dataframe using data and column names
df = spark.createDataFrame(data, columns)

# View initial 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
+----+----------+------+-------+

Practical Demonstration: Dropping by Index Position

The index-based dropping mechanism is powerful because it does not require prior knowledge of the column's name, only its sequential position within the DataFrame schema. Since all column indexing in Python (and thus PySpark) is **zero-based**, the first column is always accessed using the index `0`.

To implement this, we utilize the `df.columns` attribute, which returns a list of all column names. By accessing `df.columns`, we dynamically retrieve the string name of the first column ('team'). This string name is then passed as the argument to the `.drop()` function, ensuring the correct column is targeted regardless of how the DataFrame was initialized or generated.

The code below executes this transformation and displays the resulting schema. As expected, the new DataFrame, `df_new`, maintains all rows but excludes the initial `team` column, verifying the successful execution of the index-based drop operation:

 # Create new DataFrame that drops first column by index position (df.columns = 'team')
df_new = df.drop(df.columns)

# View new DataFrame schema and contents
df_new.show()

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

Upon inspection of the output, observe carefully that the `team` column has been successfully removed, leaving `conference`, `points`, and `assists` as the remaining schema fields. This method is concise but inherently relies on the assumption that the schema order will not change upstream.

Practical Demonstration: Dropping by Column Name

While dropping by index works, dropping a column by its explicit name is generally considered the **safest practice** in production environments. This method ensures that if a preceding ETL step

reorders the columns, the correct target (in this case, `team`) is still removed without accidentally dropping another critical piece of data.

The implementation is arguably simpler, requiring only the column name string passed directly into the `.drop()` function. For our example, since the first column is named `team`, we pass the string `'team'`.

Executing the command below results in an identical DataFrame schema to the index-based approach, but with enhanced resilience against unexpected schema modifications. This robustness is paramount when building reliable data processing pipelines on distributed systems:

 # Create new DataFrame that drops the 'team' column by name
df_new = df.drop('team')

# View new DataFrame schema and contents
df_new.show()

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

The resulting DataFrame confirms that the column `team` was successfully identified and excluded. We can clearly see the benefits of using the name: if `conference` had somehow become the first column, the index method would have removed `conference`, whereas this name-based method would still correctly remove `team`, regardless of its position.

Performance Considerations and Best Practices

When working with massive datasets in Spark, understanding the performance implications of transformations like column dropping is crucial. The `.drop()` operation is considered an optimized transformation because it only modifies the metadata (the schema) of the existing RDDs (Resilient Distributed Datasets) that back the DataFrame, rather than immediately triggering a full data shuffle or computation.

Specifically, dropping a column is part of Spark's **lazy execution** model. The actual physical removal of the data does not occur until an action (like `.show()`, `.count()`, or `.write()`) is called. When an action is initiated, the Spark engine optimizes the DAG and ensures that the discarded column is simply not included in the subsequent physical execution plan across the cluster worker nodes. This efficiency minimizes resource utilization, making column dropping a very fast schema operation.

For best practices, always prioritize dropping columns by **name** in production pipelines for schema stability, as demonstrated in Method 2. While indexing (Method 1) is useful for exploration or scenarios where column order is guaranteed (like immediately after a known selection operation), relying on index positions can lead to catastrophic data corruption if upstream processes alter the column sequence without warning. Furthermore, if you need to drop multiple columns, the `.drop()` function accepts multiple string arguments simultaneously (e.g., `df.drop('col1', 'col2', 'col3')`), which is far more efficient than applying multiple single drops sequentially.

Conclusion and Further Resources

Dropping the first column in a Spark DataFrame is a routine preprocessing step, and the `.drop()` function offers both flexibility and high performance for this task. Whether you choose the dynamic approach of identifying the column by its index position (`df.columns`) or the more robust, explicit approach of using the column's string name, both methods leverage Spark's underlying efficiencies to ensure rapid schema modification without unnecessary data processing.

The key takeaway is to embrace the immutability of Spark DataFrames and ensure that the result of the drop operation is always captured in a new variable. This adherence to functional programming principles maintains data integrity across complex distributed transformations.

Mastering fundamental operations like dropping columns is essential for effective data manipulation in the big data environment. Continue exploring the extensive capabilities of the Spark SQL API to optimize your data workflows.

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