Learning PySpark: How to Drop the First Column of a DataFrame

Authored by

Mohammed looti

November 11, 2025

Introduction to Efficient Column Management in PySpark

Apache Spark, particularly when utilized through its Python API, PySpark DataFrame, is the dominant engine for large-scale data processing and transformation in modern data engineering pipelines. A fundamental task in data preparation involves managing the structure of these DataFrames, which frequently requires the removal of unnecessary or redundant columns. While removing columns by name is straightforward, situations often arise where the need is to dynamically drop the very first column, regardless of its specific label. This scenario typically occurs when dealing with sequential data loads, temporary index columns generated during ingestion, or legacy files that include an unwanted header or sequence number as the initial field.

Understanding how to precisely target and remove columns is crucial for maintaining data integrity and optimizing processing speed. Since Spark operations are immutable--meaning they return a new DataFrame rather than modifying the original in place--the method used must be efficient and clear. This article explores the two primary, reliable methods for dropping the leading column from a PySpark DataFrame: utilizing the column's **index position** and referencing the column's **explicit name**. Both techniques yield the same result but are suitable for different programming contexts and levels of structural uncertainty within the input data.

We will demonstrate these techniques using practical, reproducible code examples. Whether you are dealing with schema evolution or simply performing routine data cleaning, mastering these methods is essential for any data professional working within the Spark ecosystem. The choice between index-based and name-based dropping often depends on whether the column order is guaranteed to be stable throughout the processing pipeline.

Prerequisites: Initializing the Environment and Sample Data

Before demonstrating the column dropping techniques, we must first establish a working SparkSession and define a sample PySpark DataFrame. The creation of a SparkSession is the entry point to using Spark functionality, allowing us to interact with the underlying cluster resources and create structured data objects. For demonstration purposes, our DataFrame will consist of four columns: `team`, `conference`, `points`, and `assists`. The goal in the subsequent steps will be to effectively remove the `team` column, which occupies the first position in the schema.

The following code snippet initializes the necessary components and creates the initial DataFrame, which serves as the baseline for all subsequent transformations. Pay close attention to how the DataFrame is constructed from a list of tuples and a defined list of column names, ensuring the `team` column is correctly placed at index position zero (0).

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 observed in the output, the `team` column is clearly the first column. This structure allows us to test both the index-based and name-based removal methods effectively, confirming that the resulting DataFrame retains the three subsequent columns: `conference`, `points`, and `assists`.

Core Technique 1: Dropping the First Column Using Index Position

One of the most robust ways to ensure the removal of the very first column, regardless of its specific name, is by utilizing its index position. In Python and thus in PySpark, indexing is zero-based, meaning the first element or column is always located at index `0`. The key to this technique lies in accessing the DataFrame's list of column names, which is available via the `df.columns` attribute, and then applying the standard `drop()` transformation.

The syntax for retrieving the name of the first column is `df.columns`. When this expression is

passed as an argument to the `df.drop()` method, Spark processes the request by identifying the column name dynamically and returning a new DataFrame that excludes that specific field. This approach is highly valuable in automated pipelines where the schema might vary slightly, but the unwanted column is consistently positioned first--for instance, an auto-generated row ID from an upstream system.

Here is the implementation of the index-based dropping technique, followed by the verification of the resulting DataFrame structure. The use of `df.columns` guarantees that the column at the start of the current DataFrame schema is targeted for removal.

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

Upon viewing the output, we confirm that the original first column, the **team** column, has been successfully removed. This method provides a flexible and dynamic way to manage column exclusion based purely on structural position rather than relying on a hardcoded name. This flexibility is often preferred in large-scale data transformation jobs where upstream data sources may change header names unexpectedly.

Core Technique 2: Dropping the First Column by Explicit Name

The second, and often simplest, approach is to drop the column by explicitly referencing its name. While this method requires prior knowledge of the column's label, it offers the greatest clarity and stability when the schema is well-defined and unlikely to change. If you are certain that the first column is named `team`, or `row_id`, or any other specific identifier, using the name directly is the most readable and maintainable solution.

The `drop()` method in the PySpark DataFrame API is designed to accept a string argument corresponding to the column name intended for removal. This is the standard procedure for removing columns and is generally favored in environments where code stability and deterministic operations are paramount. Although we are specifically targeting the **first** column here, the mechanism is identical to dropping any column within the DataFrame.

For our sample data, where the first column is named `team`, the implementation is highly direct. We use the following syntax to instruct Spark to generate a new DataFrame that excludes the specified column. This technique is less dynamic than the index-based approach but is far safer if the column order might shift while the column name remains constant.

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

The resulting DataFrame confirms that the **team** column has been successfully eliminated, leaving the remaining columns intact. When comparing this result to the index-based method, it is evident that both achieve the same outcome for this specific dataset. The decision between the two methods should be driven by the requirements of the overall data pipeline and the expected consistency of the input schema.

Advanced Considerations: Choosing Between Index and Name

While both the index-based and name-based methods are valid for dropping the first column, data engineers must carefully consider the trade-offs associated with each approach, particularly in large-scale production environments. The choice often reflects a fundamental design philosophy concerning schema stability and code robustness.

Using the index position (`df.columns`) is inherently dynamic. Its primary advantage is resilience against unexpected column name changes. If an upstream process renames the first column from `id` to `row_key`, the index-based drop operation will continue to function correctly, ensuring the removal of the column that occupies the first position. However, this dynamism introduces risk: if a legitimate, required column is accidentally shifted into the first position due to a schema change, the index drop operation will erroneously remove it, potentially corrupting downstream analysis without immediately triggering an error. This method sacrifices safety for flexibility in column naming.

Conversely, dropping by explicit name (e.g., `df.drop('team')`) is rigid but safer. If the column name is misspelled or if the column is entirely missing from the input schema, the `drop()` method in PySpark, by default, will not raise an exception; it will simply return the original DataFrame without modification. While this avoids pipeline failure, it means the intended transformation might not occur. However, if the column order shifts--say, `points` is moved to the first position--dropping by name guarantees that only the specifically designated column (`team`) is removed, preventing accidental deletion of the newly positioned first column. This method prioritizes safety and deterministic action over adaptability to schema reordering.

In pipelines requiring high integrity, it is often best practice to use column names. If, however, the structure dictates that a non-semantic column (like an auto-generated row number) will **always** appear first, regardless of the upstream system, the index approach offers a necessary level of automation. Data professionals should document their rationale clearly when choosing the index-based method, acknowledging the potential risk of an unintended column deletion if schema reordering occurs.

Conclusion and Best Practices for Data Cleaning

Removing unwanted columns is a routine yet critical step in preparing data for analysis or storage. PySpark provides straightforward and highly efficient mechanisms for this task through the `drop()` method, whether the target column is identified by its specific name or by its index position. Mastery of both techniques allows data engineers to write robust and adaptive data transformation code, suitable for the complex and often volatile nature of big data sources.

For maximum clarity and maintainability, the name-based approach is generally recommended when the column name is known and stable. However, for dynamic cleaning tasks where an initial placeholder column must be removed irrespective of its current label, the index-based approach (`df.drop(df.columns)`) provides the necessary flexibility. Regardless of the method chosen, remember that Spark operations are immutable; the result of the drop operation must always be assigned to a new DataFrame variable (e.g., `df_new`) to capture the transformation.

To ensure the integrity of your data pipeline, always include checks for the resulting DataFrame

schema (using `df_new.printSchema()`) after dropping columns to verify that the intended fields were removed and that no critical columns were inadvertently affected. Effective column management is a cornerstone of performance optimization and reliable data engineering within the Spark ecosystem.

Additional Resources for PySpark Column Operations

The following resources offer further insights into performing other common data manipulation tasks within PySpark, building upon the foundational knowledge of dropping columns:

Tutorial on renaming multiple columns simultaneously in PySpark DataFrames.

Guide to selecting columns based on data type or pattern matching using the Python API.

Documentation covering the use of the `withColumn` and `withColumnRenamed` methods for complex transformations.