How to Select and Keep Specific Columns in PySpark DataFrames

Authored by

stats writer

February 11, 2026

Keep Certain Columns in PySpark (With Examples)

In the expansive landscape of **big data** processing, efficiency is the cornerstone of successful **data analysis**. When working with **PySpark**, developers frequently encounter **DataFrames** that contain a vast number of columns, many of which may be irrelevant to the specific task at hand. Learning how to effectively isolate and retain only the most pertinent data points is a fundamental skill that enhances both code readability and computational performance. By filtering out unnecessary attributes early in the workflow, you can significantly reduce the memory overhead and speed up the execution of complex transformations across a distributed cluster.

The process of keeping specific columns is generally achieved through two primary approaches: explicit selection or strategic exclusion. The **Apache Spark** engine is designed to handle these operations lazily, meaning that the actual data movement only occurs when an action is triggered. This architectural nuance allows for optimized execution plans where the system only fetches the required data from storage. Consequently, defining a streamlined schema by keeping only certain columns is not just a matter of organizational tidiness; it is a critical optimization technique for any developer working within the **PySpark** **API**.

This comprehensive guide will explore the various methodologies available for column management in **DataFrames**. We will delve into the syntax requirements, the underlying logic of different functions, and provide practical, real-world examples to illustrate these concepts. Whether you are dealing with a small exploratory dataset or a massive production-grade data lake, mastering these column-retention techniques will empower you to write cleaner, more efficient, and more maintainable **PySpark** code. We will cover the use of the select function, the drop function, and how to handle columns dynamically using programmatic lists.

You can use the following methods to only keep certain columns in a **PySpark** **DataFrame**:

Method 1: Specify Columns to Keep

from pyspark.sql.functions import col

#only keep columns 'col1' and 'col2'df.select(col('col1'), col('col2')).show()

Method 2: Specify Columns to Drop

from pyspark.sql.functions import col

#drop columns 'col3' and 'col4'df.drop(col('col3'), col('col4')).show()

Core Foundations of Column Selection

In **Apache Spark**, the **DataFrame** is a distributed collection of data organized into named columns. Conceptually, it is equivalent to a table in a relational database or a data frame in R or Python (pandas), but with much richer optimizations under the hood. When we talk about keeping certain columns, we are essentially performing a "projection" in **SQL** terms. This operation tells **PySpark** to ignore the physical data associated with the unselected columns, which can lead to dramatic improvements in I/O performance, especially when using columnar storage formats like Parquet or Avro.

The decision to keep specific columns usually stems from the need to simplify the data model. For instance, in a dataset containing user telemetry, you might have hundreds of sensor readings, but your current objective might only require the timestamp and a specific error code. By focusing only on these two columns, the resulting subset is much easier to visualize, debug, and process. Furthermore, many machine learning algorithms in the **Spark MLlib** library require a specific set of features; maintaining a clean **DataFrame** ensures that no "noise" columns are accidentally passed into the model training phase.

One must also consider the impact on network traffic. In a distributed environment, data often needs to be shuffled between different nodes in the cluster. Shuffling a **DataFrame** with fifty columns is substantially more expensive than shuffling one with only five. Therefore, early column selection serves as a proactive measure to minimize the volume of data being moved across the network. This practice is a hallmark of high-performance **Apache Spark** application development and is highly recommended for any production-level ETL (Extract, Transform, Load) pipelines.

Deep Dive into the Select Function

The `select()` function is perhaps the most frequently used method for column retention in **PySpark**. It provides a straightforward way to create a new **DataFrame** that contains only the columns specified in the arguments. This function is highly versatile; it can accept simple string names of the columns, or it can take column objects created via the `col()` function or the `df.column_name` syntax. Using column objects is generally preferred in complex applications because it allows for more advanced operations, such as aliasing or applying mathematical transformations directly within the selection process.

A significant advantage of the `select()` method is its ability to reorder columns simultaneously. If your original data has columns in the order A, B, C, D, and you need them in the order D, B, you can simply call `df.select("D", "B")`. This creates a new **DataFrame** with the desired structure in a single step. Additionally, `select()` can be used to cast data types on the fly. For example, you could select a column and cast it from a string to an integer, ensuring that your data is not only

filtered but also correctly typed for subsequent analytical steps.

When working with nested data structures, such as those found in JSON files, the `select()` function becomes even more powerful. It allows developers to "reach into" complex types like Structs or Maps to pull out specific sub-fields. By using dot notation within the selection, you can flatten a complex schema into a simple, tabular format. This ability to prune and reshape data simultaneously makes `select()` the primary tool for anyone looking to refine their datasets within the **Spark** ecosystem.

Strategic Column Removal Using the Drop Method

While `select()` is ideal when you know exactly which columns you want to keep, the `drop()` function is the superior choice when you have a large dataset and only want to remove a few specific, known columns. This "negative selection" approach is particularly useful during the iterative stages of **data analysis**. For example, after performing a join operation, you might find yourself with duplicate or redundant ID columns. Instead of listing out all forty columns you want to keep, it is far more efficient to simply drop the two redundant ones.

The `drop()` method is also very helpful when dealing with privacy or security requirements. In many organizations, data engineers are required to remove PII (Personally Identifiable Information) before passing datasets to data scientists for modeling. If a dataset has a "Social Security Number" or "Home Address" column, the `drop()` function can be used to ensure these sensitive fields are purged from the **DataFrame** immediately after the data is loaded. This ensures compliance with data protection regulations while allowing the rest of the analysis to proceed unhindered.

It is important to note that `drop()`, like most **PySpark** transformations, is immutable. When you call `df.drop("column_name")`, the original **DataFrame** remains unchanged; instead, a new **DataFrame** is returned that excludes the specified column. This immutability is a core feature of functional programming in **Spark**, as it prevents accidental side effects and makes it easier to track the lineage of your data as it moves through various transformation stages.

Practical Implementation with Sample Data

To better understand these concepts, let us look at a practical implementation. We begin by initializing a **SparkSession** and defining a sample dataset that represents sports team statistics. This dataset includes various attributes such as the team name, the conference they belong to, their points scored, and their assists. Viewing the initial state of the **DataFrame** allows us to understand the schema before we apply any filtering logic.

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


In this scenario, our raw data is comprehensive, but perhaps our current analysis only focuses on scoring efficiency. We might not care about the geographical conference or the playmaking assists at this moment. By having this clear starting point, we can now demonstrate how both `select()` and `drop()` can be used to achieve the same refined view of our data, depending on which approach feels more natural for the specific task.

Detailed Example 1: Isolating Specific Attributes

The following code demonstrates the explicit selection method. By using the `select()` function combined with the `col()` helper from the `pyspark.sql.functions` module, we define a new **DataFrame** that exclusively contains the **team** and **points** columns. This is the most descriptive way to handle column retention, as anyone reading the code can immediately see exactly which

pieces of information are being utilized for the next steps.

Example 1: Specify Columns to Keep

from pyspark.sql.functions import col

#create new DataFrame and only keep 'team' and 'points' columns
df.select(col('team'), col('points')).show()

```
+----+------+
|team|points|
+----+------+
| A| 11|
| A| 8|
| A| 10|
| B| 6|
| B| 6|
| C| 5|
+----+------+
```

As observed in the output, the resulting **DataFrame** has been successfully pruned. The **conference** and **assists** columns are no longer present. This targeted selection is highly efficient in large-scale environments because **Spark** can optimize the underlying scan to only read the data for the two requested columns from the disk, bypassing the rest entirely. This results in faster execution times and less strain on the cluster's resources.

Furthermore, this approach provides a level of "future-proofing" for your code. If the upstream data source suddenly adds twenty new columns, your `select()` statement will continue to return only the two columns you need. This prevents unexpected schema changes from breaking downstream logic that might not be prepared to handle additional, unknown columns. It establishes a "contract" between the data processing step and the data itself.

Detailed Example 2: Filtering Out Unnecessary Features

In this second example, we achieve the same result as the previous step but use the `drop()` function instead. This method is often preferred when you want to remove columns that are transitional or temporary. For instance, if the **conference** and **assists** columns were only needed for a preliminary calculation and are no longer required for the final report, dropping them is a clean way to finalize the **DataFrame**.

Example 2: Specify Columns to Drop

from pyspark.sql.functions import col

#create new DataFrame that drops 'conference' and 'assists' columns
df.drop(col('conference'), col('assists')).show()

+----+------+
|team|points|
+----+------+
A	11
A	8
A	10
B	6
B	6
C	5
+----+------+

The output is identical to the first example, but the logic is inverted. Here, we told **PySpark** which columns to discard, and it implicitly kept everything else. This is particularly advantageous in exploratory **data analysis** where you might be working with a dataset whose full schema is not yet documented. You can keep dropping columns that you find irrelevant until you are left with a manageable set of variables for your study.

One caveat to keep in mind is that if you try to drop a column name that does not exist, **PySpark** will generally not throw an error; it will simply return the **DataFrame** as is. This can be both a benefit (making scripts more resilient) and a challenge (making typos harder to spot). Therefore, it is always a good practice to verify the schema using `df.printSchema()` after performing drop operations to ensure the resulting structure matches your expectations.

Advanced Dynamic Column Selection

In advanced scenarios, you might need to keep columns based on specific criteria rather than hardcoding names. For instance, you might want to keep all columns that are of a "String" data type or all columns whose names start with the prefix "sensor_". **PySpark** makes this possible by allowing you to use Python list comprehensions in conjunction with the `df.columns` or `df.dtypes` properties. This dynamic approach is essential for building flexible data pipelines that can adapt to changing input formats without manual intervention.

Consider a situation where you have a list of one hundred columns and you need to keep all of

them except those that contain null values above a certain threshold. You could programmatically generate a list of "clean" columns and then pass that list to the `select()` function using the star (*) expansion operator. This level of automation is what separates basic scripts from sophisticated, production-grade **big data** applications. It allows for a metadata-driven approach to data engineering where the code adapts to the data it receives.

Another powerful technique involves the use of **regex** (regular expressions) for column selection. While `select()` does not natively support regex, you can use Python's `re` module to filter the `df.columns` list before passing it to the selection function. This enables very granular control, such as selecting all columns that end with a specific date format or those that match a complex naming convention. By mastering these programmatic selection methods, you can handle even the most complex and volatile data schemas with ease.

Performance Optimization and Best Practices

To truly excel at managing **PySpark** **DataFrames**, one must understand "Projection Pushdown." This is an optimization where **Spark** pushes the column selection logic as close to the data source as possible. If you are reading from a Parquet file and immediately follow the read with a `select()`, **Spark** will only read the relevant columns from the disk. This drastically reduces the amount of data loaded into memory. Therefore, it is a best practice to perform your column selection as early as possible in your script to maximize this optimization.

Another best practice is to avoid using `df.select("*")` unless absolutely necessary. While it is convenient, explicitly naming your columns makes your code more readable and prevents issues if the source schema changes. Furthermore, when selecting columns, try to use the `col()` function or `expr()` for complex logic. This makes it easier to add aliases, which improve the clarity of your final output. Clear, descriptive column names are vital when the resulting **DataFrame** is exported to a **SQL** table or a business intelligence tool for reporting.

Finally, remember to monitor your **Spark** UI. The "SQL" tab in the UI will show you the physical plan of your query. By looking at the "FileScan" nodes, you can verify if projection pushdown is working correctly. If you see that only the required columns are being read, you know your column retention strategy is working efficiently. Monitoring these metrics is key to troubleshooting performance bottlenecks in large-scale data processing tasks.

Conclusion and Next Steps

Mastering the ability to keep certain columns in **PySpark** is a foundational skill that bridges the gap between basic coding and professional data engineering. Whether you choose the explicit clarity of `select()` or the targeted convenience of `drop()`, the goal remains the same: to create a lean,

efficient, and purpose-driven dataset. By reducing the volume of data processed, you not only save time and computational costs but also make your **data analysis** pipelines much more resilient and easier to maintain.

As you continue your journey with **Apache Spark**, we encourage you to experiment with these methods on larger datasets and explore how they interact with other transformations like filtering and grouping. The more comfortable you become with manipulating **DataFrame** schemas, the more effective you will be at extracting meaningful insights from **big data**. Always remember to prioritize early selection and to document your schema transformations clearly to help your collaborators understand the flow of information.

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

How to filter rows based on multiple conditions in **PySpark**.
Understanding the difference between `select()` and `withColumn()`.
How to rename multiple columns efficiently using a dictionary.
Techniques for handling missing or null values in **DataFrames**.