Keep Certain Columns in PySpark (With Examples)

Authored by

stats writer

November 17, 2025

When working with large-scale data processing using Apache Spark, efficient manipulation of PySpark DataFrames is critical for optimizing resource consumption and speeding up analysis. One of the most fundamental operations involves controlling the dimensionality of the dataset--specifically, selecting or excluding columns. Keeping only the necessary fields is not just a matter of convenience; it significantly reduces the data shuffle size and computational overhead for subsequent transformations. This guide, intended for data engineers and analysts, details the two primary and most efficient methods available in PySpark for managing column subsets, ensuring your workflows remain lean and powerful.

The core of this column manipulation lies within the powerful DataFrame API, which offers declarative ways to transform data without requiring manual iteration. Whether you are filtering sensitive information, preparing a feature set for machine learning models, or simply reducing data volume for faster development cycles, mastering column selection is non-negotiable. We will explore both the inclusive approach, where you explicitly name the columns to retain, and the exclusive approach, where you specify columns for deletion. Understanding the nuances of each method will allow you to choose the most readable and performant option based on the context of your data engineering task.

The following sections will demonstrate these techniques using explicit code examples and detailed explanations. We will focus on the two canonical methods for achieving precise column control: the additive `select()` function, designed for inclusion, and the subtractive `drop()` function, designed for exclusion. Both methods return a new PySpark DataFrame, preserving the immutability of the original dataset, a crucial concept in distributed computing frameworks.

Method 1: Selective Inclusion using the select() Transformation

The most common and often preferred method for retaining a subset of columns is utilizing the `select()` transformation. This method requires you to explicitly list every column you wish to keep in the resulting PySpark DataFrame. This approach is highly transparent, as the resulting schema is immediately apparent from the function call. When you have a DataFrame containing dozens or even hundreds of columns, but only need a handful for downstream processing, `select()` provides the cleanest syntax. It ensures that only the necessary data is carried forward, preventing accidental inclusion of unwanted or potentially large columns.

The `select()` function accepts either string names of columns or, preferably, Column objects imported via the col() function, which is generally safer and more expressive, especially when performing complex operations or renaming columns during selection. The example below illustrates the basic syntax for selecting only `col1` and `col2`. Note that the use of `.show()` is merely for immediate output display and is typically replaced by further transformations in a real pipeline.

from pyspark.sql.functions import col

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

Using the select() method is generally recommended when the number of columns to keep is significantly smaller than the total number of columns available. This minimizes typographical errors and makes the intent of the data transformation explicit. Furthermore, the `select()` function is highly versatile; it is not limited to simple column retention. It allows for advanced operations, such as deriving new columns, applying aggregate functions, or casting data types simultaneously with the selection process, making it a cornerstone of PySpark data manipulation.

Method 2: Selective Exclusion using the drop() Transformation

In contrast to the additive nature of `select()`, the `drop()` transformation allows you to define a new DataFrame by specifying the columns you wish to remove or exclude. This method is particularly efficient and readable when your DataFrame is very wide (contains many columns), but you only need to discard a small number of unwanted fields, such as temporary calculation columns, verbose identifiers, or fields flagged for removal due to privacy concerns. Instead of listing potentially hundreds of columns to keep, you only list the few you intend to eliminate.

The `drop()` function accepts column names as strings or Column objects. If multiple columns are provided, PySpark efficiently processes the removal of all listed fields in a single step. It is crucial to remember that like all PySpark DataFrame operations, `drop()` is immutable; it generates a completely new DataFrame instance that lacks the specified columns, leaving the original DataFrame unchanged. This behavior is fundamental to Apache Spark's fault tolerance and distributed execution model.

from pyspark.sql.functions import col

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

For scenarios involving programmatic column removal--such as iterating through a list of columns identified by dynamic business logic or based on data quality reports--the `drop()` function is extremely flexible. It handles large lists of column names efficiently. A common pitfall for newcomers is attempting to drop columns that do not exist in the DataFrame; PySpark will typically ignore these non-existent columns without raising an error when using the standard string-based drop, but it is best practice to ensure the column list is accurate to maintain predictable results and a clear schema definition.

Setting Up the Foundation: Sample PySpark DataFrame

To effectively illustrate the application of both the `select()` and `drop()` methods, we first need a structured dataset. We will create a simple, small `PySpark DataFrame` using the `SparkSession` to simulate real-world data containing various types of information, such as categorical team identifiers, regional groupings, and numerical performance metrics. This sample DataFrame will serve as the baseline for all subsequent column manipulation examples.

The creation process involves defining the dataset rows (`data`) and specifying the corresponding column names (`columns`). We instantiate the Apache Spark environment using `SparkSession.builder.getOrCreate()`, which is standard practice for initiating Spark functionalities in a Python environment. The final step is utilizing `spark.createDataFrame(data, columns)` to materialize the distributed collection, allowing us to inspect its initial structure using `df.show()`.

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
+----+----------+------+-------+

The resulting DataFrame `df` contains four distinct columns: `team` (categorical identifier), `conference` (regional grouping), `points` (a numerical performance metric), and `assists` (another numerical performance metric). Our goal in the following examples will be to transform this initial structure into narrower DataFrames tailored for specific analytical tasks, demonstrating how to isolate or remove fields effectively while maintaining data integrity and performance characteristics inherent to Spark execution.

Practical Example 1: Isolating Essential Variables using select()

In many analytical scenarios, only a subset of data is required for a particular computation, such as calculating team averages or preparing input for a basic regression model. For instance, if we are solely interested in analyzing the relationship between the **team** identifier and the number of **points** scored, we should use the select() method to project the DataFrame onto just these two specific columns. This practice is vital in minimizing memory overhead and increasing the speed of subsequent group-by operations.

The code below demonstrates how to invoke `select()`, passing the desired column names, referenced using the `col()` function, ensuring that the transformation is explicit and well-defined. The resulting DataFrame, which we might assign to a new variable (e.g., `df_performance`), will inherit only the data associated with the `team` and `points` fields, effectively dropping the `conference` and `assists` fields from the projection.

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|
+----+------+

Observing the output confirms that the new DataFrame retains only the variables explicitly specified in the `select()` call. This is the hallmark of the inclusive strategy: you define what stays, and everything else is implicitly discarded. This approach ensures maximum clarity, which is especially important when dealing with complex data pipelines where multiple column transformations occur sequentially. Furthermore, if you needed to rename a column during selection (e.g., `col('points').alias('score')`), `select()` provides the immediate pathway for such structural modifications alongside data filtering.

Practical Example 2: Removing Redundant Data using drop()

Conversely, the `drop()` method becomes highly advantageous when the majority of your columns are relevant, but a small subset must be removed. In our sample DataFrame, suppose the **conference** information is redundant for a global analysis, and the **assists** metric is unreliable or irrelevant for the current task. Instead of listing `team` and `points` using `select()`, we can explicitly target `conference` and `assists` for removal using the `drop()` function.

This exclusive strategy maintains brevity and clarity. When the number of columns to drop (N) is much smaller than the total number of columns (M), writing N column names is far more efficient than writing M-N names. The implementation below shows how the original DataFrame is processed to exclude these two specific fields, retaining the rest of the schema effortlessly.

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 confirms that the resulting DataFrame is identical in structure to the output of Example 1, but achieved through the inverse operation. This demonstrates that both `select()` and `drop()` can yield the same final `PySpark DataFrame`, making the choice between them a matter of context, code readability, and maintenance ease. When defining large data pipelines, consistency in choosing the most appropriate method for column management contributes significantly to overall code quality and understandability.

Understanding Performance Implications of Column Selection

While both the `select()` and `drop()` transformations achieve the goal of modifying the schema of a DataFrame, understanding their performance implications, especially in large-scale Apache Spark environments, is crucial. Since PySpark operates on distributed data across a cluster, minimizing the volume of data that needs to be moved or processed is the primary objective for optimization. When you execute a column selection or drop, this operation is pushed down to the executors, determining which data partitions are read and materialized.

Generally, neither `select()` nor `drop()` inherently triggers a heavy data shuffle across the network. They are both narrow transformations, meaning the data required to compute the results for a single partition is available within that partition. However, the true performance benefit comes from reducing the subsequent size of the data being processed. If you have 100 columns and only need 5, using `select()` immediately prunes the width of the data records, meaning subsequent wide transformations (like joins or aggregations) will operate on significantly smaller data payloads. This effect is magnified when dealing with columns containing very large data types, such as lengthy strings or complex arrays.

The choice between `select()` and `drop()` is often one of convenience and readability, rather than raw computational speed, assuming both are used correctly. If 95% of your columns are needed, using `drop()` to remove the remaining 5% is faster to write and maintain. Conversely, if 5% of your columns are needed, select() is the obvious choice. Choosing the method that requires the shortest list of column names minimizes potential runtime errors from typos and ensures that the execution plan (the DAG) generated by Spark's Catalyst optimizer is as efficient as possible for the task at hand. The Catalyst optimizer is highly effective at optimizing these specific narrow operations, ensuring that column pruning occurs early in the execution plan.

It is also important to note how PySpark handles metadata. When you use these functions, the underlying data files (like Parquet or ORC) are often read columnarly. By applying `select()` early in your pipeline, Spark can entirely skip reading the bytes associated with the unselected columns from disk, leading to substantial I/O savings. This optimization is far more impactful than any minor difference in the execution time between the `select()` and `drop()` functions themselves. Therefore, the best practice is always to perform column pruning as early as possible after loading

the initial raw DataFrame.

Handling Column Lists Programmatically

While the previous examples demonstrated column selection using hardcoded string names or the col() function for clarity, real-world data engineering often requires dynamic manipulation of column sets. Instead of writing out every column name individually, which is tedious and error-prone, PySpark allows you to pass lists of column names directly to both the `select()` and `drop()` methods, significantly enhancing code modularity and maintainability.

For example, if you wanted to select a predefined list of features, you can store those names in a standard Python list. The `select()` function can accept this list directly when combined with the Python splat operator (`*`) to unpack the list elements into positional arguments. Alternatively, if you are only using string names, `select()` can often accept the list directly, but using the splat operator on `col()` wrapped objects is the robust way to handle programmatic selection, especially if you need to derive expressions within the selection.

Consider a scenario where you want to drop all columns that end with '_temp'. You could first generate the list of columns to be dropped by introspecting the DataFrame schema and then passing that list to `drop()`:

Retrieve all column names using `df.columns`.
Filter this list using standard Python list comprehension to identify target columns.
Pass the resulting list of strings to `df.drop(*columns_to_drop)`.

This technique decouples the column selection logic from the core transformation code, making it easier to manage evolving schemas and feature sets.

When working with `select()` and dynamically generated lists, a useful pattern involves ensuring the order of the selected columns matches a specific requirement, such as alphabetical order or a defined input order for a downstream model. Because `select()` respects the order in which columns are provided, controlling the input list is critical for maintaining a predictable output schema. This is particularly relevant when interacting with systems that rely on strict column ordering, such as some legacy data warehouses or specialized machine learning serving layers.

Best Practices for Column Management in Complex Pipelines

Effective column management extends beyond simply knowing how to use `select()` and `drop()`; it involves strategic placement of these operations within the larger data pipeline to maximize efficiency and maintainability. Adopting a set of best practices ensures that your Apache Spark jobs are resilient, fast, and easy to debug.

The primary best practice is **Early Pruning**. As noted earlier, column pruning should occur immediately after data ingestion or large joins. Reading less data from storage and transferring fewer bytes across the network is the most significant performance gain available. If you load a massive dataset from S3 or HDFS, apply `select()` immediately to reduce the DataFrame width before performing any costly transformations or filters. This approach adheres to the principle of "fail fast and thin," ensuring that only essential data proceeds through the pipeline.

Another crucial practice is **Explicit Column Referencing**. Although you can pass raw string names to `select()` and `drop()`, utilizing the col() function is generally preferred, especially for complex column expressions or when defining transformations that involve aliases or arithmetic. Explicitly referencing columns using `col('name')` makes the code less ambiguous and helps catch errors related to column existence or syntax during development, although the function is often implicitly applied by Spark when using strings in simple selections. For consistency and robustness, especially when dealing with ambiguous column names or programmatic construction, stick to explicit references.

Finally, ensure **Schema Documentation and Validation**. Any significant change in column structure (whether via `select()` or `drop()`) should be documented. In production environments, it is often worthwhile to implement schema validation checks after critical column management steps. This involves comparing the resulting DataFrame's schema against an expected template. Tools like Spark's built-in schema introspection (`df.printSchema()`) or external libraries can confirm that only the intended columns remain and that their data types were preserved correctly, preventing downstream failures related to missing fields or unexpected type coercion.

Summary of Column Management Techniques

In summary, managing columns in Apache Spark is a fundamental skill for efficient data processing. The decision between using `select()` and `drop()` is primarily guided by the cardinality of the column list you need to define.

Use `select()`: When the number of columns to keep is much smaller than the total number of columns. This is the inclusive approach, offering high visibility into the resulting schema. It is also mandatory if you need to perform transformations, derivations, or renaming during the selection process.

Use `drop()`: When the number of columns to remove is much smaller than the total number of columns. This is the exclusive approach, optimizing for minimal code when most fields are required. It is straightforward and avoids the necessity of listing dozens or hundreds of column names to retain.

Mastering these two transformations allows developers to create precise, performant, and

maintainable data pipelines in PySpark, ensuring that computational resources are focused only on the essential data required for insightful analysis.