Learning PySpark: Creating New DataFrames from Existing DataFrames

Authored by

Mohammed looti

November 10, 2025

Mastering PySpark DataFrame Derivation and Projection

In the world of big data, particularly within the Apache Spark ecosystem, the efficient handling of massive datasets is non-negotiable. PySpark DataFrames serve as the foundational, structured abstraction for processing data, mirroring the functionality of tables found in a traditional relational database. A common and critical requirement in analytical workflows is the need to create a smaller, more focused DataFrame derived from a larger, existing source. This process--known as projection or subsetting--is essential for optimizing computational resources, streamlining subsequent transformations, and enhancing the overall performance of large-scale data pipelines, especially when dealing with wide tables that may contain hundreds of unnecessary fields.

The ability to rapidly define a new DataFrame based on a specific subset of columns from a parent DataFrame is a fundamental skill for any data professional utilizing PySpark. This technique adheres strictly to Spark's philosophy of **lazy evaluation**, meaning that the definition of the transformation is merely recorded until an action is triggered (such as `show()`, `collect()`, or `write()`). We primarily rely on two highly effective and contrasting methods for achieving this column-based derivation, each tailored to different requirements concerning the target schema size and clarity.

These two core transformations offer approaches based either on explicit inclusion or explicit exclusion of fields. Understanding the technical nuances and practical trade-offs between these methods is vital for writing clean, performant, and maintainable code. Choosing the correct approach can significantly impact both code readability and execution efficiency across the cluster.

The following sections detail these essential approaches for data manipulation:

Method 1: Explicit Inclusion -- Defining the new DataFrame by specifying the columns to **keep** (using the `select` transformation).

Method 2: Explicit Exclusion -- Defining the new DataFrame by specifying the columns to **drop** (using the `drop` transformation).

Method 1: Selective Column Inclusion using `df.select()`

The `select()` transformation is arguably the most common and robust method used for creating a derived DataFrame. Its core function is to project a new DataFrame whose schema consists solely of the fields explicitly listed in the method call. This approach is strongly favored when the source DataFrame is extremely wide (containing many dozens or hundreds of columns), but the analytical task only requires a small, specific subset of those fields. Explicitly listing the few columns needed is far more practical and less error-prone than attempting to list the many columns that should be excluded.

The general syntax for implementing df.select() is straightforward: the method is called on the existing DataFrame, and the desired column names are passed as sequential string arguments. This methodology offers immense benefits in terms of code clarity and documentation, as the resulting DataFrame's schema is immediately visible and verifiable. Furthermore, by minimizing the data volume associated with the resulting DataFrame, developers reduce **memory consumption** and decrease the data payload required for subsequent shuffle operations across the distributed cluster.

Beyond simple column projection, the `select()` function is extremely flexible and supports complex expressions. Developers can rename columns, apply built-in PySpark SQL functions (like aggregates or window functions), or perform calculations directly within the `select()` call. This powerful capability allows complex data preparation and schema refinement, such as transforming `F.col('points').alias('score')`, to be executed simultaneously with the column selection process, thereby streamlining the overall data preparation pipeline.

The following code snippet demonstrates the basic application of `select()` to create a new DataFrame (`df_new`) containing only the `team` and `points` fields from the original DataFrame (`df`):

#create new dataframe using 'team' and 'points' columns from existing dataframe
df_new = df.select('team', 'points')

Method 2: Excluding Unnecessary Columns using `df.drop()`

In contrast to `select()`, the df.drop() method is utilized when the primary objective is to maintain the vast majority of columns from the source DataFrame while specifically excluding a small, designated set of fields. This methodology is particularly useful when dealing with DataFrames that have a moderate number of columns (e.g., 10 to 20 fields), where listing the one or two columns to remove is significantly more concise and readable than listing the eighteen columns intended to be kept.

The fundamental strength of `drop()` lies in its simplicity when addressing redundant, **sensitive data** (such as PII that must be filtered out early in the pipeline), or low-value metadata that is no longer relevant for downstream analytics or machine learning model training. By systematically removing these superfluous fields, the data footprint is minimized, which is crucial for improving the performance of iterative processes and reducing storage requirements. This method offers a cleaner alternative to `select()` when the exclusion list is short.

The syntax for `drop()` is analogous to `select()`, requiring the developer to pass the names of the columns as string arguments. However, instead of including them, the function processes the request by discarding these fields. Importantly, df.drop() is engineered to handle multiple column

inputs seamlessly, accepting them as separate positional arguments, thereby allowing for the rapid removal of several fields in a single, concise command.

The following example illustrates how to use `drop()` to create a new DataFrame containing all columns except `conference`:

#create new dataframe using all columns from existing dataframe except 'conference'
df_new = df.drop('conference')

It is paramount to note the underlying principle of **immutability** in PySpark DataFrames: both `select()` and `drop()` always return a completely new DataFrame object. The original DataFrame (e.g., `df`) remains entirely unaltered, ensuring its integrity is preserved for any parallel or subsequent transformations that might require the full initial schema.

Setting Up the Environment and Source Data

To effectively demonstrate the practical application of `select()` and `drop()`, we must first initialize the execution environment. This begins with establishing a SparkSession, which acts as the definitive entry point for all PySpark programming functionality. Once the session is active and configured, the next step involves defining a sample dataset and its corresponding structure, which will serve as the necessary source for all subsequent column manipulation operations.

For this demonstration, we utilize a small, representative dataset detailing simple sports statistics, including team identifiers, conference affiliations, and key numerical metrics such as points and assists. This diverse composition of fields allows us to clearly illustrate how both the inclusion (`select()`) and exclusion (`drop()`) transformations effectively partition the source data structure according to specific analytical needs.

The following code block initializes the Spark environment, defines the structured data, creates the foundational source DataFrame (`df`), and displays both the resulting schema and the content of the data:

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

This initial DataFrame, `df`, provides the necessary context for the subsequent practical examples. It allows us to demonstrate how efficient data derivation can focus the dataset for specific analytical tasks, such as performance calculation (requiring only `team` and `points`) or geographical distribution studies (requiring `conference` and `assists`).

Practical Example 1: Explicit Inclusion with `select()`

Imagine an analyst requires a dataset focused solely on measuring team scoring performance, necessitating only the `team` identifier and the `points` metric. The `conference` and `assists` columns are irrelevant for this specific task. Utilizing the select() function is the most explicit and reliable mechanism for achieving this precise data projection. By explicitly naming the fields to be included, we immediately discard the unnecessary data, ensuring maximum efficiency.

The resulting DataFrame generated by this selection operation is substantially narrower than the source. This reduction in width translates directly into improved performance, especially if the derived DataFrame is destined for a resource-intensive operation, such as a large-scale join or persistence to a storage layer (writing to disk). By processing only the necessary attributes, we significantly optimize resource utilization across the underlying Spark cluster.

The following syntax demonstrates the creation of `df_new` containing only the **team** and **points** columns:

```
#create new dataframe using 'team' and 'points' columns from existing dataframe
df_new = df.select('team', 'points')

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

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

The output confirms that the new DataFrame, `df_new`, successfully incorporates only the **team** and **points** fields, precisely adhering to the projection specified in the `select()` transformation. This level of explicit control over the resulting data structure is a primary reason why `select()` is often the preferred choice for precise data preparation tasks.

Practical Example 2: Explicit Exclusion with `drop()`

Consider an alternative scenario where nearly all data points are required, but a specific column--say, `conference`--must be scrubbed because it contains sensitive organizational data that should not be exposed to a downstream application. In this situation, where three out of four columns must be retained, removing the single unwanted column via df.drop() represents the most concise and efficient coding solution.

This technique excels when the number of columns slated for exclusion is minimal relative to the overall schema size. Employing `drop()` results in significantly cleaner and more maintainable code, particularly when the source DataFrame schema is large and subject to incremental changes or additions. It minimizes the risk of typographical errors inherent in listing many columns and reduces cognitive load for developers.

The following syntax demonstrates the creation of `df_new` containing all columns from the existing DataFrame *except* the **conference** column:

#create new dataframe using all columns from existing dataframe except 'conference'
df_new = df.drop('conference')

```
+----+------+-------+
|team|points|assists|
+----+------+-------+
| A| 11| 4|
| A| 8| 9|
| A| 10| 3|
| B| 6| 12|
| B| 6| 4|
| C| 5| 2|
+----+------+-------+
```

As confirmed by the output, the resulting DataFrame successfully retains **team**, **points**, and **assists**, while the target **conference** column has been correctly eliminated. Furthermore, remember that `drop()` is versatile: to remove multiple columns, one simply provides additional column names as comma-separated arguments, such as `df.drop('conference', 'assists')`.

Choosing the Optimal Transformation: Select vs. Drop

Although both `select()` and `drop()` fulfill the core requirement of deriving a new DataFrame, selecting the optimal method is crucial for ensuring coding best practices, minimizing maintenance overhead, and guaranteeing data pipeline stability. The decision should primarily hinge on the ratio of columns to be kept versus columns to be discarded.

If a developer is required to retain only a small number of columns (e.g., keeping 5 out of 100), `select()` is the unequivocal choice. Listing 5 columns is faster, easier, and dramatically less susceptible to typographical errors than attempting to enumerate 95 columns to drop. Crucially, `select()` offers superior stability against upstream schema changes: if 50 non-essential columns are added to the source dataset, the `select()` statement remains perfectly functional and continues to produce the exact, desired schema. A `drop()` statement, however, would inadvertently include all those new, potentially unwanted fields.

Conversely, if 95 columns must be retained from a source of 100, `drop()` is superior for code brevity. Listing the 5 columns for removal is far more manageable than listing 95 columns for retention. However, this convenience introduces a maintenance risk: if one of the columns intended for removal is renamed upstream, the `drop()` function will silently fail to find the old column name and will consequently pass the newly renamed column through to the resulting DataFrame. This failure could potentially lead to data leakage or unnecessary inflation of data size,

necessitating vigilant schema management when `drop()` is used on volatile source schemas.

Ultimately, both transformations are indispensable tools for efficient data manipulation in Apache Spark. They are foundational operations that enable data engineers to manage the intrinsic scale of big data by ensuring that only the absolutely necessary information is carried forward throughout the complex processing pipeline.

Conclusion and PySpark Best Practices

Deriving a new PySpark DataFrame from an existing one is a core, repetitive task in data engineering. By mastering the application of the `select()` and `drop()` transformations, developers gain fine-grained control over the data structure, which is essential for maximizing computational efficiency and optimizing cluster resources. The `select()` method champions explicit inclusion and guarantees schema consistency, while the `drop()` method prioritizes brevity through explicit exclusion when only a few fields need elimination.

As a key best practice, developers should generally favor the most explicit method possible, even if it results in slightly longer code, to maximize code readability and resilience. If a particular column is mandatory for downstream analysis, explicitly selecting it is always preferred over relying on `drop()` to implicitly keep it. This strategy substantially lowers the cognitive burden for future code maintainers and acts as a robust safeguard against unintended data exposure or processing errors triggered by unforeseen upstream schema volatility.

These transformations, combined with the inherent immutability of Spark's core data structures, ensure that data pipelines built on SparkSession are robust, predictable, and highly scalable, forming the technical backbone of effective big data analysis.

Additional Resources for Advanced PySpark

For deeper technical insights into advanced DataFrame operations, schema manipulation, and performance tuning within the PySpark environment, please consult the official Apache Spark documentation and related academic publications.