PySpark: Create New DataFrame from Existing DataFrame

Authored by

stats writer

November 16, 2025

ARABPSYCHOLOGY.COM

RECOMMENDED CITATION

stats writer (2025). *PySpark: Create New DataFrame from Existing DataFrame*. PSYCHOLOGICAL SCALES. Retrieved from https://scales.arabpsychology.com/?p=92571

Mastering PySpark DataFrame Manipulation

As data volumes continue to grow exponentially, the ability to efficiently manipulate and subset large datasets becomes paramount for modern data engineers and scientists. PySpark, the Python API for Apache Spark, provides robust tools for handling massive distributed datasets, primarily through its core structure: the **DataFrame**. Creating a new, smaller, or filtered **DataFrame** from an existing one is a fundamental task necessary for focused analysis, reducing memory footprint, and preparing data for specific modeling stages. This process ensures that downstream operations are performed only on relevant data, optimizing computational resources across the cluster.

The essence of efficient data processing in Spark lies in avoiding unnecessary computations. When dealing with wide tables (DataFrames containing numerous columns), retaining only the columns required for a specific task dramatically improves performance across the cluster. While there are many ways to transform DataFrames in **PySpark**, the two most common and idiomatic methods for creating a new **DataFrame** based on the column structure of an existing one involve explicit inclusion or explicit exclusion. These methods offer straightforward syntax and integrate seamlessly into complex ETL pipelines, ensuring data integrity while optimizing schema management.

This expert guide will systematically detail the two primary mechanisms for deriving a new **DataFrame**: specifying which columns to keep (the inclusive method using `select()`) and specifying which columns to discard (the exclusive method using `drop()`). We will explore the syntax, practical implementation, and best practices associated with each approach, providing concrete code examples using **PySpark's** built-in functions. Understanding these methods is crucial for anyone working with large-scale data processing workflows and seeking to write clean, maintainable, and highly optimized **Spark** code.

Overview of PySpark Subsetting Methods

The decision between including columns via selection or excluding columns via dropping often depends directly on the scale and complexity of the operation. If an existing **DataFrame** has dozens of columns but you only need three, the selection method is superior due to its explicitness and brevity. Conversely, if you need almost all columns but wish to omit just one or two temporary calculation columns, the exclusion method is faster to implement and easier to read, as it avoids listing every column name you intend to keep. Importantly, both techniques yield an immutable transformation, meaning the original **DataFrame** remains unchanged, and a new **DataFrame** object is instantiated reflecting the desired schema subset.

These transformation operations are built upon **Spark's** lazy evaluation engine. When you call `select()` or `drop()`, you are not immediately executing a costly operation across the cluster;

ARABPSYCHOLOGY.COM

rather, you are building a Directed Acyclic Graph (DAG) of transformations. The actual computation only occurs when an action (such as `show()`, `collect()`, or writing the data) is called. This lazy nature allows **Spark's** Catalyst Optimizer to intelligently manage the execution plan, ensuring that only the minimum necessary data reading and processing are performed to achieve the final required schema.

To demonstrate these concepts clearly, we first outline the syntax for both approaches. These short examples demonstrate the elegant efficiency of **PySpark's** API, highlighting how complex distributed operations are condensed into simple, readable method calls attached to the **DataFrame** object itself, laying the groundwork for the detailed examples that follow.

Method 1: Precision Column Selection using `select()`

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

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

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

Practical Implementation: Setting up the Environment and Data

Before diving into the specific examples of column manipulation, we must first establish a working environment and generate the source **PySpark DataFrame** that will be used throughout the demonstration. All distributed processing within **Spark** requires an active SparkSession, which serves as the entry point for interacting with the underlying **Spark** cluster infrastructure. We initiate this session using the `builder.getOrCreate()` pattern, which ensures we reuse an existing session if available, or create a new one otherwise.

The dataset we will employ models sports performance statistics, containing information on team, conference affiliation, points scored, and assists made. This realistic, yet simple, data structure allows us to demonstrate how to select specific metrics (e.g., scoring data) while ignoring categorical identifiers (e.g., conference), or vice versa, providing a clear context for our subsetting operations. The following code snippet defines the raw data as a list of lists and defines the corresponding column names, which are essential for creating a structured **DataFrame**.

The creation of the **DataFrame** using `spark.createDataFrame()` is the essential preparatory step. It transforms the local Python list structure into a distributed, schema-aware object that **PySpark** can process efficiently across partitions. We then use the `df.show()` action to visualize

ARABPSYCHOLOGY.COM

the initial state of the dataset, confirming its structure and content before proceeding with any subsequent column selection or dropping operations. This confirmation step is crucial for verifying the integrity of the input data.

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

# Define data structure
data = ,
,
,
,
,
]

Define column names
columns =

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

# View original 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
+----+----------+------+-------+

Detailed Walkthrough: Using `select()` to Isolate Data Subsets (Example 1)

The `select()` function is arguably the most fundamental method for reshaping and refining a **DataFrame's** structure. It operates by allowing the user to specify, either by name or by expression, exactly which columns they wish to retain in the resulting **DataFrame**. This approach is highly effective when working with DataFrames where the vast majority of columns are irrelevant to

ARABPSYCHOLOGY.COM

the current analytical context, minimizing data transfer and processing overhead. For our first example, we assume we are only interested in correlating the **team** identity with the raw **points** metric, and we do not require the 'conference' or 'assists' data for this specific analysis.

To achieve this schema isolation, we simply pass the desired column names as distinct string arguments to the `select()` method. The output, assigned to `df_new`, will be a brand new **DataFrame** containing only those specified columns, meticulously preserving the original data integrity and row alignment while discarding the unnecessary fields. This method ensures maximum clarity regarding the resultant schema, as the code explicitly lists every column that will be present in the output. Furthermore, `select()` is highly versatile, permitting simultaneous column renaming, applying transformations, or calculating aggregate metrics within the same function call.

Observe the executed code block below. The transformation is concise yet powerful. We explicitly list 'team' and 'points', and only these two columns are retained in `df_new`. This demonstrates the precision control offered by the inclusive selection method, confirming that schema projection is achieved with minimal code complexity, which is essential when developing robust and maintainable PySpark code in large environments.

 # 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 resulting **DataFrame** confirms that only the **team** and **points** columns were propagated from the existing **DataFrame**, effectively filtering the schema down to the required elements. This approach is often the preferred method when the number of desired columns is significantly smaller than the total number of columns available in the source data.

ARABPSYCHOLOGY.COM

Detailed Walkthrough: Using `drop()` for Efficient Column Exclusion (Example 2)

In contrast to the inclusive nature of `select()`, the `drop()` function provides an exclusive mechanism for creating a new **DataFrame**. This method is highly efficient when the goal is to remove a small subset of columns while preserving the overwhelming majority of the existing structure. Common use cases include removing calculated intermediate variables, sensitive data fields that need to be masked, or temporary metadata columns that are no longer needed for the final output stage of an ETL pipeline.

For this second example, let us assume we require all statistical measures ('points' and 'assists') along with the 'team' identifier, but the 'conference' field is deemed unnecessary for downstream processing or final storage. Rather than listing the three columns we wish to keep, it is computationally and syntactically simpler to use `drop()` to explicitly remove the single unwanted column. The syntax involves calling the `drop()` method on the existing **DataFrame** and passing the name of the column, or columns, to be excluded as a string argument.

This approach shines particularly when the source **DataFrame** is subject to frequent schema updates, perhaps adding new, relevant columns over time. If you use `drop()` to exclude known unneeded columns, the subsequent inclusion of new, necessary columns in the source data will automatically be inherited by the new **DataFrame** without requiring code modification, significantly enhancing pipeline robustness and minimizing maintenance overhead associated with schema evolution.

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

The resultant **DataFrame** clearly shows that the 'conference' column has been successfully excluded from the schema, leaving 'team', 'points', and 'assists'. This confirms the efficiency of the

ARABPSYCHOLOGY.COM

exclusive subsetting method when the objective is to prune a small number of fields while retaining the overall structural integrity of the data.

Handling Multiple Columns and Dynamic Schemas

While the previous examples focused on manipulating a single column for simplicity, both `select()` and `drop()` are designed to handle multiple columns seamlessly. Utilizing multiple columns is standard practice in real-world data processing where schemas are often complex and transformations need to be applied across several fields simultaneously. The method for including or excluding multiple columns is consistent across both functions, relying on passing multiple column names as sequential string arguments.

When using the `drop()` function, one can specify any number of columns to exclude simply by listing them, separated by commas, within the function call. For example, `df_new = df.drop('conference', 'assists')`. This flexibility allows for the rapid removal of many auxiliary columns in a single, clear line of code, significantly improving script conciseness. It is critical to ensure that the column names provided exactly match the case and spelling of the columns in the original **DataFrame's** schema, as **PySpark** operations are case-sensitive by default.

For advanced PySpark development, especially in production environments, managing column lists dynamically is often necessary to handle schema changes gracefully. Instead of hardcoding column names, a best practice is to define lists of desired or excluded columns (e.g., `KEEP_COLUMNS = `) and pass these lists to the functions using the Python argument unpacking operator (`*`). This looks like: `df.select(*KEEP_COLUMNS)`. This technique enhances code modularity and dramatically simplifies updates when schema changes occur in upstream systems, reinforcing the principle of writing robust and adaptable distributed processing code.

Advanced Considerations: Performance and Optimization

While both `select()` and `drop()` appear functionally similar in achieving data subsetting, there are nuanced performance and optimization considerations to be aware of in large-scale **PySpark** deployments. Since Spark processes transformations lazily, the underlying mechanism attempts to optimize the required data reads and projections. In most modern **Spark** environments using the Catalyst Optimizer, the performance difference between `select()` and `drop()` for simple column manipulation is generally negligible, as the plan is optimized before execution. However, choosing the most concise syntax remains a critical component of code quality.

A key recommendation is to utilize `select()` explicitly when the task involves more than simple subsetting, such as renaming columns (e.g., `select(col('points').alias('score'))`),

ARABPSYCHOLOGY.COM

applying calculated fields, or changing data types. Since `select()` is designed for schema projection, it is the more versatile tool for complex transformations. Conversely, if the schema is large (e.g., hundreds of columns) and you only need to remove a small number of fields (e.g., less than 5%), `drop()` is typically the cleaner and faster operation from a development perspective, minimizing keystrokes and potential errors from listing extensive column names.

It is important to remember that these transformations are only schema modifications until an action is triggered. The true performance impact comes from the optimization of subsequent operations. By subsetting early in the ETL process, you reduce the size of the intermediate data structures, minimizing shuffling and serialization costs across the cluster. This proactive pruning of irrelevant columns is fundamental to achieving high-throughput and low-latency data pipelines in a distributed computing framework like **Spark**.

Conclusion: Leveraging Efficient Data Subsetting in PySpark

The ability to quickly and accurately subset a **PySpark DataFrame** is indispensable for achieving efficiency in distributed computing environments. By mastering the complementary functions `select()` and `drop()`, data professionals can ensure their analytical pipelines operate on the minimal necessary dataset, leading to significantly reduced execution times and lower resource consumption on the Spark cluster.

To summarize the best approach: use `select()` when you know precisely which few columns you require (inclusion), and use `drop()` when you know precisely which few columns you must remove (exclusion). Both methods strictly adhere to **Spark's** core principles of immutability and lazy evaluation, guaranteeing that the original source data remains untouched while generating a clean, new **DataFrame** tailored precisely to immediate analytical needs.

By implementing these powerful and efficient subsetting techniques, you elevate your data manipulation skills from basic handling into advanced, optimized PySpark development. Consistent and judicious application of these column management strategies is a hallmark of high-quality, production-ready big data code, paving the way for scalable and cost-effective data solutions.

ARABPSYCHOLOGY.COM