# How to Create a New DataFrame in PySpark from an Existing One

Authored by

**stats writer**

February 10, 2026

## Introduction to Data Transformation in PySpark

In the contemporary landscape of **data engineering** and **data science**, the ability to manipulate large-scale datasets efficiently is a foundational skill. PySpark, the **Python API** for Apache Spark, serves as a premier tool for this purpose, offering a robust framework for processing **big data** across distributed clusters. One of the most fundamental operations a developer performs is creating a new DataFrame from an existing one. This process is not merely a copy-and-paste action but involves the application of **transformations** that redefine the data's structure, content, and utility. By leveraging the built-in functions within the **pyspark.sql** module, users can derive specialized datasets tailored for specific **machine learning** models or analytical reports.

The transformation process typically involves isolating relevant information while discarding noise. This can be achieved through various high-level operations such as selecting specific columns, filtering rows based on logical predicates, or performing complex **aggregations**. Furthermore, the PySpark **API** allows for the integration of **user-defined functions** (UDFs), which enable custom logic to be applied to every row within a DataFrame. Once the transformation logic is defined, the resulting object can be assigned to a new variable or committed to a persistent storage system like **Hadoop Distributed File System** (HDFS) or **Amazon S3**. This flexibility ensures that data workflows remain modular and reproducible, which is essential for maintaining production-grade **data pipelines**.

Efficiency in Apache Spark is largely driven by its execution engine, which treats DataFrame operations as a series of instructions to be optimized. When you create a new DataFrame from an existing one, you are essentially building a **directed acyclic graph** (DAG) of computations. This **lazy evaluation** model means that the data is not actually moved or transformed until an **action**, such as `show()` or `write()`, is called. Consequently, understanding how to structure these transformations using methods like `select()` and `drop()` is vital for optimizing performance and minimizing **resource consumption** in a distributed environment.

## PySpark: Create New DataFrame from Existing DataFrame

## Methodologies for Deriving New DataFrames

When working with PySpark, developers frequently encounter scenarios where an original dataset contains an excessive amount of information. To streamline processing, there are two primary and highly effective ways to create a new DataFrame from an existing one, depending on whether you prefer to whitelist or blacklist specific attributes:

**Method 1: Specify Columns to Keep From Existing DataFrame**

This approach involves explicitly identifying the columns that are necessary for the next stage of

your analysis. By using the `select()` method, you instruct the **Spark engine** to project only the chosen fields into the new DataFrame. This is often the preferred method when you are moving from a wide table to a narrow one for **exploratory data analysis**.

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

### Method 2: Specify Columns to Drop From Existing DataFrame

Conversely, the `drop()` method is useful when you wish to retain the majority of the existing **schema** but want to remove a few specific columns, such as **personally identifiable information** (PII) or redundant metadata. This method effectively "drops" the specified columns and keeps everything else, which is highly efficient for cleaning datasets with hundreds of features.

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

The following examples demonstrate the practical application of these methods using a sample dataset within a **SparkSession** environment. These examples assume you have a running cluster or a local development environment configured for PySpark.

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

## Example 1: Selective Projection of Data Columns

In many **data science** workflows, feature selection is critical. We can utilize the following syntax to create a new DataFrame that isolates only the **team** and **points** columns. This operation is technically known as a **projection** in **relational algebra**, and it helps in reducing the memory footprint of the dataset during intensive computations.

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

Upon execution, you will observe that the resulting object is a distinct entity containing only the requested attributes. The **conference** and **assists** columns are excluded, streamlining the data for targeted analysis of scoring performance. This method is highly scalable and forms the basis for more advanced **SQL**-like queries within the Apache Spark ecosystem.

## Example 2: Streamlining via Column Exclusion

In scenarios where a dataset contains a multitude of features and only a few are irrelevant, the `drop()` function is the most pragmatic choice. For instance, if the **conference** classification is not required for a specific statistical model, we can generate a new DataFrame that excludes just that single attribute while preserving the integrity of all other data points.

**#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 output confirms that the new DataFrame successfully omits the **conference** column while retaining the **team**, **points**, and **assists** values. This approach is particularly beneficial when dealing with **schema** evolution, where new columns might be added to a source, and you want your transformation logic to remain robust by only specifying what to remove.

**Note**: While the provided example focuses on removing a single column, the `drop()` function is quite versatile. You can exclude multiple columns simultaneously by passing a list of names or providing them as comma-separated arguments. This capability allows for complex data cleaning tasks to be expressed in a single, readable line of code, adhering to the **clean code** principles of Python development.

## Understanding Immutability and Transformations

A crucial concept to grasp when working with Apache Spark is the principle of **immutability**. Unlike standard **Python** lists or Pandas DataFrames, where you might modify an object in-place, a PySpark DataFrame cannot be changed once it is created. This design choice is fundamental to the **fault tolerance** of Spark. Because the data is distributed across many nodes in a cluster, allowing in-place modifications would create massive synchronization overhead and risk **data inconsistency**. Therefore, every transformation you apply--be it a selection, a filter, or a join--

strictly results in a brand-new DataFrame object.

This immutable nature is what allows Spark to track the **lineage** of a dataset. If a portion of the data is lost due to a node failure, the system can refer back to the original source and re-apply the recorded transformations to reconstruct the missing pieces. When you write `df_new = df.select('team')`, you are not duplicating the data in memory immediately. Instead, you are defining a new logical step in the **lineage** graph. This is a key distinction between **distributed computing** and local data processing, as it emphasizes the management of logic over the movement of physical bytes.

Moreover, this paradigm encourages a functional programming style. By treating data as a series of immutable snapshots, developers can easily debug their **data pipelines**. You can inspect `df`, `df_new`, and any subsequent versions at various points in your code without worrying that a later operation accidentally corrupted an earlier state. This leads to more predictable and maintainable codebases, especially when working on complex **ETL** (Extract, Transform, Load) processes that involve multiple stages of refinement and **data validation**.

## Performance Optimization and Lazy Evaluation

The efficiency of creating new DataFrames in PySpark is significantly enhanced by **lazy evaluation**. In many other programming environments, a command is executed as soon as it is encountered. In Spark, however, transformations are "lazy." When you call `select()` or `drop()`, the **Catalyst Optimizer** simply records the operation in a logical plan. It does not actually scan the data or perform any computation. This allows Spark to look at the entire chain of transformations and optimize them as a single unit before execution. For example, if you select three columns and then immediately filter the rows, Spark can combine these steps to minimize the amount of data read from disk.

This optimization is part of the **Query Execution** process, which transforms a logical plan into a physical plan. The **physical plan** determines how the data will be partitioned and distributed across the executors in the cluster. By delaying execution until an action like `count()` or `collect()` is triggered, Spark can avoid unnecessary **shuffling** of data across the network. **Shuffling** is often the most expensive part of a distributed job, so the ability to optimize transformations before they happen is a massive performance advantage for **big data** tasks.

Furthermore, understanding the difference between **narrow transformations** and **wide transformations** is essential for high-performance PySpark development. Operations like `select()` and `drop()` are considered narrow transformations because they do not require data to be moved between different nodes; each partition can be processed independently. In contrast, operations like `groupBy()` or `join()` are wide transformations that require a **shuffle**. By

prioritizing narrow transformations and creating new DataFrames efficiently, you ensure that your **Spark jobs** run as quickly and cheaply as possible.

## Advanced Data Manipulation Techniques

Beyond the basic `select()` and `drop()` methods, PySpark offers a suite of advanced functions to create new DataFrames with enhanced data. The `withColumn()` method is perhaps the most common way to create a new DataFrame by adding a new column or replacing an existing one. This is often used for **type casting**, mathematical calculations, or applying string manipulations. For example, you might create a new column that calculates the ratio of points to assists, providing deeper insights into player efficiency. This method follows the same immutable pattern, returning a new DataFrame while leaving the original untouched.

Another powerful tool is the `filter()` or `where()` method, which creates a new DataFrame containing only the rows that meet a certain condition. Combined with selection, filtering allows you to create highly specific subsets of data. For instance, you could create a new DataFrame that only includes teams from the 'East' conference with more than 10 points. These operations can be **chained** together in a single statement, such as `df.select('team', 'points').filter(df.points > 10)`, which is both readable and performant thanks to the aforementioned **Catalyst Optimizer**.

Finally, the **Spark SQL** interface allows users to create new DataFrames using standard **SQL** queries. By registering a DataFrame as a temporary view, you can use the `spark.sql()` function to write complex queries that involve joins, subqueries, and **window functions**. The result of `spark.sql()` is always a new DataFrame, which seamlessly integrates back into the PySpark **API**. This hybrid approach allows developers to choose the most expressive or efficient syntax for their specific task, whether it is programmatic DataFrame methods or declarative **SQL** statements.

## Persisting and Exporting Your Results

Once you have successfully created a new DataFrame and applied all necessary transformations, the final step is often to persist that data for future use. In Apache Spark, you have several options for saving your results. You can write the DataFrame to a variety of file formats, including CSV, **JSON**, and highly optimized columnar formats like **Apache Parquet** or **Avro**. Using the `write` property of the DataFrame, you can specify the output path, the format, and the **save mode** (such as overwrite or append).

Choosing the right storage format is critical for the performance of downstream applications. **Parquet**, for example, is highly recommended for **big data** because it stores metadata about the **schema** and provides efficient compression. When you read a Parquet file back into PySpark, the

system can use **predicate pushdown** to only read the necessary columns and rows, significantly speeding up the creation of the next DataFrame in your pipeline. This creates a virtuous cycle of efficiency throughout your **data architecture**.

In addition to file-based storage, you may also choose to **cache** or **persist** a new DataFrame in memory. This is particularly useful if you plan to use that specific DataFrame in multiple subsequent actions. By calling `df_new.cache()`, you instruct Spark to store the data in the **RAM** of the executors after the first time it is computed. This prevents the system from having to re-calculate the entire **lineage** every time the DataFrame is accessed, which can save a tremendous amount of time in iterative **machine learning** algorithms or complex analytical workflows.