How to Select Multiple Columns in PySpark: A Step-by-Step Guide

Authored by

stats writer

February 11, 2026

Introduction to Data Selection in the PySpark Ecosystem

The PySpark framework has revolutionized the way data engineers and scientists interact with **Big Data** by providing a high-level API in **Python** that leverages the power of Apache Spark. At the heart of this ecosystem lies the DataFrame, a distributed collection of data organized into named columns. Selecting specific columns from these datasets is not merely a common task but a fundamental operation required for data cleaning, **feature engineering**, and optimizing memory usage during **distributed computing** tasks. By narrowing down the scope of data to only the necessary attributes, users can significantly reduce the overhead on the **Spark cluster** and improve the performance of downstream processing steps.

In the realm of Data Analysis, the ability to efficiently subset data is critical. When dealing with datasets containing hundreds or thousands of features, the **select** function serves as the primary tool for **projection**, which is a relational algebra operation that chooses specific attributes from a relation. PySpark provides multiple syntactical approaches to perform this action, catering to different scenarios such as hard-coded column selection, dynamic list-based selection, and positional slicing. Understanding these methods is essential for writing clean, maintainable, and efficient **Spark** applications that can scale to petabytes of information across a Computer cluster.

Moreover, the **PySpark DataFrame** is designed with **Lazy Evaluation** in mind. This means that when you invoke a **select** operation, the framework does not immediately execute the data retrieval. Instead, it records the operation in a **Logical Plan**. This plan is later optimized by the Catalyst Optimizer, which can perform **projection pushdown** to ensure that only the required columns are read from the underlying storage, such as **Apache Parquet** or **HDFS**. Consequently, mastering column selection is not just a syntax requirement but a core practice for building high-performance **Data Pipelines**. This guide will explore three robust methods to select multiple columns, providing **code examples** and technical context for each.

Select Multiple Columns in PySpark (With Examples)

There are three common ways to select multiple columns in a PySpark DataFrame:

Overview of Column Selection Methodologies

When working within the PySpark environment, developers often encounter different requirements for

data subsetting. The most straightforward approach is Explicit Selection, where column names are passed directly as strings. This is ideal for exploratory Data Science tasks where the **Schema** is well-known and static. However, for Automated Workflows, a more dynamic approach is often required. This is where List-Based Selection becomes invaluable, allowing for the programmatic generation of column lists based on metadata or specific filtering criteria. Finally, Index-Based Selection offers a positional method, which is particularly useful when the exact names of columns are less important than their relative order in the dataset.

Each of these methodologies has its own place in a robust ETL (Extract, Transform, Load) process. For instance, using strings is highly readable and makes the code self-documenting for other developers. On the other hand, utilizing Python lists and the unpacking operator allows for highly flexible code that can adapt to changing input **schemas** without manual intervention. Positional slicing, while less common in production environments due to the risk of schema shifts, is a powerful tool during initial data inspection.

By mastering all three, you ensure that your PySpark toolkit is versatile enough to handle any data engineering challenge.

Method 1: Select Multiple Columns by Name

#select 'team' and 'points' columnsdf.select('team', 'points').show()

Method 2: Select Multiple Columns Based on List

#define list of columns to select
select_cols =

#select all columns in list
df.select(*select_cols).show()

Method 3: Select Multiple Columns Based on Index Range

#select all columns between index 0 and 2 (not including 2)
df.select(df.columns).show()

Initializing the SparkSession and Defining the DataFrame

Before demonstrating the selection techniques, we must establish a working environment by initializing a **SparkSession**. The SparkSession is the entry point to programming Spark with the Dataset and DataFrame API. In a typical production environment, this session would connect to a cluster manager like YARN or Kubernetes. For demonstration purposes, we use the builder pattern to create a local session. This setup allows us to define a small dataset consisting of sports-related data, which we will use to illustrate how different columns can be isolated and manipulated.

The creation of a DataFrame involves combining raw data--often in the form of a list of lists or a list of tuples--with a defined list of column names. This process effectively imposes a **Schema** on the unstructured or semi-structured data. Once the DataFrame is created, it is immutable, meaning any operation like select will return a new DataFrame rather than modifying the original one. This immutability is a core principle of Functional Programming that **Apache Spark** adopts to ensure fault tolerance and consistent state across a distributed system. The following code snippet demonstrates the initialization of the session and the

construction of our primary dataset.

The following examples show how to use each method in practice with the following PySpark DataFrame:

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

Method 1: The Explicit Selection of Columns by String Literals

The most intuitive way to select multiple columns in PySpark is to provide the column names as comma-separated String arguments within the select() method. This approach is highly readable and is the standard for most basic SQL-like queries. When you pass strings to the select function, PySpark internally converts these strings into Column objects. This method is particularly effective when the required columns are few and their names are unlikely to change during the execution of the script. It provides a clear Data Lineage that is easy for other team members to follow during code reviews.

In the provided example, we target the "team" and

"points" columns. By executing df.select('team', 'points'), we instruct the Spark engine to create a new DataFrame projection. While the original dataset contains four columns, the resulting object will only carry the metadata and data for the two specified fields. It is important to note that the order in which you list the strings determines the order of the columns in the output DataFrame. This allows users to not only filter columns but also reorder them in a single operation, which is useful for preparing data for specific Machine Learning models or report exports.

Beyond simple strings, the select method can also accept Column objects explicitly (e.g., df.team or col('team')). However, using string literals is often the most concise syntax for simple selection tasks. This method is highly compatible with the PySpark SQL module, and it mirrors the SELECT statement found in standard SQL. For developers transitioning from traditional relational databases to Big Data platforms, this familiarity reduces the learning curve significantly. Below is the practical implementation of this method on our sample dataset.

Example 1: Select Multiple Columns by Name

We can use the following syntax to select the team and points columns of the DataFrame:

```
#select 'team' and 'points' columns
df.select('team', 'points').show()

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

Notice that the resulting DataFrame only contains the team and points columns, just as we specified.

Method 2: Leveraging List Unpacking for Dynamic Selection

As Data Engineering pipelines become more complex, there is often a need to select columns

programmatically. For instance, you might want to select all columns that start with a certain prefix or all columns that represent Numerical data types. In such cases, hard-coding strings is inefficient. Instead, you can define a Python list containing the desired column names and pass it to the select method. However, because the select function expects multiple arguments rather than a single list object, we must use the Python asterisk (*) operator to unpack the list into individual arguments.

This "unpacking" technique is a powerful feature of the Python language that **PySpark** developers use extensively. By using df.select(*select_cols), the list is expanded so that each element is treated as a separate input to the function. This makes your code much more Modular. You can define your list of columns at the beginning of your script or even load it from a configuration file, allowing the same logic to be applied to different datasets with varying structures. This approach is a cornerstone of Generic Programming within the Spark ecosystem.

Furthermore, this method facilitates easier

maintenance. If the set of required columns changes, you only need to update the list definition rather than hunting through multiple select statements throughout your codebase. It also allows for the use of List Comprehensions to filter columns based on their names or data types stored in df.schema. For example, one could easily create a list of all columns except for those containing sensitive information, thereby enhancing Data Privacy and security within the Data Lake. The example below illustrates the basic usage of list-based selection.

Example 2: Select Multiple Columns Based on List

We can use the following syntax to specify a list of column names and then select all columns in the DataFrame that belong to the list:

```
#define list of columns to select
select_cols =

#select all columns in list
df.select(*select_cols).show()

+----+------+
|team|points|
```

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

Notice that the resulting DataFrame only contains the column names that we specified in the list.

Method 3: Utilizing Index Ranges for Column Slicing

In certain scenarios, particularly during Exploratory Data Analysis (EDA), you might prefer to select columns based on their position within the DataFrame. PySpark provides access to the column names via the df.columns attribute, which returns a standard Python list. Because this is a standard list, we can use Python Slicing notation to select a range of columns. For instance, df.columns will return a list containing the first and second column names. This list can then be passed into the select method to retrieve the corresponding data.

Positional selection is extremely useful when dealing with wide datasets where you want to grab the first few "identifier" columns or a specific block of features located in the middle of the **Schema**. However, developers should exercise caution with this method in production environments. If the upstream data source changes and columns are reordered, index-based selection might inadvertently grab the wrong data. Despite this risk, it remains a fast and effective way to slice data during the development phase or when working with fixed-format legacy files where column positions are guaranteed to be static.

The syntax df.select(df.columns) combines the attribute access of Spark with the powerful slicing capabilities of Python. Note that in this specific case, PySpark is intelligent enough to handle the list returned by the slice without requiring the explicit unpacking operator, although using the asterisk is still considered a best practice for clarity. This method highlights the seamless integration between the Spark engine and the Python language, allowing for concise and expressive data manipulation code. The following example demonstrates how to select the first two columns using

this positional approach.

Example 3: Select Multiple Columns Based on Index Range

We can use the following syntax to specify a list of column names and then select all columns in the DataFrame that belong to the list:

```
#select all columns between index positions 0 and 2 ( not including 2)
df.select(df.columns).show()
```

```
+----+----------+
|team|conference|
+----+----------+
| A| East|
| A| East|
| A| East|
| B| West|
| B| West|
| C| East|
+----+----------+
```

Notice that the resulting DataFrame only contains the columns in index positions 0 and 1.

Performance Considerations and Strategic Best Practices

Selecting columns in PySpark is more than just a syntactic choice; it is a performance optimization strategy. In Big Data environments, I/O (Input/Output) is often the primary bottleneck. By selecting only the columns necessary for a specific computation, you enable Columnar Storage formats like Apache Parquet or ORC to skip reading irrelevant data from disk. This is known as Predicate Pushdown and Columnar Projection. Even if your source is a CSV file, reducing the number of columns in memory minimizes the Serialization overhead and reduces the pressure on the Java Virtual Machine (JVM) heap, which is where Spark executors perform their work.

When choosing between the three methods discussed, consider the Maintainability and Robustness of your code. For static scripts, Method 1 (names) is excellent. For reusable Data Pipelines, Method 2 (lists) is the industry standard because it allows for Dynamic Logic. Method 3 (indexing) should be used sparingly, primarily for interactive sessions or when the schema is guaranteed by a strict Data Contract. Additionally, always remember that PySpark columns are case-

sensitive or insensitive depending on the configuration of your SparkSession, so consistency in naming is vital to avoid runtime errors.

Finally, it is worth mentioning that for very large selections, you might want to explore the col function from pyspark.sql.functions. Using col() allows you to perform transformations--such as aliasing or casting--within the select statement itself. For example, df.select(col('team').alias('team_name')) provides more control than simple string selection. By combining these basic selection techniques with advanced functions, you can build highly efficient and readable Spark applications that handle massive datasets with ease. This concludes our deep dive into selecting multiple columns in **PySpark**.

Additional Resources for Mastery

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