Select Multiple Columns in PySpark (With Examples)

Authored by

stats writer

November 17, 2025

The Importance of Column Selection in PySpark

Efficient data manipulation is central to large-scale data processing, and selecting the right subset of columns from a massive DataFrame is often the first crucial step. In PySpark, selecting columns is necessary not only for filtering data display but also for optimizing performance by reducing the overall data footprint processed by the cluster. The primary mechanism for achieving this selective behavior is the powerful `select()` transformation, which returns a new DataFrame containing only the specified columns.

While simple selection involves naming a single column, practical data engineering scenarios frequently require selecting dozens or hundreds of columns simultaneously. PySpark offers robust, flexible methods to handle these complex selections, integrating seamlessly with underlying Python capabilities. Understanding these different approaches ensures that developers can choose the most readable and performant syntax for any given task, whether they are working interactively in a notebook or defining production-level ETL pipelines.

We will explore the three most common and effective ways to select multiple columns within a PySpark DataFrame. These methods vary based on whether you prefer naming columns explicitly, managing selections via standard Python lists, or leveraging positional indexing for dynamic range-based selection. Mastery of these techniques is essential for any professional working with distributed data processing environments.

Overview of PySpark Column Selection Techniques

The core of column selection in PySpark revolves around the `df.select()` function. This function accepts column names as string arguments or as column objects. When selecting multiple columns, how those arguments are passed determines the method used. Selecting only the necessary columns drastically reduces the amount of data shuffled and processed, which is critical for lowering execution times in distributed computing environments.

Here are the three fundamental methods we will detail, providing context and examples for each:

Direct Naming: Passing individual column names as separate string arguments to `df.select()`. This is the simplest method for a small, known number of columns, offering maximum clarity at the expense of scalability.

List Expansion: Defining a Python list of column names and using the splat operator (`*`) to unpack the list directly into the `select()` function arguments. This method enhances code cleanliness and is ideal for dynamic or lengthy selection requirements.

Index Range Selection: Utilizing Python's powerful slicing functionality on the DataFrame's

column names attribute (`df.columns`) to select columns based on their sequential position. This is useful when the exact names are unknown but their order is consistent.

Establishing the PySpark Environment and Sample Data

Before diving into the column selection methods, we must initialize a SparkSession and construct a sample DataFrame. All subsequent examples will operate on this structured dataset, which simulates typical tabular data found in analytical environments. This setup process ensures reproducibility and clarity throughout the demonstrations. The DataFrame we create represents performance data for several teams across different conferences.

We define our data rows and column headers explicitly before invoking `spark.createDataFrame()`. This foundational PySpark object allows us to execute complex distributed operations efficiently. It is crucial to use a consistent setup so that the results from each selection method can be accurately compared.

The resulting DataFrame contains detailed information about teams, their conferences, and performance metrics (points and assists), providing a clear context for our column selection exercises. The schema contains four columns: `team`, `conference`, `points`, and `assists`.

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

# Define the dataset containing team statistics
data = ,
,
,
,
,
]

Define the corresponding column names
columns =

# Create the PySpark DataFrame
df = spark.createDataFrame(data, columns)

# Display the initial structure of the 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: Direct Selection by Supplying Column Names

The most straightforward approach to selecting multiple columns involves passing the names of the desired columns as individual, comma-separated string arguments directly to the `df.select()` transformation. This method is highly intuitive and maintains excellent code clarity, especially when dealing with a small, fixed number of columns whose names are known prior to execution. It minimizes reliance on intermediate Python structures, making the transformation explicit and immediately traceable within the PySpark code block.

This technique is generally favored when the column selection is **static** or when dealing with interactive data exploration where quick, precise retrieval of specific fields is required. It reads almost like a natural language command: "select column A, column B, and column C." However, it becomes cumbersome and error-prone if the list of columns extends beyond five or six, as managing long lists of string literals inside a single function call can reduce readability and increase the chances of typographical errors.

For this example, we aim to isolate the `team` identification column along with the performance metric for `points`. We simply list these two column names as separate arguments inside the `select()` function, instructing PySpark to project only these two fields onto the resulting DataFrame.

 # Select 'team' and 'points' columns explicitly by name
df.select('team', 'points').show()

+----+------+
|team|points|
+----+------+
A	11
A	8
A	10
B	6

| B| 6|
| C| 5|
+----+------+

As demonstrated by the output, the resulting DataFrame successfully retains only the `team` and `points` columns, effectively dropping `conference` and `assists` from the projection. This confirms the successful application of the direct naming convention for multi-column selection.

Method 2: Leveraging Python Lists with Argument Unpacking

When the selection involves many columns, or when the list of desired columns is generated **dynamically** (e.g., loaded from a configuration file, derived from a schema analysis, or created through programmatic filtering), relying on a standard Python list is the preferred method. This approach involves defining all desired column names within a list variable and then using that list to populate the arguments of the `df.select()` function.

The key technical component that enables this is the splat operator (*). When placed before a list variable within a function call, the splat operator performs argument unpacking, treating each element of the list as a separate positional argument. Since `df.select()` is designed to accept multiple positional arguments (one for each column name), unpacking the list precisely fulfills the required syntax, making the transition from a Python list to a PySpark selection seamless.

This method significantly improves code maintainability and scalability. If the column requirements change, only the contents of the column list variable need to be modified, rather than updating a lengthy function signature. This is critical for scenarios where functional programming techniques are used to determine which columns should be included based on specific criteria (e.g., selecting all columns that match a certain regular expression pattern).

In the demonstration below, we create a list named `select_cols` containing `'team'` and `'points'`. We then call `df.select(*select_cols)`. The splat operator transforms into the equivalent of `df.select('team', 'points')` at runtime, achieving the same result as Method 1, but using a far more flexible programming pattern suitable for larger-scale operations.

 # Define list of columns to select
select_cols =

# Select all columns in list using the splat operator (*) for unpacking
df.select(*select_cols).show()

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

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

Confirming the efficacy of the list expansion technique, the output DataFrame mirrors the structure achieved in Method 1. This powerful method is the standard solution for professional PySpark development requiring dynamic column selection logic.

Method 3: Dynamic Selection Using Positional Indexing and Slicing

Occasionally, the requirement is to select a contiguous block of columns without needing to specify their exact names. This is especially relevant in scenarios where schema consistency is guaranteed, but the dataset contains many generic fields that must be preserved sequentially (e.g., selecting all columns between the ID field and the timestamp field). PySpark facilitates this through the use of standard slicing applied to the DataFrame's column attribute.

The attribute `df.columns` returns the full list of column names defined within the DataFrame as a native Python list of strings. Since this is a standard Python list, we can apply standard Python indexing rules to it. Slicing allows us to extract a subset of this list based on index positions, using the notation , where the `end` index is **exclusive**. The resulting slice is a new list of column names, which is then passed to `df.select()`.

This method is particularly useful for rapid prototyping or when the structure of the input file dictates a stable column order (e.g., legacy fixed-width files). However, users must exercise caution, as this method is susceptible to failures if the upstream schema changes its column order, resulting in incorrect data being selected silently.

For our demonstration, the columns are indexed sequentially: `'team'` is at index 0, `'conference'` is at index 1, `'points'` is at index 2, and `'assists'` is at index 3. To select the first two columns (`team` and `conference`), we use the slice . This selects indices 0 and 1, but excludes index 2.

 # Select all columns between index positions 0 and 2 (excluding index 2)
df.select(df.columns).show()

```
+----+----------+
|team|conference|
```

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

The output confirms that only the columns at positional indices 0 (`team`) and 1 (`conference`) have been retained in the resulting DataFrame. This method provides dynamic control over column selection based purely on sequence, offering a powerful tool when column ordering is stable across datasets.

Choosing the Optimal Column Selection Method

While all three methods successfully achieve the goal of selecting multiple columns, the choice of which method to implement should be guided by specific engineering criteria, including readability, flexibility for dynamic requirements, and reliance on positional stability. Data engineers must carefully assess the context of their transformation when deciding on the optimal syntax for longevity and maintainability.

For instance, Method 1 (Direct Naming) is unmatched in terms of explicit clarity; if a schema is rigid and only two or three specific columns are needed, this approach is the most efficient to read and debug. It requires minimal cognitive load for maintenance engineers reviewing the code. However, scalability is its major drawback. A transformation requiring 50 columns would result in an unmanageably long function call that violates clean code principles.

Method 2 (List Expansion) represents the ideal balance between clarity and flexibility. It cleanly separates the definition of the required columns from the execution of the transformation. This separation is vital for modular code design and is strongly recommended for production ETL environments where column lists are frequently derived from metadata or configuration settings. The explicit use of the splat operator clearly signals to the reader that a list is being unpacked into arguments, which is a powerful Python idiom.

Method 3 (Index Range Selection) should be used judiciously. Relying on column index positions introduces a dependency on schema order, which can be highly volatile in evolving datasets. If a downstream process relies on columns 0 and 1 being `team` and `conference`, the introduction of a new column at index 0 upstream would silently break the logic, leading to difficult-to-trace data quality issues. It is best reserved for controlled environments or highly standardized schemas

where positional stability is guaranteed, such as selecting all measure columns located after a fixed set of dimension keys.

Advanced Column Selection and Exclusion Techniques

Beyond simple selection, PySpark offers methods for exclusion and more complex pattern matching, which often rely on iterating or manipulating the list of columns obtained via `df.columns` before calling `df.select()`. Mastering the manipulation of the column list using native Python functionality is key to advanced transformations that go beyond fixed column lists.

One frequent requirement is dropping a specific column or set of columns. While PySpark provides a dedicated `df.drop()` method, achieving exclusion through selection involves creating a list of columns that specifically excludes the unwanted fields. For example, to select all columns except `'assists'`, one would first retrieve all column names, then use a list comprehension to filter out the undesired column, and finally apply Method 2: `all_cols = df.columns`, then `selected_cols = `, followed by `df.select(*selected_cols).show()`.

Furthermore, for large schemas where columns follow specific naming conventions, engineers frequently leverage Python's list comprehensions in conjunction with string methods (like `startswith()` or `endswith()`) to dynamically filter column lists. If we needed to select all columns that contain 'e' in their name, filtering against the `df.columns` attribute provides maximum power and flexibility:

 # Example of dynamic selection using filtering based on column name substring
filter_cols =
df.select(*filter_cols).show()

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

This powerful combination of native Python list manipulation and PySpark's `select()` function provides unlimited flexibility for tailoring data views to specific analytical needs, ensuring that

processing remains optimized by only handling relevant fields in distributed memory across the cluster.