How to Select Columns by Index in a PySpark DataFrame

Authored by

stats writer

February 11, 2026

Introduction to Column Selection in PySpark DataFrames

In the expansive ecosystem of **big data** processing, **PySpark** stands as a cornerstone for developers and data scientists who require a robust interface for **Apache Spark** using the **Python** programming language. One of the most fundamental operations performed during the **data transformation** phase is the manipulation of **DataFrame** columns. While selecting columns by their string names is the most common approach, there are numerous scenarios where selecting columns by their numerical index is far more efficient, especially when dealing with high-dimensional datasets where column names may be programmatic, repetitive, or unknown at runtime.

The process of selecting columns by index in a **PySpark DataFrame** involves a sophisticated understanding of how **Spark** manages its **schema** and metadata. Unlike traditional **relational databases**, a **DataFrame** in **Spark** is a **distributed** collection of data organized into named columns. By leveraging the `columns` attribute of the **DataFrame**, which returns a standard **Python** list of strings, users can utilize native **Python** list indexing and **slicing** techniques to identify specific positions. This bridge between **Python's** sequence handling and **Spark's** optimized execution engine allows for dynamic and flexible data manipulation workflows.

Choosing to work with indexes rather than hard-coded strings enhances the **modularity** of your code. For instance, if a data pipeline receives **CSV** files where the column headers change frequently but the positional order remains constant, index-based selection ensures the pipeline remains resilient to naming fluctuations. This method is particularly useful in **machine learning** pipelines where feature engineering steps often involve selecting a specific range of columns for vectorization or **scaling** without explicitly listing every feature name.

Furthermore, mastering these techniques contributes to cleaner and more **maintainable** codebases. By abstracting the selection logic through index references, developers can write generalized functions that apply to various **DataFrames** regardless of their specific **schema** details. In the following sections, we will explore the precise syntax and practical applications of selecting specific columns, excluding columns, and selecting ranges of columns using **PySpark's** powerful **select()** and **drop()** methods, ensuring you have the tools necessary to handle complex **data analysis** tasks with ease.

PySpark: Select Columns by Index in DataFrame

Methodologies for Positional Column Selection

In the realm of **PySpark**, several distinct strategies exist for accessing **DataFrame** columns based on their ordinal position. These methods rely on the fact that the `df.columns` property provides an

ordered list of all column identifiers within the current **distributed** dataset. By combining this list with the **PySpark API**, users can perform precise extractions. The following methodologies represent the most common and effective ways to achieve this:

Method 1: Isolating a Specific Column Using a Single Index

This approach is ideal when you need to extract a single attribute from a **DataFrame** based on its known position. By passing the index to the `columns` list, you retrieve the string name of the column, which is then passed to the `select()` function. This is highly useful for targeting specific **target variables** in a dataset.

#select first column in DataFrame
df.select(df.columns).show()

Method 2: Excluding Specific Columns via Indexing

In many **data preprocessing** tasks, it is easier to define what you do not want rather than what you do. The `drop()` method, when paired with an indexed reference from the `columns` list, allows you to remove unwanted data points effectively. This is often used to strip **identifiers** or index columns that are not relevant to **statistical analysis**.

#select all columns except first column in DataFrame
df.drop(df.columns).show()

Method 3: Retrieving a Sequential Range of Columns

When dealing with **time-series** data or datasets with grouped features, you may need to select a contiguous block of columns. **Python's** slice notation (start:stop) can be applied directly to the `columns` list. This returns a subset of column names that **PySpark** can then process in bulk, significantly reducing the verbosity of your **source code**.

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

Practical Implementation and Environment Setup

To demonstrate these techniques in a real-world context, we must first establish a **SparkSession**. The **SparkSession** serves as the entry point to all **Spark** functionality and is responsible for managing the underlying **cluster** resources. Once the session is active, we can construct a sample **DataFrame** representing a simple sports dataset to visualize the effects of our selection

operations.

The following code snippet initializes the environment, defines a structured **dataset** containing team names, conference affiliations, and points scored, and then instantiates the **DataFrame**. This setup provides a clear **visualization** of how data is organized before any transformations are applied, which is a critical step in **debugging** and verifying **Spark** logic.

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|
+----+----------+------+
A	East	11
A	East	8
A	East	10
B	West	6
B	West	6
C	East	5
+----+----------+------+

Detailed Walkthrough: Selecting a Specific Column

When you perform a selection using the index `0`, you are instructing **PySpark** to look at the first

entry in the **schema's** internal list of field names. In our specific example, the first column is "team". By executing `df.select(df.columns)`, you effectively translate a positional request into a name-based request that the **Spark Catalyst Optimizer** can understand and execute across the **cluster**.

This method is highly performant because it avoids the overhead of manual string entry and reduces the likelihood of **typos**. In a production **pipeline**, this level of automation is essential for maintaining **data integrity**. The resulting **DataFrame** will contain the same number of rows as the original but will be narrowed down to only the targeted attribute, as shown in the output below.

#select first column in DataFrame
df.select(df.columns).show()

```
+----+
|team|
+----+
| A|
| A|
| A|
| B|
| B|
| C|
+----+
```

Note that only the first column, which corresponds to the **team identifier**, has been successfully isolated. This granular control is the basis for more complex **data wrangling** procedures where only a subset of features is required for **exploratory data analysis**.

Excluding Data: Selecting All Columns Except a Specific Index

There are instances where a dataset contains a high volume of columns, and it is more practical to specify which columns to exclude rather than which ones to keep. By using the `drop()` function in conjunction with an index reference, **PySpark** creates a new **DataFrame** that retains all existing columns while omitting the one at the specified index. This is particularly useful for removing **sensitive information** like **Personally Identifiable Information (PII)** located at a specific index.

In the following demonstration, we target the first index for exclusion. This results in a **DataFrame** that maintains the "conference" and "points" columns while stripping the "team" information. It is important to remember that **DataFrames** in **Spark** are **immutable**; therefore, the `drop()` operation does not modify the original **DataFrame** but instead returns a new pointer to the transformed data

structure.

#select all columns except first column in DataFrame
df.drop(df.columns).show()

```
+----------+------+
|conference|points|
+----------+------+
| East| 11|
| East| 8|
| East| 10|
| West| 6|
| West| 6|
| East| 5|
+----------+------+
```

Observe that the **team** column has been removed, leaving only the remaining attributes. This technique is invaluable when performing **feature selection** where certain columns might introduce **multicollinearity** or are simply redundant for the analytical model being constructed.

Utilizing Index Slicing for Range-Based Selection

One of the most powerful features of using **Python** with **PySpark** is the ability to use **slicing**. Slicing allows you to define a range of indices to select multiple columns simultaneously. This is done using the `start:stop` syntax, where the `start` index is inclusive and the `stop` index is exclusive. This mirrors the behavior of **Python** lists and **NumPy** arrays, providing a familiar interface for data scientists.

In our example, using the slice `0:2` selects the columns at index 0 and index 1. This captures the first two columns of our **DataFrame**. This method is exceptionally efficient for processing **wide datasets** where you might want to extract the first 50 columns for one process and the next 50 for another, facilitating parallel **data processing** workflows.

#select all columns between index 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|
+----+----------+
```

The resulting output confirms that the "team" and "conference" columns were selected, while the "points" column at index 2 was omitted. This range-based selection is a key component in **batch processing** scenarios where data is partitioned not just by rows, but logically by groups of columns for specific **computational** tasks.

Best Practices and Performance Considerations

When selecting columns by index in **PySpark**, it is essential to consider the **readability** and **robustness** of your code. While indexing is powerful, it can lead to errors if the underlying data **schema** changes unexpectedly. To mitigate these risks, always ensure that your **data pipelines** include **validation** steps to verify the column order before performing index-based operations. This practice prevents logical errors that could propagate through your **analytics** layers.

From a **performance** standpoint, **Spark** is highly optimized for column-based operations. Because **Spark** uses a **columnar format** internally (via **Apache Parquet** or **Apache Arrow**), selecting specific columns--whether by name or index--allows the engine to perform **projection pushdown**. This means **Spark** will only read the necessary columns from the storage layer, drastically reducing **I/O** overhead and accelerating execution times.

To maintain high standards in your **PySpark** development, consider the following **best practices**:

Use Indexing for Dynamic Schemas: Apply index selection when column names are generated dynamically or are inconsistent across different **data sources**.
Document Index Logic: Clearly comment on why a specific index is being used, especially if the **DataFrame** contains dozens or hundreds of columns.
Combine with Schema Validation: Use `df.printSchema()` or `df.dtypes` to verify column positions during the **development phase**.
Prefer Slicing for Blocks: Use range slicing instead of multiple single-index selections to keep the code concise and readable.
Leverage Lazy Evaluation: Remember that selection operations are **lazy**; they are only executed when an **action** like `show()` or `collect()` is called, allowing **Spark** to optimize the overall **execution plan**.

Conclusion and Further Learning

Mastering column selection by index in **PySpark** is a vital skill for any data professional working with **big data**. By understanding the interplay between **Python's** list management and the **PySpark DataFrame API**, you can create more flexible, efficient, and **scalable** data processing workflows. Whether you are isolating a single feature, excluding irrelevant data, or selecting broad ranges of columns, these techniques provide the precision required for sophisticated **data science** projects.

As you continue to develop your **PySpark** expertise, exploring the nuances of **query optimization** and **memory management** will further enhance your ability to handle massive datasets. The ability to manipulate **DataFrames** programmatically via indexes is just one of many ways **Spark** empowers users to transform raw data into actionable insights with unparalleled speed and reliability.

The following tutorials explain how to perform other common tasks in **PySpark**, providing a comprehensive guide to mastering **distributed computing** and **cloud-based** data architectures: