Learning PySpark: Selecting DataFrame Columns by Index

Authored by

Mohammed looti

November 10, 2025

The Necessity of Index-Based Column Selection in PySpark

Working efficiently with large-scale, distributed datasets demands precise control over the data structure, or schema. In the realm of big data processing using PySpark, selecting columns based on their positional index rather than their explicit name is a powerful and often essential technique. This method proves invaluable in situations where column names are not static, such as when parsing generalized log files, handling raw CSV inputs without headers, or dealing with schemas that are generated dynamically during a data pipeline. Index-based selection guarantees flexibility and robustness in complex ETL processes.

Although best practices in PySpark generally advocate for referencing columns by name to maintain code clarity and stability, positional selection provides a critical fallback for programmatic operations. The foundation of this technique is simple: retrieve the list of column names from the DataFrame using the built-in `df.columns` attribute. This attribute returns a standard Python list, allowing us to leverage native Python list indexing and slicing to identify the specific column names we intend to include or exclude from the resulting DataFrame.

This guide details three primary and highly effective methods for manipulating a DataFrame's structure based on positional indices. These methods span requirements from isolating a single column to managing complex range-based selections or exclusions. Mastering these techniques is fundamental for developing adaptive and efficient data engineering workflows utilizing the Spark Python API.

Method 1: Selecting a Specific Column by Index

The most common requirement is often the selection of a single column based purely on its position within the schema. This is seamlessly achieved by accessing the `df.columns` list and applying a simple, **zero-based index**. The resulting column name string is then supplied to the `df.select()` transformation, which generates a new DataFrame containing only the requested field.

It is imperative to remember the principle of zero-based numbering inherent to Python lists. This means the first column resides at index 0, the second at index 1, and so forth. If you are targeting the fifth column of your schema, the correct index to use is 4. This indexing mechanism is standard across most programming environments and forms the basis for correctly applying all index-based selections in PySpark.

The primary benefit of this method is its reliability and precision. It guarantees that you select the column occupying that exact structural position, irrespective of any potential changes to the column's name. This feature is particularly beneficial in production environments where automated scripts rely on a stable incoming data structure. The code snippet below demonstrates the syntax required to select the very first column (index 0) of the target DataFrame:

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

Method 2: Excluding Columns by Positional Index

In many data preparation and feature engineering tasks, the requirement is to exclude a specific column while retaining all others. If the positional index of the column to be removed is known, the most straightforward and performance-optimized approach in PySpark is utilizing the `df.drop()` transformation.

Similar to the inclusion method, we first resolve the actual name of the column slated for removal by accessing the `df.columns` list using the correct index. Once the name is retrieved, it is passed directly into the `df.drop()` method. This approach is highly performant because it explicitly instructs the Spark execution engine which field to discard during transformation, avoiding unnecessary processing or serialization of that field.

This technique is particularly valuable when managing wide datasets. Specifying a long list of column names for inclusion can be cumbersome and error-prone. By contrast, simply identifying the one or two columns to exclude results in code that is far cleaner, more concise, and easier to maintain. The example below illustrates how to generate a new DataFrame containing all columns except for the first one (located at index 0):

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

Method 3: Selecting a Range of Columns Using List Slicing

When a task involves selecting a contiguous block of columns--such as isolating a sequence of feature columns or a block of metadata fields--Python's robust list slicing capability is essential. When applied to the `df.columns` attribute, slicing efficiently extracts a list of column names that fall within a defined positional range. This resulting list of names is then passed directly as arguments

to the `df.select()` method.

The standard syntax for Python list slicing is . It is crucial to internalize the rule that the `start` index is **inclusive**, while the `stop` index is **exclusive**. This exclusivity means that the column positioned at the stop index will not be included in the final selection. Therefore, if the goal is to include columns at indices 0, 1, and 2, the correct range specification must be .

This method offers remarkable flexibility. For example, omitting the starting index (e.g.,) selects all columns from the beginning up to index N-1, and omitting the stopping index (e.g.,) selects all columns starting from index N until the end of the DataFrame. This dynamic ranging significantly simplifies schema manipulation when working with data structures that frequently change size or order. The following example demonstrates how to select columns starting from index 0 up to, but specifically excluding, index 2:

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

Practical Implementation and Setup

To effectively illustrate the three index-based selection techniques, we must first establish a functional DataFrame environment in PySpark. This foundational setup involves initializing a `SparkSession`, defining the raw dataset, and explicitly specifying the column schema. This initial step is non-negotiable before attempting any structural transformations.

The sample data utilized here is intentionally simple, consisting of three distinct fields: `team` (assigned index 0), `conference` (assigned index 1), and `points` (assigned index 2). This structure provides a clear, visual mapping between the numerical index and the corresponding column name, which is essential for verifying the results of our index-based manipulations. The defined order in the `columns` list precisely dictates the numerical indices we will target throughout the subsequent examples.

The code block provided below handles the setup, initializing the Spark context and creating the sample DataFrame used for all demonstrations. It concludes by displaying the original data structure, which is crucial for confirming that the index selections perform as expected against the known schema.

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

Example 1: Isolating the First Column by Index

In this first demonstration, our objective is to isolate only the `team` column. According to our established schema, the `team` column is located at the zero index. We accomplish this by first fetching the list of column names via `df.columns` and then precisely indexing into that list at position . The resulting column name, 'team', is subsequently passed to the `df.select()` method.

The output clearly validates the successful application of this technique: the resulting DataFrame

maintains all original rows but is projected onto a new schema containing only the specified column. Utilizing `df.columns` followed by `.select()` is the standard, reliable pattern for dynamically selecting a single column based on its positional index in PySpark.

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

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

The resulting DataFrame confirms that only the first column (the **team** column) has been successfully isolated from the original structure.

Example 2: Excluding the First Column by Index

In sharp contrast to the previous example, we now employ the `df.drop()` method to remove the column located at index 0, which is the `team` column. This approach offers an efficient mechanism for data cleansing when a field's position is known but is no longer needed for subsequent analysis--for example, dropping a temporary metadata field or an initial identifier.

The `drop` transformation efficiently processes the data structure, returning a new DataFrame that omits the specified column entirely while preserving the order of the remaining fields. Because this operation is highly optimized within Spark, it is generally the superior method for column exclusion, offering much better readability and maintainability compared to constructing a complex selection list that attempts to include every column except the one to be discarded.

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

The resulting output confirms that all columns except the column at index 0 (the **team** column) have been successfully retained, leaving a structure containing only `conference` and `points`.

Example 3: Selecting a Range of Columns Using Slicing

Our final example leverages Python list slicing to select a consecutive block of columns. Our goal is to retrieve the `team` (index 0) and `conference` (index 1) columns. Given that Python slicing is end-exclusive, we define the range as . This specification ensures that the selection begins at index 0 and continues up to, but explicitly does not include, the column at index 2.

The true utility of slicing is evident here: we rely entirely on positional indices rather than manually listing 'team' and 'conference'. This is essential when dealing with wide datasets where hardcoding names is impractical. The list of names derived from the slice operation is automatically unpacked and passed as a sequence of arguments to the `df.select()` method, generating the desired output.

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

As intended, the output displays only the columns corresponding to indices 0 and 1, confirming the successful implementation of range-based selection using list slicing on the column list.

Advancing Beyond Index-Based Selection

The capability to dynamically manipulate a DataFrame's structure is a core competency for any data engineer working in PySpark. While selecting columns by index provides necessary automation and positional consistency, continuous mastery requires exploring more advanced transformations essential for cleaning, aggregating, and analyzing large distributed datasets.

To further optimize your skills within the Spark ecosystem and build upon this foundational knowledge of structure management, we recommend dedicating time to the following essential tasks and concepts:

Understanding and manipulating complex data types (e.g., **StructType**, **ArrayType**).

Implementing advanced filtering techniques using `where()` or `filter()` clauses for precise row selection.

Applying User Defined Functions (UDFs) to create custom column transformations for complex business logic.

Optimizing joins and aggregations across massive, partitioned datasets to maximize performance.

These advanced concepts build directly upon the ability to manage column selection and schema structure demonstrated in this guide, paving the way for developing robust and highly scalable data solutions.