PySpark: Select Columns by Index in DataFrame

Authored by

stats writer

November 17, 2025

The Necessity of Index-Based Column Selection

When working with large datasets in PySpark, efficient column management is fundamental to data preparation and analysis. While selecting columns by name is common, there are frequent scenarios--especially when dealing with schema changes, automated processing pipelines, or DataFrames with hundreds of columns--where selecting based on the numerical index becomes necessary. This approach leverages the underlying Python data structures used by the PySpark API.

The key to index-based selection in DataFrame operations lies in accessing the `.columns` attribute. This attribute returns a standard Python list containing the names of all columns in the order they appear in the DataFrame. By applying standard Python list indexing or slicing to this list, we can retrieve the required column name(s), which are then passed to PySpark transformation functions like `.select()` or `.drop()`.

Understanding zero-based indexing is crucial here. The first column is always accessed via index 0, the second via index 1, and so on. This article details three distinct and highly effective methods for performing index-based selection, ranging from isolating a single column to excluding or ranging multiple columns simultaneously.

Method 1: Isolating a Specific Column Using Indexing

The most straightforward requirement is selecting a single column when only its position (index) is known. We achieve this by accessing the `df.columns` list and specifying the desired index within square brackets (e.g., for the first column). This returns the column name as a string, which `.select()` then uses to filter the DataFrame.

This technique is particularly valuable when you need to extract the primary key or a specific identifier field whose position is fixed, regardless of the overall schema's column names. It provides resilience against changes in naming conventions across different data ingestion runs.

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

It is important to ensure that the specified index exists within the DataFrame's structure; attempting to access an index outside the valid range (e.g., index 5 in a 3-column DataFrame) will result in an `IndexError` from Python, halting the Spark job.

Method 2: Excluding Columns with the .drop() Function

Often, the goal is not to select a specific set of columns, but rather to drop one or two columns that

are not needed for analysis. The PySpark `.drop()` method is designed for this purpose. Unlike `.select()`, which explicitly names the columns to keep, `.drop()` names the column(s) to remove.

To exclude a column by its index, we again utilize `df.columns` to retrieve the specific column name string. We then pass this string directly into the `.drop()` function. This method is highly efficient for tasks like removing audit trails, metadata, or redundant identifier columns.

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

The resulting DataFrame retains all columns except the one specified by the index. If multiple columns need to be dropped by index, a list of column names (derived using slicing, as shown in Method 3) can be passed to the `.drop()` function.

Method 3: Selecting Sequential Columns via List Slicing

When dealing with a contiguous block of columns--such as sensor readings or temporal data fields grouped together--list slicing provides a concise way to select them. Python list slicing allows us to specify a starting index and an ending index (exclusive), returning a list of elements between those boundaries.

We apply this slicing directly to the `df.columns` list: `df.columns`. The result is a list of column names, which can be unpacked and passed to the `.select()` method. This is significantly cleaner than manually typing out every column name in a sequence.

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

Remember the fundamental rule of Python slicing: the range is inclusive of the start index but exclusive of the end index. In the example above, selects the column at index 0 and the column at index 1, but excludes the column at index 2. This precise control over column subsets is crucial for targeted feature engineering.

Setting Up the PySpark Environment and Sample DataFrame

To properly illustrate these indexing techniques, we must first establish a working PySpark environment and create a sample DataFrame. This setup ensures reproducibility and provides a concrete structure against which we can test our index selections. We define a simple dataset containing team performance metrics.

We initialize the `SparkSession`, define our raw data (a list of lists), and specify the column names: . This three-column structure means the indices are 0, 1, and 2, respectively.

The following code snippet demonstrates the necessary imports and the construction of our sample DataFrame, followed by a display of the resulting structure:

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

With this setup, we can confirm the index mapping: `team` is at index 0, `conference` at index 1, and `points` at index 2. We will use these indices in the following practical examples.

Practical Demonstration: Selecting a Single Column (Example 1)

This example directly implements Method 1. We aim to isolate the first column, `team`, which resides at index 0. We pass the result of `df.columns`--which is the string 'team'--to the `.select()` transformation.

The `.select()` function is additive; it builds a new DataFrame containing only the columns explicitly named. By using indexing, we dynamically supply the name of the column we wish to retain, ensuring the operation is robust even if the column names were to change, provided their order remains constant.

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

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

As observed in the output, only the first DataFrame column, `team`, has been successfully selected and displayed. This confirms the efficacy of using Python indexing on the `.columns` attribute for single-column retrieval in PySpark.

Practical Demonstration: Excluding a Column (Example 2)

In this demonstration of Method 2, we will exclude the first column (`team` at index 0) and retain the rest (`conference` and `points`). We achieve this by supplying the column name derived from `df.columns` to the `.drop()` function.

The `.drop()` operation is essential for cleaning datasets prior to modeling, particularly when certain identifiers might introduce bias or redundancy. Using index selection here streamlines the code, making the intention clear: remove the column at this specific position.

#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 DataFrame now contains only `conference` and `points`, demonstrating that all columns except the one corresponding to the 0th index were successfully retained. This illustrates a highly efficient way to manage column exclusion based on position.

Practical Demonstration: Utilizing Column Ranges (Example 3)

For the final example, applying Method 3, we demonstrate selecting a range of columns using Python list slicing. We want to select the first two columns (`team` and `conference`). Since Python slicing is exclusive of the end index, we specify the range , which captures indices 0 and 1.

The output of `df.columns` is the list . This list is then passed to the `.select()` function, which expects a list of column names when selecting multiple columns. This is a common requirement when subsets of related features need to be isolated.

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

We successfully selected the `team` (index 0) and `conference` (index 1) columns. The `points`

column (index 2) was excluded because the slice operation stops just before the end index specified. This ability to use list slicing significantly enhances the flexibility and readability of column selection logic in PySpark transformations.

Summary of Best Practices

Selecting columns by index in a DataFrame provides a powerful mechanism for automating data preparation, especially when column positions are more stable than column names. The core principle for all methods is leveraging the underlying `df.columns` Python list structure.

When implementing these methods in production code, consider the stability of your schema. If the order of columns is highly volatile, relying on index selection can lead to brittle code that selects the wrong data after a schema change. In such cases, selection by name is preferable. However, in environments where the data source dictates a fixed order (such as reading fixed-width files or specific file formats), index-based selection ensures concise and efficient code execution, minimizing lookups and increasing performance on the PySpark driver.

To summarize the techniques demonstrated:

Single Column Selection: Use `df.select(df.columns)`.
Column Exclusion: Use `df.drop(df.columns)`.
Range Selection: Use `df.select(df.columns)`, understanding the exclusive nature of the end index.

Mastering these techniques ensures you maintain high standards of flexibility and robustness when manipulating large-scale data structures in the Python-based Spark ecosystem.