Print One Column of a PySpark DataFrame

Authored by

stats writer

November 16, 2025

Introduction to PySpark and DataFrame Column Selection

As the leading PySpark programming model for large-scale data processing, the DataFrame is the cornerstone of modern data workflows. DataFrames provide a structured representation of data, enabling highly optimized operations via the Catalyst Optimizer. A common task in data exploration and transformation is isolating a single column, either for quick visual inspection or for extracting the raw values for use in traditional Python libraries or machine learning models.

When working with distributed datasets, simply accessing a column index is insufficient. We must employ specific methods provided by the PySpark SQL API to effectively project the desired column. This article details the two primary approaches for printing or extracting a specific column from a PySpark DataFrame, focusing on clarity, efficiency, and understanding the differences in the resulting output formats.

The choice between these methods hinges entirely on the final desired output format. If you require a neat, tabular display suitable for console output--complete with headers and alignment--you should use the first method. However, if the goal is to obtain a standard, iterable Python list containing only the raw data points, facilitating integration into non-Spark processing pipelines, the second method involving RDD transformations is necessary.

Setting Up the PySpark Environment and Sample Data

Before demonstrating the column extraction methods, we must initialize a SparkSession and construct a sample DataFrame. The SparkSession is the entry point to all functionality in Spark, and defining a clear dataset ensures that our examples are reproducible and easy to follow. We will use a small dataset representing team performance metrics across different conferences.

The dataset includes four columns: **team** (categorical identifier), **conference** (region), **points** (numerical score), and **assists** (numerical score). By creating this structured data, we simulate a typical scenario encountered in real-world data analysis where specific variables need to be isolated for inspection or downstream processing. The following code block initializes Spark and creates the demonstration 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|
+----+----------+------+-------+
```

This resulting DataFrame, stored in the variable `df`, serves as the foundation for our exploration. Note that PySpark automatically infers the data types, treating 'team' and 'conference' as strings and 'points' and 'assists' as integers, which is important for subsequent operations.

Method 1: Displaying a Column Using select().show()

The most straightforward approach to visualizing a single column within PySpark is by utilizing the combination of the select transformation followed by the `show()` action. The `select()` method is a fundamental operation that projects a subset of columns from the current DataFrame, generating a new DataFrame containing only those specified columns.

When you call `select('column_name')`, you are creating a narrow DataFrame with just one column. Subsequently, appending `.show()` triggers the computation required to execute the selection and formats the resulting data into a readable, truncated console output, which is highly beneficial for debugging or quick data checks.

To demonstrate this, we will isolate and display the values of the **conference** column. This method preserves the structural integrity of the output, maintaining the column header and the typical

Spark tabular display format, making it instantly recognizable as a subset of a distributed DataFrame.

#print 'conference' column (with column name)
df.select('conference').show()

```
+----------+
|conference|
+----------+
| East|
| East|
| East|
| West|
| West|
| East|
+----------+
```

Understanding the Output of select().show()

The output generated by `df.select('conference').show()` is a new table containing only the values for 'conference', along with the column's label. This tabular representation is often preferred for human readability, ensuring that the context of the displayed data is immediately clear. The `show()` action is designed to handle distributed data efficiently, fetching only a limited number of rows (defaulting to 20) and truncating long values, preventing console overflow.

It is crucial to understand that even though we are displaying the output in the console, the underlying structure remains a Spark DataFrame. This means the data is still distributed across the cluster until the `show()` action forces the computation and collection of the necessary results for local display. Furthermore, this method is idempotent in terms of data transformation; it does not alter the original DataFrame, only returning a view of the projected data.

If you needed to display more than the default number of rows, `show()` accepts parameters to control row limits and truncation (e.g., `df.select('col').show(n=100, truncate=False)`). However, for simple inspection of a single column's data points and ensuring the data types are correct, the basic `.show()` call is usually sufficient and highly performant, relying entirely on the optimized SQL engine for execution planning.

Method 2: Extracting Raw Values Using RDD Conversion

While the `.show()` method provides excellent console output, it often falls short when the

requirement is to obtain the data as a standard Python object, such as a list, for subsequent operations outside of the Spark ecosystem (e.g., passing data to a third-party library that expects standard Python iterables). In such cases, we must bypass the DataFrame's structured display and extract the underlying raw values, which requires converting the selected column back into a low-level RDD.

The standard syntax for extracting raw column values without the column name involves a four-step pipeline: first, selecting the column; second, converting to an RDD; third, flattening the results; and finally, collecting the distributed data back to the driver program. This sequence ensures that we strip away the DataFrame's metadata and structured Row containers, leaving only the atomic values.

We apply this detailed syntax to the **conference** column again. This demonstrates how to retrieve only the values as a plain Python list, ready for integration with native Python tools.

#print values only from 'conference' column
df.select('conference').rdd.flatMap(list).collect()

Deconstructing the RDD Transformation Pipeline

Understanding the transformation steps in Method 2 is crucial because it involves transitioning from the optimized SQL world of DataFrames back to the less-structured RDD framework. The process begins with `df.select(...)`, which, as discussed, creates a narrow DataFrame. The subsequent call, `.rdd`, performs the key conversion, transforming the DataFrame into an RDD of `Row` objects. Since we only selected one column, each `Row` object within this RDD contains a single element.

The most technical part of this chain is `.flatMap(list)`. The `flatMap()` transformation applies a function to every element of the RDD and then flattens the resulting structure. In this case, each element is a `Row` object, which behaves somewhat like a tuple or a list. By passing the native Python function `list` (or `lambda x: list(x)`) to `flatMap`, we instruct PySpark to convert the single-element `Row` into a single-element list, and then `flatMap` removes the surrounding wrapper, effectively "unboxing" the raw value from the `Row` container.

Finally, the `.collect()` action is executed. Like `show()`, `collect()` is an action that forces the distributed computation to run and gathers all the resulting data partitions back to the driver program's local memory, returning it as a standard Python list. A critical warning accompanies `collect()`: this method should only be used on RDDs or DataFrames small enough to fit comfortably in the driver's memory. Using it on massive datasets can lead to out-of-memory errors on the driver node.

Choosing the Right Method for Data Extraction

When deciding between `.select().show()` and `.select().rdd.flatMap(list).collect()`, the primary consideration should be the goal of the operation and the size of the data being processed. For operations primarily focused on debugging, data exploration, or visual confirmation of a transformation step, Method 1 is superior. It is faster for small checks, preserves Spark's optimized execution path (staying within the SQL API as long as possible), and handles large data volumes gracefully by only displaying a truncated sample.

Method 2, while more complex due to the conversion to RDD, is indispensable when the data must leave the distributed environment. Typical use cases include integration with libraries like NumPy or scikit-learn that require native Python data structures, generating reports where the output must be a clean, unformatted list of values, or writing the raw data points to a non-Spark system. However, this method carries the inherent risk of memory overflow if the number of records is too large for the driver node.

In summary, always default to `.select().show()` for inspection purposes. Reserve the RDD conversion method for production tasks where downstream systems explicitly require a flat Python list structure, ensuring that memory management concerns are addressed by filtering or aggregating the dataset beforehand if necessary. Efficiency in PySpark often means deferring local collection until absolutely necessary.