How to Check Column Data Types in a PySpark DataFrame

Authored by

stats writer

February 9, 2026

To check the data type of columns in a PySpark DataFrame, you can utilize the `dtypes` function (or attribute). This mechanism returns a list of tuples, where each tuple contains the name of the column and its corresponding data type. This provides a quick and efficient way to ascertain the data types of all columns in your DataFrame, which is essential for effective data exploration and subsequent transformation.

PySpark: Check Data Type of Columns in DataFrame

Core Methods for Checking Data Types

When dealing with schema validation in PySpark, two primary approaches exist for inspecting the data type of columns within a DataFrame. The selection between these methods depends on whether you need a comprehensive overview or targeted information on a single field.

Method 1: Checking All Columns

This approach uses the native `dtypes` attribute directly applied to the DataFrame object. This is the simplest and most commonly used method when performing initial data inspection or when auditing the entire schema after complex transformations. The output is a list of tuples detailing every column name and its corresponding inferred data type.

#return data type of all columns
df.dtypes

Method 2: Targeting a Specific Column

While `df.dtypes` returns a list of tuples, Python allows for seamless conversion of this structure into a dictionary format. By casting the output of `df.dtypes` to a dictionary, we enable rapid key-based lookup using the column name. This technique is highly efficient when analysts need to confirm the specific data type of just one field without iterating through the entire list.

#return data type of 'conference' columndict(df.dtypes)

Setting Up the Environment and Sample DataFrame

To demonstrate the utility and implementation of these methods, we must first establish a SparkSession and generate a representative sample DataFrame. This sample dataset simulates real-world data, including textual (string) fields and numerical (integer) fields, some of which contain null values. Understanding how PySpark infers data types from such mixed inputs is crucial

for accurate schema validation.

The setup involves importing the necessary `SparkSession` class, defining the tabular data as a list of lists, and specifying the column headers. When the DataFrame is created, PySpark automatically scans the data to assign the most appropriate data type to each column, a process known as schema inference.

The following code block executes the setup, creating and displaying our sample dataset. Observe the resulting output, particularly the handling of null values and the automatic type assignment:

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	null	8	9
A	East	10	3
B	West	null	12
B	West	null	4
C	East	5	2
+----+----------+------+-------+

Practical Application 1: Querying a Specific Column's Type

When performing targeted data quality checks or preparing a specific feature for modeling, the ability to quickly retrieve the data type for an individual column is invaluable. This avoids the manual effort of scanning a potentially long list of all data types. We achieve this efficiency by converting the result of `df.dtypes` into a Python dictionary, allowing O(1) complexity lookup based on the column name key.

Let us apply this dictionary lookup method to the **conference** column. Since this column contains categorical text data ('East', 'West', or null), we expect PySpark to classify it as a string type. The code snippet below confirms this assumption:

#return data type of 'conference' columndict(df.dtypes)

'string'

The output, `'string'`, confirms the appropriate data type for textual content. This is a critical validation step before applying string-specific functions or one-hot encoding techniques during feature engineering.

Similarly, we can verify the data type of the **points** column. As **points** contains whole numbers, PySpark typically infers this numerical field as a 64-bit integer, or **bigint**, by default, ensuring sufficient capacity for large values:

#return data type of 'points' columndict(df.dtypes)

'bigint'

The confirmation that **points** is a `'bigint'` is important for performance optimization, especially when performing large-scale numerical computations where precise control over integer size is required. Utilizing bigint ensures the numerical integrity of the data.

Practical Application 2: Checking All Columns Simultaneously

The most direct way to understand the full structure of the dataset is by invoking the `df.dtypes` attribute without any wrapping functions. This returns a comprehensive list that defines the schema, which is vital for communicating the data structure to other analysts or for programmatic schema checks within automated pipelines.

The output is a standard Python list containing two-element tuples. The first element of the tuple identifies the field name, and the second element identifies its inferred data type. This clarity

ensures immediate understanding of how Spark stores and handles the data.

Executing `df.dtypes` on our sample DataFrame provides the following structured output, revealing the types assigned to all four columns:

#return data type of all columns
df.dtypes

Interpreting the Full Output Schema

The structured list of tuples generated by `df.dtypes` allows for rapid interpretation of the DataFrame's contents. Each tuple provides specific insight into the storage format chosen by Apache Spark, ensuring compatibility with analytical operations. This detailed schema is essential for maintaining data type integrity across various stages of the data pipeline.

The **team** column has a data type of **string**, which is appropriate for categorical text identifiers.
The **conference** column also has a data type of **string**, necessary because it holds textual categories like 'East' and 'West'.
The **points** column is classified as **bigint** (64-bit integer), which is the standard inferred type for numerical integer fields in PySpark when no explicit schema is provided.
Similarly, the **assists** column is designated as **bigint**, allowing it to handle whole numbers efficiently.

This clear visibility into the schema is paramount for ensuring that joins, aggregations, and window functions operate correctly across columns with compatible data types.

Conclusion: Enhancing Data Quality and Analysis

Mastering the use of the `dtypes` attribute is fundamental for anyone working with PySpark DataFrames. Whether you require a high-level overview of the entire dataset schema or need to confirm the type of a single column before a critical transformation, these methods offer precision and speed.

By consistently verifying data types, engineers can proactively identify schema drift, enforce data integrity, and optimize performance within the Spark cluster. The simplicity of accessing the `dtypes` attribute ensures that schema inspection remains a quick, routine part of the data preparation workflow.

The following tutorials explain how to perform other common data manipulation and inspection tasks in PySpark: