How to Convert Multiple Columns to Different Data Types in PySpark Using cast()

Authored by

stats writer

February 3, 2026

Working with large datasets often requires rigorous data cleaning and transformation, especially when dealing with inconsistent or improperly inferred data types. The `cast()` function in PySpark is a powerful tool designed to manage this process efficiently. While it is commonly used for single column conversions, its true efficiency shines when applied to transforming the DataFrame schema for multiple columns simultaneously.

This approach significantly streamlines data manipulation tasks, eliminating the necessity of writing repetitive code for separate conversions. By defining a list of target columns and iterating through them, users can apply standardized type transformations, such as converting numerical fields to strings or vice versa. Mastering this technique is essential for effective large-scale data preparation in the Spark ecosystem.

Efficient Data Transformation using PySpark's cast()

In PySpark, the `cast()` function is implemented on a column object (`pyspark.sql.Column`) and is used to explicitly change the data type of that column. This is a non-mutating transformation; it returns a new column with the specified type, which is then typically used in conjunction with methods like `withColumn()` to update the DataFrame.

When dealing with many columns requiring the same type conversion, manually calling `withColumn()` and `cast()` for each field becomes tedious and error-prone. The most effective strategy involves defining a collection of columns and processing them iteratively, thus applying the transformation in a single, clean block of code.

The general methodology involves iterating over a list of column names. Inside the loop, we use `df.withColumn(x, col(x).cast('dataType'))`, where `x` represents the current column name, and `col(x)` is necessary to reference the column object itself before applying the `cast()` function method.

General Syntax for Multi-Column Casting

To implement simultaneous data type conversions across several fields, we leverage Python's iteration capabilities combined with PySpark's declarative functions. This process requires importing the necessary functions, defining the target columns, and then executing the iterative conversion logic.

Consider a scenario where numerical columns must be converted to strings for downstream processing, such as preparing data for specific API consumption or reporting. The following snippet illustrates the core syntax required to achieve this conversion for columns named **points** and **assists**:

my_cols =

for x in my_cols:
df = df.withColumn(x, col(x).cast('string'))

In this efficient loop, for every column name `x` listed in `my_cols`, the original `withColumn()` operation updates the DataFrame `df`. The transformation uses the `cast('string')` method applied to the column reference, ensuring that only the specified columns receive the new string data type while all other fields remain unchanged.

Practical Example: Setting Up the PySpark Environment and Data

To demonstrate this powerful multi-column casting technique, let us first establish a foundational DataFrame containing basketball statistics. This initial setup is crucial for simulating a real-world scenario where data ingestion might default numerical columns to generic integer types (like **bigint**), requiring later conversion for specific analytical tasks.

We begin by initializing a Spark session and defining our sample data, which includes categorical fields (**team**, **conference**) and quantifiable metrics (**points**, **assists**):

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 table, **df**, provides the necessary structure for our demonstration. We can now proceed to inspect the inherent data types that PySpark automatically inferred during the DataFrame creation process.

Initial Inspection of Data Types (`df.dtypes`)

Before performing any transformations, it is always best practice to verify the current schema. Understanding the initial data types is critical because incorrect types can lead to errors in subsequent analytical operations or output inaccuracies. We can use the `df.dtypes` attribute to display the current schema as a list of tuples (column name, data type).

Executing the following simple command provides a clear view of the underlying schema:

```
#check data type of each column
df.dtypes
```

From this output, we observe that the categorical columns, **team** and **conference**, were correctly inferred as **string**. However, the quantitative columns, **points** and **assists**, were inferred as **bigint** (a standard integer type in Spark). Our objective is to now explicitly convert these two numerical columns into the **string** data type using a single, unified method.

Implementing the Multi-Column Conversion Loop

Now that we have identified the target columns (**points** and **assists**) and their current types (**bigint**), we apply the iterative `cast()` function logic. This ensures a clean and repeatable conversion process, essential when managing dozens or hundreds of columns.

We first define the list of columns to modify and then loop through this list, applying the `cast('string')` transformation within the `withColumn` operation, updating the DataFrame in each iteration.

#specify columns to convert to different dataType
my_cols =

#convert dataType of each column in list to string
for x in my_cols:
df = df.withColumn(x, col(x).cast('string'))

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

It is important to note that while the underlying values remain numerically represented in the displayed output, the internal schema of the DataFrame has been updated. The conversion is successful without altering the visual presentation of the data itself.

Verification of Converted Data Types

The final crucial step is confirming that the iterative conversion process successfully modified the schema as intended. We must re-run the `df.dtypes` command to inspect the updated data types for all columns.

This verification confirms the success of the multi-column casting approach, demonstrating that the code efficiently transformed the numerical fields into strings while maintaining the integrity of the rest of the schema.

#check data type of each column
df.dtypes

The output clearly shows that both the **points** and **assists** columns have been successfully converted to the **string** data type, meeting the conversion objective. Crucially, the data types for

the **team** and **conference** columns remained unchanged, proving the targeted application of the conversion loop.

Conclusion: Streamlining PySpark Data Preparation

The ability to apply the `cast()` function across multiple columns simultaneously is a fundamental technique for enhancing efficiency in PySpark data preparation pipelines. By utilizing Python loops in conjunction with PySpark's declarative APIs, data engineers can manage complex schema changes rapidly and reliably.

This method drastically reduces boilerplate code, improves readability, and minimizes the risk of human error associated with manual conversion of many columns. Whether converting strings to numerical types for calculations or numbers to strings for standardized reporting, this iterative casting approach is highly scalable and recommended for production environments.

For those looking to deepen their expertise in data manipulation, exploring other advanced operations within PySpark is highly recommended. The following resources explain how to perform other common tasks crucial for effective data engineering in the Spark environment: