How to Cast Multiple Columns in PySpark with the `cast()` Function

Authored by

stats writer

January 18, 2026

Understanding Data Type Coercion in PySpark

Data transformation is a fundamental process in any large-scale data pipeline, and frequently involves ensuring that columns adhere to the necessary data type specifications for subsequent analysis or storage. In the context of big data processing using PySpark, the ability to accurately change the type of data stored within a column is critical for maintaining data integrity and enabling proper computation. Operations like joining datasets, filtering, or performing aggregations often fail silently or produce incorrect results if the underlying data types are mismatched or improperly formatted.

The standard method for modifying a column's data type within a PySpark DataFrame is through the utilization of the **cast() function**. This powerful built-in function allows developers and data engineers to explicitly coerce the data stored in a column from its current type (e.g., integer, float, or date) into a target type (e.g., string, boolean, or decimal). While casting a single column is straightforward, real-world data science tasks often require applying the same transformation logic across numerous columns simultaneously, necessitating a programmatic and efficient approach rather than manual, repetitive calls.

This article focuses on the advanced application of the **cast() function** in PySpark, specifically detailing how to effectively manage type conversions across multiple columns within a DataFrame. We will explore the common pitfalls of manual conversion, introduce an efficient iterative solution using Python loops, and provide a comprehensive, step-by-step example demonstrating this technique in a practical scenario, ensuring that your data transformation workflows are both robust and scalable.

The PySpark cast() Function Explained

The core of data type transformation in PySpark resides in the **cast() function**, which is available as a method on a PySpark Column object. When applied, this function attempts to convert the existing column values into the specified target data type. For instance, converting a numerical column containing identifiers into a string type is a common requirement before exporting data to certain systems or performing string-based operations. Conversely, converting string-represented numbers back into numerical types like **integer** or **float** is necessary before performing mathematical calculations.

Understanding the mechanism of the cast function is crucial. When you execute a cast operation, PySpark assesses whether the values in the source column can be reliably represented in the target data type. If the conversion is feasible (e.g., converting an integer '123' to a string '123'), the operation succeeds. However, if the conversion encounters an invalid value (e.g., trying to cast a non-numeric string like 'ABC' to an integer), the resulting value in the new column will typically be assigned a `null` value, depending on the Spark configuration and the data type involved.

Therefore, careful data profiling should precede any mass casting operation.

While the syntax for a single column conversion is simple--typically involving `df.withColumn('col_name', col('col_name').cast('target_type'))`--relying on this approach for dozens or hundreds of columns quickly becomes cumbersome, error-prone, and violates the principles of clean, maintainable code. This inefficiency mandates the use of dynamic programming techniques, specifically iterative methods, to handle large-scale transformations gracefully within the DataFrame structure.

Handling Multiple Columns: The Challenge and the Loop Solution

In complex datasets, it is common to find scenarios where multiple columns, often grouped logically, require the exact same data type conversion. For example, if a dataset contains ten separate measure columns (e.g., Q1_Sales, Q2_Sales, etc.) that were mistakenly imported as strings and need to be treated as decimals, applying the conversion individually is tedious. The primary challenge here is achieving efficiency and scalability without sacrificing readability.

The most effective and idiomatic PySpark solution for applying the **cast() function** across a list of columns involves leveraging a standard Python `for` loop in conjunction with the PySpark `withColumn` method and the `col` function. This approach allows us to define the specific columns that require modification in a list, and then iterate over that list, applying the transformation one column at a time within the same DataFrame variable. Since PySpark operations are lazy, this series of transformations is efficiently batched and executed by the Spark engine.

To implement this, we first define a Python list containing the names of all target columns. Inside the loop, for each column name `x` in the list, we call `df.withColumn(x, col(x).cast('target_type'))`. The `withColumn` method, when used with an existing column name, overwrites that column with the result of the expression provided. By continuously reassigning the transformed DataFrame back to the variable `df` (i.e., `df = df.withColumn(...)`), we chain the conversions, ensuring that each iteration builds upon the result of the previous one.

Detailed Syntax Walkthrough for Multi-Column Casting

The powerful synergy between standard Python iteration and PySpark column manipulation provides a concise and readable pattern for bulk type conversion. Let us examine the precise syntax required to execute this transformation. We must first define the list of columns targeted for casting, followed by the iteration block that applies the transformation using the **cast() function**.

Consider the scenario where we need to convert two numerical columns, `points` and `assists`, into string format. The following code snippet elegantly handles this requirement, illustrating the best practice for batch type coercion in PySpark:

my_cols =

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

In this code block, `my_cols` serves as the configuration list, clearly delineating which parts of the DataFrame are subject to change. The subsequent `for` loop iterates through this list. Inside the loop, `df.withColumn(x, ...)` is called, where `x` is the current column name being processed. The expression `col(x).cast('string')` is the core transformation logic, instructing Spark to treat the values in column `x` as strings. Crucially, this iterative structure ensures that all other columns in the DataFrame maintain their original dataType, isolating the change only to the specified columns.

Practical Example: Setting Up the PySpark Environment

To fully illustrate the mechanism of multi-column casting, we will establish a small PySpark environment and define a sample dataset relevant to sports statistics. This dataset will mimic common scenarios where numerical metrics might need to be converted to strings (perhaps for export into a JSON format or display purposes) or vice versa. We begin by initializing the Spark session, which is necessary to interact with the Spark cluster and utilize the DataFrame API.

We define our raw data, which includes player information, team affiliations, and two key performance indicators: `points` and `assists`. These metrics are initially defined as integers within the raw Python list structure. We then create the PySpark DataFrame using the `spark.createDataFrame` method, explicitly providing the column names. After execution, we display the DataFrame content using `df.show()` to confirm the structure and data validity.

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

The resulting DataFrame `df` is now ready for inspection. Before performing any transformations, it is a critical step in the data workflow to verify the current inferred schema. This verification step ensures we understand the starting data types, particularly for the columns we intend to convert, guaranteeing that our conversion strategy targets the correct columns with the expected initial types.

Initial Data Inspection and Schema Verification

Before applying the cast function, it is imperative to confirm the current schema of the DataFrame. PySpark attempts to infer the data types during DataFrame creation, but this inference might not always align perfectly with the developer's intent, especially when reading from raw files. The `dtypes` attribute of the DataFrame provides a quick and clean way to retrieve a list of tuples, where each tuple contains the column name and its corresponding data type.

Upon examining our newly created DataFrame, we utilize the `df.dtypes` command to check the inferred types for all four columns:

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

As shown in the output, the categorical columns (`team` and `conference`) have been correctly identified as **string** types. Crucially, the quantitative performance metrics, `points` and `assists`,

have been inferred as bigint (a large integer type), which is a common default for whole numbers in PySpark. Our goal in the next step is to convert these two **bigint** columns into the string type, demonstrating the effectiveness of the multi-column casting technique. This initial inspection verifies the current state and sets the stage for the required transformation.

Implementing the Multi-Column Transformation

With the initial schema verified, we can now proceed with the core objective: converting the data type of the `points` and `assists` columns from **bigint** to string using the iterative approach. This procedure involves defining the target list and employing the Python loop structure to apply the **cast() function** repeatedly, leveraging PySpark's optimization engine for efficient execution.

The following syntax encapsulates the entire transformation process. Notice the clarity provided by separating the list definition from the loop logic, a programming standard that enhances code maintainability. This structure ensures that if we later need to add or remove columns from the conversion set, only the `my_cols` list needs modification.

#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
+----+----------+------+-------+

After executing the loop, we immediately view the DataFrame using `df.show()`. Visually, the data content remains identical to the output shown previously, as converting a number to a string does not alter the displayed representation of the value. However, the critical change has occurred at

the structural level of the DataFrame schema. To confirm this successful conversion, we must proceed to the final verification step: checking the resultant data types.

Validating the Transformed Schema

The validation stage is essential to guarantee that the multi-column casting operation achieved the intended result. Simply viewing the DataFrame content is insufficient, as the display format often obscures the underlying dataType. We once again rely on the `df.dtypes` attribute to inspect the structure of the transformed DataFrame. This final check confirms whether `points` and `assists` are now correctly classified as **string** type columns.

We execute the schema check immediately after the transformation loop:

#check data type of each column
df.dtypes

The output unequivocally demonstrates the success of the bulk conversion. Both the `points` and `assists` columns, previously holding bigint types, are now recorded as **string** types. Furthermore, it is important to note that the data types for the `team` and `conference` columns, which were not included in the `my_cols` list, have correctly remained unchanged as **string** types. This confirms that the iterative approach is both targeted and non-disruptive to the remainder of the DataFrame structure.

Best Practices and Performance Considerations

While using a Python loop with the **cast() function** is the most common and readable method for batch conversions in PySpark, understanding best practices ensures optimal performance, especially when dealing with truly massive datasets. One key consideration involves using the `withColumn` operation: since each iteration of the loop creates a new DataFrame object, chaining these operations can sometimes lead to slightly less optimal query plans compared to defining all transformations simultaneously. However, for type casting, the iterative approach is generally acceptable due to its clarity.

For scenarios involving a very large number of columns (e.g., hundreds) or complex logic beyond simple casting, developers might consider creating a single expression list that can be passed to the `select` method. This alternative involves generating all the column transformation expressions upfront and then selecting the complete set of columns (both original and transformed) in a single operation. This ensures Spark sees all transformations simultaneously, potentially optimizing the physical plan more effectively than repeated `withColumn` calls. However, for straightforward tasks

like casting, the loop method demonstrated remains the most intuitive and maintainable pattern.

Ultimately, selecting the appropriate data type is paramount. Incorrectly casting data can lead to data loss (e.g., casting a float to an integer truncates decimal parts) or runtime errors (e.g., casting incompatible string data to a numerical type). Always use the `cast() function` judiciously, preceded by thorough exploratory data analysis and schema inspection, to ensure that the data transformation aligns with both the analytical goals and the intrinsic properties of the data.

Here are some other common tasks and tutorials explaining how to perform them in PySpark:

Merging DataFrames based on specific key columns.
Filtering rows using complex conditional logic.
Performing window functions for rolling calculations.
Optimizing data partitioning for improved query performance.