How to Select Numeric Columns in PySpark DataFrames

Authored by

stats writer

January 20, 2026

Understanding Data Type Selection in PySpark DataFrames

When working with large-scale data processing using PySpark, it is frequently necessary to isolate columns based on their data type for specific analytical or transformation tasks. For instance, statistical operations like correlation matrices or aggregations are typically only meaningful when applied to **numeric data types**. Selecting all numeric columns efficiently is a foundational skill for data engineers and scientists utilizing the **PySpark framework**.

The standard mechanism for filtering columns involves inspecting the PySpark data types assigned during schema inference or definition. Since numeric types--such as `bigint`, `double`, `float`, and `integer`--often need to be treated separately from categorical or textual data (`string`), we leverage the DataFrame's schema information. The most concise and idiomatic Pythonic approach to finding these columns involves using a **list comprehension** combined with the DataFrame's `dtypes` attribute.

Below is the direct syntax used to identify and select only the columns within a PySpark DataFrame that contain **numeric data**. This method is highly efficient as it processes the schema metadata rather than the underlying data itself, ensuring fast execution even on massive datasets.

 # Find all numeric columns by iterating through the schema (c = column name, t = data type)
numeric_cols =

# Select only the identified numeric columns in the DataFrame using the unpacked list
df.select(*numeric_cols).show()

Practical Implementation: Setting Up the PySpark Environment

To fully illustrate this technique, we will walk through a complete example, starting with the initialization of a SparkSession and the creation of a sample DataFrame. This step ensures we have a structured data environment where the schema can be accurately inspected and manipulated. The example DataFrame models statistical information about basketball players, containing a mix of textual (categorical) and **numeric data types**.

In this scenario, we define five distinct columns: `team` and `position` (expected to be strings), and `points`, `assists`, and `minutes` (expected to be numeric). Properly setting up the data structure is crucial, as the accuracy of our column selection depends entirely on how PySpark infers or assigns the initial data types to these fields.

The following code block demonstrates the necessary steps to define the data, specify the column names, create the DataFrame using the `spark.createDataFrame` method, and display the

resulting data structure using `df.show()`.

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

# Define sample basketball player data (mixed types)
data = ,
,
,
,
,
,
,
]

Define descriptive column names
columns =

# Create the DataFrame using the data and column names
df = spark.createDataFrame(data, columns)

# View the DataFrame content and structure
df.show()

+----+--------+------+-------+-------+
|team|position|points|assists|minutes|
+----+--------+------+-------+-------+
A	Guard	11	4	22.4
A	Guard	8	5	34.1
A	Forward	22	6	35.1
A	Forward	22	7	18.7
B	Guard	14	12	20.2
B	Guard	14	8	15.6
B	Forward	13	9	20.9
B	Center	7	9	4.8
+----+--------+------+-------+-------+

Inspecting DataFrame Schemas and Data Types

Before we can accurately filter the columns, we must confirm the data types that PySpark has assigned to each column. This is achieved by accessing the `df.dtypes` attribute, which returns a

list of tuples. Each tuple contains two elements: the column name (a string) and its inferred PySpark data type (also a string). Understanding this structure is essential for applying the subsequent filtering logic using **list comprehension**.

In the example below, we expect `team` and `position` to be `string` types, while `points` and `assists`, being whole numbers, are inferred as `bigint` (a type of numeric data type), and `minutes`, containing decimal values, is inferred as `double`. The filtering logic must correctly identify all these numeric variations while excluding the textual types.

Executing the `df.dtypes` command provides the necessary metadata required for programmatic column selection:

 # Display data type of each column in the DataFrame
df.dtypes

Generating the List of Numeric Column Names

The core mechanism for selecting numeric columns relies on iterating over the `df.dtypes` list and applying a conditional filter. We use a powerful Python feature known as list comprehension to create a new list containing only the names of the columns whose data type does not start with `'string'`. This simple inverse check effectively captures all **numeric data types**, including `int`, `float`, `double`, `bigint`, and `decimal`, while excluding standard textual columns.

The condition `t.startswith('string') == False` evaluates to true for any data type `t` that is not a string. This approach is highly flexible and handles variations in numeric type inference seamlessly, whether PySpark chooses `int`, `long`, `bigint`, or `double` based on the data provided. The resulting list, `numeric_cols`, holds exactly the column names we intend to select.

Executing the following syntax extracts the column names and verifies the filtered result:

 # Find all numeric columns in DataFrame using list comprehension
numeric_cols =

# View list of numeric columns
print(numeric_cols)

Applying the Selection Logic to the DataFrame

Once the list of desired column names (`numeric_cols`) has been successfully generated, the final

step is to apply this list to the PySpark DataFrame using the `.select()` transformation. The crucial detail here is the use of the **asterisk operator (*)** before the list name. This operator unpacks the list, treating each element (column name) as a separate argument passed to the `select` function.

The `.select()` method is a fundamental operation in PySpark, projecting a new DataFrame that contains only the specified columns. By dynamically generating the column list based on data type, we create a robust and scalable method that adapts automatically to changes in the underlying data schema, rather than requiring manual specification of column names. This is especially useful in ETL pipelines where schema drift is a concern.

The resulting DataFrame will contain only the columns that hold **numeric data types**, allowing for immediate subsequent analysis, such as calculating descriptive statistics or performing mathematical transformations.

 # Select only numeric columns in DataFrame using the unpacked list
df.select(*numeric_cols).show()

```
+------+-------+-------+
|points|assists|minutes|
+------+-------+-------+
| 11| 4| 22.4|
| 8| 5| 34.1|
| 22| 6| 35.1|
| 22| 7| 18.7|
| 14| 12| 20.2|
| 14| 8| 15.6|
| 13| 9| 20.9|
| 7| 9| 4.8|
+------+-------+-------+
```


Analysis of the Selection Results

The output confirms that the resulting DataFrame successfully isolated the `points`, `assists`, and `minutes` columns. These were the three fields in our initial data structure identified by PySpark data types as `bigint` and `double`, which fall under the category of **numeric data types**. The textual columns, `team` and `position`, were correctly excluded because their data type string matched the filter condition `t.startswith('string')`.

This confirms the efficiency and accuracy of using schema introspection for large-scale column management in PySpark. By relying on the metadata provided by `df.dtypes`, we avoid potentially

slow operations that might involve sampling or checking actual cell values, thereby ensuring that the selection process remains computationally lightweight, regardless of the dataset size.

Why Filtering by 'string' Data Type is Effective

The selection technique demonstrated here hinges on the simple yet highly effective use of the `startswith('string')` filter applied inversely. In PySpark, virtually all non-numeric, categorical, or free-text data types are classified as `string`. Conversely, all standard numerical data types, such as `int`, `long`, `float`, `double`, `decimal`, and `bigint`, have data type names that do not begin with 'string'.

Therefore, filtering for columns where the data type does **not** start with 'string' provides a comprehensive way to capture all desired numeric data types without having to explicitly list every possible numeric variation (e.g., `t.startswith('int') or t.startswith('double')...`). This approach reduces code complexity and maintains compatibility across various Apache Spark versions and underlying JVM types.

It is important to note that this method assumes standard PySpark data types are used. While edge cases exist (e.g., highly specialized user-defined types), for typical analytical workloads, excluding types starting with 'string' is the most reliable and concise method for identifying all numerical columns within a **PySpark DataFrame** structure.

Summary of PySpark Selection Techniques

The robust and scalable approach outlined utilizes Python's expressive power alongside PySpark's schema metadata access:

Schema Inspection: Accessing `df.dtypes` provides quick, metadata-level insight into the column names and their associated types (e.g., ('points', 'bigint')).

Conditional Filtering: A Python list comprehension is used to iterate through these schema tuples, applying the inverse string filter (`t.startswith('string') == False`) to capture all **numeric data types**.

Dynamic Selection: The resulting list of column names is unpacked (using `*`) and passed directly to the `df.select()` method for efficient DataFrame projection.

This method is highly recommended for developers seeking to automate data preprocessing steps and ensure that subsequent statistical models or aggregations operate exclusively on appropriate **numeric columns**, leading to cleaner and more reliable data pipelines.

Related PySpark Tutorials and Resources

The following tutorials explain how to perform other common data manipulation and analysis tasks in PySpark, building upon the foundational knowledge of DataFrame creation and column selection:

How to filter rows based on conditions in PySpark.

Performing aggregations and grouping data in a PySpark DataFrame.

Understanding schema definition and type casting in PySpark.