How to Select PySpark Columns Containing a Specific String

Authored by

stats writer

February 5, 2026

PySpark: Select Columns Containing a Specific String

Introduction to Dynamic Column Selection in PySpark

Working with extensive and evolving datasets in PySpark necessitates tools for dynamic data manipulation. One of the most common requirements during data preparation or feature engineering is the ability to select a subset of columns whose names match a particular pattern, such as containing a specific keyword or prefix. While the standard `select` method excels at handling static lists of columns, filtering columns based on a dynamic string match requires a powerful, programmatic approach that efficiently combines Python's native capabilities with the robust processing environment of Spark.

This capability is absolutely crucial when managing wide tables where manual specification of columns is not only time-consuming but also prone to error, especially as schema changes over time. For instance, if a vast PySpark DataFrame contains dozens of related metrics--like `sales_2020_q1`, `sales_2021_q1`, `inventory_level`, and `inventory_cost`--pattern matching allows developers to instantly isolate only the

'sales' or 'inventory' related fields. The most straightforward, pythonic, and performant way to accomplish this in PySpark is by utilizing Python's built-in list comprehension feature on the `df.columns` attribute.

Although some introductory documentation might suggest using SQL-like operators such as `like` for generalized column selection, for manipulating column metadata (the names themselves), the list comprehension approach is superior. It ensures that the column filtering is handled entirely in the fast, client-side Python environment before the request is handed over to the distributed Spark engine for projection, thereby maximizing efficiency and minimizing overhead.

The Core Technique: Leveraging List Comprehension

The foundation of effective column selection based on a substring relies on inspecting the column names array and filtering it dynamically. The `df.columns` attribute provides a standard Python list of all column names, making it perfectly suited for native Python filtering techniques.

The following syntax illustrates the canonical method used to select only columns that contain a specific string within a PySpark DataFrame. This particular example selects all columns that contain the substring 'team':

```
df_new = df.select()
```

This single, highly readable line of code executes a powerful operation. It iterates through every column name (represented by `x`) in the list of all column names (`df.columns`) and applies a conditional filter (`if 'team' in x`). This filter checks for the presence of the substring 'team' anywhere within the column name. The output of the list comprehension is a new, filtered Python list containing only the matching column names. This resulting list is then passed directly as arguments to the select function, instructing Spark to project the data onto this reduced set of columns.

Understanding the Mechanics of Column Filtering

To fully appreciate the efficiency of this technique, it is vital to distinguish between the two stages of processing. The first stage involves pure Python

execution: the program accesses the column metadata and uses string matching logic. This process is instantaneous, regardless of the size of the underlying distributed data, because it only interacts with the schema information stored on the driver node.

The conditional logic `if 'substring' in x` is a simple, case-sensitive string containment check. For instance, if the search term is 'id', it will match columns like `user_id` and `transaction_id`, but not `ID_number` (due to case sensitivity). Developers can easily adjust this logic--for example, by using `if 'substring'.lower() in x.lower()`--to implement case-insensitive matching if necessary.

The second stage begins when the resulting list of column names is supplied to the `df.select()` method. The select function is a transformation in Spark, meaning it generates a new logical execution plan. Spark then executes this plan across the cluster, ensuring that only the data corresponding to the requested columns is retained for subsequent operations, thus minimizing memory footprint and improving downstream processing speed.

Setting Up the Practical PySpark Demonstration

To provide a concrete example of this technique, we will first set up a sample PySpark DataFrame. This DataFrame will contain information about basketball players, where some column names are intentionally structured to contain the target substring 'team', while others are not. This setup allows us to clearly demonstrate the effectiveness of the filtering logic.

The process begins with initializing a SparkSession, which is the entry point for using Spark functionality. We then define our raw data and explicitly specify the column names. Note that we define `team_name` and `team_position`, both containing the substring 'team', alongside `player_points` and `assists`.

The following code snippet details the complete setup, resulting in the source DataFrame `df`:

```
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_name|team_position|player_points|assists|

+---------+-------------+-------------+-------+

| A| Guard| 11| 4|

| A| Forward| 8| 5|

| B| Guard| 22| 6|

| A| Forward| 22| 7|

| C| Guard| 14| 12|

| A| Guard| 14| 8|

| B| Forward| 13| 9|
| B| Center| 7| 9|
+---------+-------------+-------------+-------+

Executing the Substring Selection

We now apply the dynamic selection logic to the previously created `df`. Our goal is to extract only those columns containing the substring 'team', discarding the columns related to specific player statistics (points and assists). This execution vividly demonstrates the power and simplicity of filtering based on metadata patterns.

The list comprehension first checks the columns . The filter `if 'team' in x` matches the first two elements and excludes the latter two. The resulting list is then used for the projection.

Review the code and output below to confirm that the resulting DataFrame, `df_new`, contains only the columns related to the team:

```
#select columns that contain 'team' in the name
df_new = df.select()

#view new DataFrame
```

```
df_new.show()

+---------+-------------+
|team_name|team_position|
+---------+-------------+
| A| Guard|
| A| Forward|
| B| Guard|
| A| Forward|
| C| Guard|
| A| Guard|
| B| Forward|
| B| Center|
+---------+-------------+
```

The output confirms that the dynamic selection was successful, yielding a new **DataFrame** with only `team_name` and `team_position`. This approach is highly scalable and ensures that the selection logic remains valid even if the underlying schema changes, provided the naming convention is consistently followed.

Combining Dynamic and Explicit Column Selection

In real-world data pipelines, it is frequently necessary to

select columns based on a pattern while simultaneously retaining one or more essential columns that do not fit that pattern. For instance, we might need all 'team' columns (dynamically selected) and the `assists` column (explicitly required) for a specific calculation.

Because the result of the pattern match is a standard Python list of column names, we can easily use Python's list concatenation operator (`+`) to append the list of explicitly required columns. This maintains the conciseness of the dynamic selection while providing the necessary flexibility for combining selection strategies.

To select all columns containing 'team' plus the `assists` column, we structure the select function call by concatenating the list generated by the list comprehension with a single-element list containing `'assists'`:

#select columns that contain 'team' in the name and the 'assists' column
df_new = df.select(+)

#view new DataFrame

```
df_new.show()

+---------+-------------+-------+
|team_name|team_position|assists|
+---------+-------------+-------+
| A| Guard| 4|
| A| Forward| 5|
| B| Guard| 6|
| A| Forward| 7|
| C| Guard| 12|
| A| Guard| 8|
| B| Forward| 9|
| B| Center| 9|
+---------+-------------+-------+
```

The final output confirms the inclusion of all dynamically matched columns and the explicitly added `assists` column, demonstrating a powerful and highly adaptable methodology for column filtering in **PySpark**.

Advanced Filtering Using Regular Expressions

While the simple `in` operator handles basic substring matching flawlessly, complex data transformation tasks may necessitate more sophisticated filtering, such as

selecting columns that match one of several possible patterns, or excluding columns based on a specific suffix. For these scenarios, Python's built-in `re` module, which handles regular expression processing, can be seamlessly integrated into the list comprehension.

For example, if you wanted to select columns that start with 'team' (using regex `^team.*`) or exclude columns that end with '_temp' (using `re.search('.*_temp$', x) is None`), you would import the `re` module and incorporate its functions into the conditional statement of the list comprehension. This approach maintains the high performance of client-side metadata processing while unlocking the full power of regular expression matching.

It is important to remember that using Python's `re` module here is distinct from using Spark SQL functions like `regexp_extract`. The former acts on the column names (metadata) before execution, while the latter operates on the data values themselves during the distributed execution phase. For column selection based on names, the metadata approach is always preferred.

Conclusion and Best Practices Summary

Dynamic column selection based on substring matching is a fundamental technique for writing robust and adaptive PySpark code. The recommended best practice involves using Python list comprehension to filter the `df.columns` list and passing the result to the `df.select()` select function. This method ensures that column filtering is performed efficiently at the metadata level, minimizing overhead and maximizing the performance of the subsequent distributed operations.

Key takeaways for efficient implementation include:

Use standard Python string operators (`in`) for simple substring matches.
Employ the `+` operator on lists for combining dynamically matched columns with explicitly named columns.
For complex pattern matching, integrate Python's `re` module within the list comprehension filter.
Always prefer this metadata-based selection over attempting to use distributed functions for column name filtering.

The following tutorials explain how to perform other common tasks in PySpark: