How to Extract Substrings in PySpark: A Step-by-Step Guide

Authored by

stats writer

February 6, 2026

Extract Substring in PySpark (With Examples)

The ability to accurately parse and manipulate string data is a cornerstone of modern **data engineering**, especially when dealing with large, complex datasets. **PySpark**, the Python API for **Apache Spark**, provides powerful and distributed tools for these tasks. Extracting a **substring**--a contiguous sequence of characters within a string--is a common requirement for data cleaning, feature engineering, and transformation workflows.

This comprehensive guide will detail the various functions PySpark offers for string manipulation, focusing specifically on how to efficiently extract desired segments of text from columns within a **DataFrame**. We will explore methods based on positional indexing (start and length) and those based on delimiter boundaries, providing clear code examples for each technique.

Understanding String Operations in PySpark

Working with string data in a distributed environment like Spark requires using specialized functions optimized for parallel processing. Unlike standard Python string methods, PySpark's operations are executed across the entire cluster, making them scalable for **big data** volumes. PySpark incorporates a wide array of functions within the `pyspark.sql.functions` module designed specifically for column-level transformations.

When extracting substrings, the primary goal is often data standardization. For instance, if you have a column containing full URLs, you might need to extract only the domain name; or if you have a column with concatenated IDs (e.g., `User-12345-RegionA`), you might need to isolate just the numeric user ID. PySpark facilitates this by providing functional programming constructs that allow users to apply transformation logic directly to column data, thereby avoiding inefficient row-by-row processing.

To successfully extract a substring, you must first ensure your data is loaded into a PySpark **DataFrame**. The core functions available for this task include `substring()`, `substr()`, `substring_index()`, and `regexp_extract()`. While `substring()` and `substr()` handle fixed-length extraction based on position, `substring_index()` and `regexp_extract()` offer more dynamic extraction capabilities based on specified delimiters or pattern matching rules.

Prerequisites for Substring Operations

Before initiating any string manipulation in PySpark, it is crucial to import the necessary components. All string manipulation functions reside within the `pyspark.sql.functions` module, conventionally imported as `F`. This alias simplifies the syntax for calling functions like `F.substring()` or `F.substring_index()` when defining new columns via the `withColumn`

transformation. Ensuring this import is correctly handled is the first step in any script involving column-based textual analysis.

The standard process involves taking an existing **DataFrame** (`df`) and applying the string extraction logic using `.withColumn()`. This transformation creates a new column populated with the extracted substrings without mutating the original data structure. If you intend to replace the original column, you would simply use the same column name in the `withColumn` call, but best practice usually dictates creating a new column for verification purposes.

You can use the following methods to extract certain substrings from a column in a PySpark **DataFrame**. These methods primarily rely on the `substring` function for positional extraction and `substring_index` for delimiter-based extraction.

Method 1: Extract Substring from Beginning of String (Positional)

This technique uses the `F.substring()` function, specifying a starting index of 1 (PySpark uses 1-based indexing for strings) and the desired length of the resulting substring. This is highly useful when extracting fixed prefixes, such as department codes or state abbreviations.

from pyspark.sql import functions as F

#extract first three characters from team column
df_new = df.withColumn('first3', F.substring('team', 1, 3))

Method 2: Extract Substring from Middle of String (Positional)

To extract a segment from the middle of a string, you must specify the exact starting position (index) and the total number of characters (length) to include in the output. This is essential for fields where important identifiers are sandwiched between fixed-length prefixes and suffixes.

from pyspark.sql import functions as F

#extract four characters starting from position two in team column
df_new = df.withColumn('mid4', F.substring('team', 2, 4))

Method 3: Extract Substring from End of String (Positional)

PySpark's `substring()` function supports negative indexing to extract characters relative to the end of the string. By setting the starting index to a negative number (e.g., -N), you instruct the function to begin counting N characters from the right end, moving leftwards, and then specifying

the length of the segment to extract.

from pyspark.sql import functions as F

#extract last three characters from team column
df_new = df.withColumn('last3', F.substring('team', -3, 3))

Method 4: Extract Substring Before Specific Character (Delimiter-Based)

The `F.substring_index(str, delimiter, count)` function is utilized for extraction based on delimiters. When `count` is a positive integer (e.g., 1), it extracts the content located to the left of the Nth occurrence of the specified delimiter. Setting `count` to 1 extracts everything before the very first occurrence of the delimiter.

from pyspark.sql import functions as F

#extract all characters before space in team column
df_new = df.withColumn('beforespace', F.substring_index('team', ' ', 1))

Method 5: Extract Substring After Specific Character (Delimiter-Based)

Conversely, setting the `count` parameter to a negative integer (e.g., -1) in the `F.substring_index()` function instructs PySpark to extract the content located to the right of the Nth occurrence of the delimiter, counting from the right end of the string. Using -1 extracts the content after the last occurrence of the delimiter.

from pyspark.sql import functions as F

#extract all characters after space in team column
df_new = df.withColumn('afterspace', F.substring_index('team', ' ', -1))

Advanced Extraction Techniques: Leveraging Regular Expressions

While positional and delimiter-based methods cover many common scenarios, complex or highly variable string patterns often require the flexibility of **regular expressions**. PySpark offers the `F.regexp_extract()` function, which allows users to define sophisticated patterns to match and capture desired text segments. This method is invaluable when dealing with semi-structured log data or when extracting multiple fields embedded within a single string.

The `regexp_extract()` function takes three primary arguments: the column name, the **regular**

expression pattern, and the capture group index (starting from 0 for the entire match, and 1 or higher for specific captured groups). Mastering regular expressions, while challenging, unlocks the most advanced capabilities for string data cleaning and normalization within the PySpark environment.

Setting Up the Example DataFrame

To illustrate the practical application of the methods described above, we first need to establish a working PySpark **DataFrame**. This DataFrame, containing basketball team names and associated points, will serve as the source data for all subsequent extraction examples. We initialize a `SparkSession` and then define the data and column schema to create the distributed dataset.

The following code block sets up the environment and displays the initial structure of our sample data. Note the use of `SparkSession.builder.getOrCreate()` to manage the Spark context, ensuring our transformations run correctly.

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|points|
+--------------+------+
| Dallas Mavs| 18|
| Brooklyn Nets| 33|

| Atlanta Hawks| 12|
|Boston Celtics| 15|
| Miami Heat| 19|
|Cleveland Cavs| 24|
| Orlando Magic| 28|
+--------------+------+

Example 1: Extract Substring from Beginning of String

In this example, we aim to extract the first three characters from the `team` column. This mimics extracting a standardized abbreviation or code. We use the `F.substring()` function, starting at position 1 and requesting a length of 3 characters. This is the simplest form of positional extraction.

The resulting new column, named `first3`, clearly shows the initial three letters of each team name. This demonstrates the efficiency and straightforward syntax of PySpark's column transformation methods for basic text truncation.

from pyspark.sql import functions as F

#extract first three characters from team column
df_new = df.withColumn('first3', F.substring('team', 1, 3))

#view updated DataFrame
df_new.show()

+--------------+------+------+
| team|points|first3|
+--------------+------+------+
Dallas Mavs	18	Dal
Brooklyn Nets	33	Bro
Atlanta Hawks	12	Atl
Boston Celtics	15	Bos
Miami Heat	19	Mia
Cleveland Cavs	24	Cle
Orlando Magic	28	Orl
+--------------+------+------+

Example 2: Extract Substring from Middle of String

To extract characters embedded within the team name, we adjust the start position. Here, we start

at position 2 and extract four characters. Remember that PySpark strings are 1-indexed, meaning position 2 is the second character in the string. This operation is useful for extracting codes that always appear after a fixed prefix.

Observe the output: for "Dallas Mavs", starting at position 2 ('a') and taking four characters results in 'alla'. This precision highlights the importance of correct index counting when performing positional substring extraction.

from pyspark.sql import functions as F

#extract four characters starting from position two in team column
df_new = df.withColumn('mid4', F.substring('team', 2, 4))

#view updated DataFrame
df_new.show()

+--------------+------+----+
| team|points|mid4|
+--------------+------+----+
Dallas Mavs	18	alla
Brooklyn Nets	33	rook
Atlanta Hawks	12	tlan
Boston Celtics	15	osto
Miami Heat	19	iami
Cleveland Cavs	24	leve
Orlando Magic	28	rlan
+--------------+------+----+

Example 3: Extract Substring from End of String

Extracting from the end of a string is often necessary when isolating extensions, file types, or, in this case, the last few letters of a team abbreviation. By using a negative index (-3), we instruct the function to start counting three positions from the end of the string and then extract the next three characters.

The code extracts 'avs' from 'Dallas Mavs' and 'ets' from 'Brooklyn Nets'. This robust feature of `substring()` eliminates the need for calculating the total length of the string before extraction, simplifying the logic for tail segments.

from pyspark.sql import functions as F

```
#extract last three characters from team column
df_new = df.withColumn('last3', F.substring('team', -3, 3))

#view updated DataFrame
df_new.show()

+--------------+------+-----+
| team|points|last3|
+--------------+------+-----+
| Dallas Mavs| 18| avs|
| Brooklyn Nets| 33| ets|
| Atlanta Hawks| 12| wks|
|Boston Celtics| 15| ics|
| Miami Heat| 19| eat|
|Cleveland Cavs| 24| avs|
| Orlando Magic| 28| gic|
+--------------+------+-----+
```

Example 4: Extract Substring Before Specific Character

When string data is separated by a known delimiter (like a space, comma, or hyphen), using `substring_index()` is more reliable than positional extraction, as it adapts to variable-length strings. In this example, we use a space (' ') as the delimiter and set the count to 1. This captures all content before the first space encountered.

For team names structured as "City Name Team Name" (e.g., Dallas Mavs), this effectively isolates the City Name by stopping at the first space. This method is highly effective for splitting concatenated fields.

```
from pyspark.sql import functions as F

#extract all characters before space in team column
df_new = df.withColumn('beforespace', F.substring_index('team', ' ', 1))

#view updated DataFrame
df_new.show()

+--------------+------+-----------+
| team|points|beforespace|
+--------------+------+-----------+
| Dallas Mavs| 18| Dallas|
```

| Brooklyn Nets| 33| Brooklyn|
| Atlanta Hawks| 12| Atlanta|
|Boston Celtics| 15| Boston|
| Miami Heat| 19| Miami|
|Cleveland Cavs| 24| Cleveland|
| Orlando Magic| 28| Orlando|
+--------------+------+-----------+

Example 5: Extract Substring After Specific Character

To extract the remainder of the string following a delimiter, we use a negative count argument in `substring_index()`. Setting the count to -1 extracts everything after the last occurrence of the delimiter, which in our case is the team's nickname following the city name.

This allows us to isolate the 'Mavs', 'Nets', and 'Hawks' components of the strings, providing the team nickname. This is a crucial technique for normalizing data where the prefix (city) might vary in length but the suffix (nickname) is needed for analysis.

from pyspark.sql import functions as F

#extract all characters after space in team column
df_new = df.withColumn('afterspace', F.substring_index('team', ' ', -1))

#view updated DataFrame
df_new.show()

+--------------+------+----------+
| team|points|afterspace|
+--------------+------+----------+
Dallas Mavs	18	Mavs
Brooklyn Nets	33	Nets
Atlanta Hawks	12	Hawks
Boston Celtics	15	Celtics
Miami Heat	19	Heat
Cleveland Cavs	24	Cavs
Orlando Magic	28	Magic
+--------------+------+----------+

Conclusion: Versatility in PySpark String Manipulation

PySpark offers a highly efficient and versatile suite of functions for extracting **substrings**, catering to both rigid positional requirements and flexible delimiter-based partitioning. Whether you are dealing with large-scale data cleansing, feature engineering for machine learning models, or simple data transformation, the methods demonstrated--`substring()` and `substring_index()`--provide powerful, distributed solutions.

By mastering these techniques, data professionals can significantly enhance the quality and structure of their textual data, making it ready for advanced analysis within the **PySpark** ecosystem. For even more complex needs, the availability of `regexp_extract()` ensures that virtually any pattern-based extraction task can be accomplished.

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