How to Remove Leading Zeros from a PySpark Column

Authored by

stats writer

February 6, 2026

The necessity of performing rigorous data cleaning operations cannot be overstated in modern data engineering workflows. When dealing with datasets originating from diverse sources--such as legacy systems or improperly formatted exports--it is common to encounter columns that use leading zeros to maintain a fixed-width string format, even if the underlying data represents a numerical identifier. Using the powerful capabilities of PySpark, data professionals can efficiently manage and transform massive datasets within a distributed computing environment.

This detailed guide focuses on the precise methodology for removing these superfluous leading zeros from a string column within a PySpark DataFrame. While simpler methods might exist for single-character trimming, the most robust and widely accepted approach leverages the flexibility of regular expressions, ensuring accuracy and minimal performance impact across large clusters. This transformation is crucial for standardizing data formats, preparing identifiers for key joining, or enabling subsequent casting into integer types for mathematical operations.

PySpark: Remove Leading Zeros in Column

The Primary Solution: Utilizing PySpark's `regexp_replace` Function

To achieve highly controlled string manipulation, particularly when dealing with patterns that must only apply at the start of a string, PySpark offers the `regexp_replace` function, available within the `pyspark.sql.functions` module. This function allows us to define a precise regular expression that targets only the leading zero characters, replacing them with an empty string, thereby effectively removing them without affecting necessary internal zeros.

The general syntax for implementing this transformation involves importing the necessary functions, selecting the target column, and applying the regular expression pattern designed to match one or more zeros beginning at the start of the string. This method is superior to simple string trimming because it guarantees that only the leading characters--and not zeros embedded within the identifier--are removed, preserving the integrity of the remaining numerical value.

The following code snippet demonstrates the fundamental usage of `F.regexp_replace` to clean the target column named `employee_ID`:

from pyspark.sql import functions as F

#remove leading zeros from values in 'employee_ID' column
df_new = df.withColumn('employee_ID', F.regexp_replace('employee_ID', r'^*', ''))

In this powerful yet concise operation, we utilize `df.withColumn` to either create a new column or overwrite the existing `employee_ID` column with the cleaned values. The crucial component here is

the regular expression pattern `r'^*'`, which ensures that only the initial zero characters are targeted for replacement. All other zero characters that are not at the beginning of the field remain untouched, maintaining the accuracy of the employee identifier.

Deconstructing the Regular Expression Pattern (`r'^*'`)

Understanding the regular expression used in the transformation is essential for successful implementation and troubleshooting. A regular expression provides a standardized sequence of characters that define a search pattern. In our case, the pattern `r'^*'` is designed specifically to isolate the initial block of zeros in a string column.

Let's dissect the components of this pattern to clarify its function:

`^`: This is the anchor character, signifying the absolute **start** of the string. By including this anchor, we guarantee that the matching process only begins at the first character, preventing internal zeros from being affected.
: This simple character class matches the literal character `0`.
`*`: This quantifier means "zero or more occurrences" of the preceding element (which is). Combining `*` means we match any sequence of zeros. Since the sequence is anchored by `^`, we match all leading zeros.

When regexp_replace encounters this pattern, it successfully identifies the entire block of leading zeros (e.g., "000501" matches "000") and replaces the match with the specified empty string (`''`), resulting in the desired output ("501"). This precise targeting ensures that the data cleaning process is both effective and non-destructive to necessary data elements.

Practical Example: Setting Up the PySpark DataFrame

To illustrate this process in a tangible context, we will construct a sample PySpark DataFrame simulating records of employee sales. These records feature an `employee_ID` column where identifiers have been padded with leading zeros, necessitating standardization before further analytical use.

The setup involves initializing a Spark session, defining the data, specifying the column schema, and creating the initial DataFrame. This foundational step is crucial for demonstrating how the transformation operates on real-world data structures within the distributed environment managed by Spark.

The following example sets up the environment and the initial DataFrame:

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()

+-----------+-----+
|employee_ID|sales|
+-----------+-----+
000501	18
000034	33
009230	12
000451	15
000239	19
002295	24
011543	28
+-----------+-----+

Upon inspecting the initial output, it is immediately clear that every string value within the `employee_ID` column contains one or more leading zeros. For example, employee '501' is represented as '000501'. This format, while potentially useful for fixed-width parsing, obstructs direct use in numerical analysis or standard database indexing, thereby mandating the cleanup operation we are about to perform.

Implementing the Zero Removal Transformation

With the DataFrame initialized, we can now apply the previously defined methodology using the

regexp_replace function. The goal is to create a new DataFrame, `df_new`, where the `employee_ID` column is updated with the standardized identifiers. This procedure showcases the power of declarative data transformation in Spark, where the logic is applied across the distributed data partitions efficiently.

We reaffirm the import of the functions module and utilize the `withColumn` method, passing the target column name, the regexp_replace function, the column reference, the regular expression pattern, and the replacement string (which is empty). This sequence ensures that the transformation is atomic and scalable, applying the cleansing logic consistently across all records in the distributed cluster.

Below is the syntax implementation and the resulting DataFrame display:

from pyspark.sql import functions as F

#remove leading zeros from values in 'employee_ID' column
df_new = df.withColumn('employee_ID', F.regexp_replace('employee_ID', r'^*', ''))

#view updated DataFrame
df_new.show()

+-----------+-----+
|employee_ID|sales|
+-----------+-----+
501	18
34	33
9230	12
451	15
239	19
2295	24
11543	28
+-----------+-----+

The updated DataFrame, `df_new`, clearly demonstrates the successful removal of the leading zeros. For instance, '000501' has been converted to '501', and '000034' is now simply '34'. This transformed data is now clean, standardized, and ready for advanced operations such as type casting to integer or performing joins with other normalized datasets. The efficiency of PySpark ensures this process scales effectively even for extremely large volumes of data.

Alternative Approaches and Conclusion

While `regexp_replace` provides the most robust and controlled mechanism for removing leading zeros, it is worth noting an alternative function, `ltrim`, which is designed for simpler trimming tasks. The `ltrim` function removes specified leading characters from a string. If the requirement is strictly to remove all leading '0' characters, `F.ltrim(df, '0')` would also achieve the result.

However, the `ltrim` function operates by removing any combination of the specified characters from the left side of the string until a non-matching character is found. In contrast, regexp_replace using the `^` anchor and the `*` quantifier is conceptually clearer and guarantees that the replacement logic strictly adheres to the regular expression pattern. For complex data cleaning tasks, especially those involving numerical identifiers that might contain internal zeros, `regexp_replace` is the recommended best practice due to its high precision and explicit handling of patterns.

In summary, leveraging PySpark's SQL functions, specifically `regexp_replace` coupled with a precise regular expression, offers a highly efficient and scalable solution for data standardization. Mastering this technique ensures that your data pipelines produce clean, reliable identifiers, which is a foundational requirement for accurate reporting and complex analytical tasks within any big data environment.

The complete documentation for the PySpark **regexp_replace** function can be found at its official source, providing further details on advanced usage scenarios and performance considerations.

Further PySpark Transformation Tutorials

The following resources explain how to perform other common data manipulation tasks in PySpark:

Tutorial on converting string columns to numerical types.
Guide on filtering rows based on complex logical conditions.
Documentation for joining multiple DataFrames efficiently.