How to Compare Strings in Two PySpark Columns

Authored by

stats writer

February 4, 2026

One of the most frequent tasks in big data processing and data analysis involves ensuring data quality and consistency across various fields. When working with large-scale datasets, the powerful distributed computing framework offered by PySpark is essential. Specifically, comparing strings stored in different columns within a DataFrame is a fundamental requirement, whether you are performing merge operations, validating user inputs, or identifying duplicates. Understanding the various methods available in the PySpark SQL module allows developers and data scientists to execute these comparisons with high efficiency and precision.

To effectively compare string data between two columns in a PySpark environment, a clear set of steps must be followed, beginning with environment setup and concluding with result evaluation. Furthermore, PySpark offers flexibility through functions like `when()` and operators such as LIKE for more complex, conditional comparisons, ensuring robust analysis and manipulation of complex string data.

PySpark: Compare Strings Between Two Columns

Prerequisites and Initial Setup of the PySpark Environment

Before executing any comparison logic, the environment must be correctly configured. This involves importing essential libraries from the PySpark framework and initializing the Spark session. The Spark session serves as the entry point to communicate with the core Apache Spark functionality and is indispensable for DataFrame operations.

The standard steps for setting up and preparing the data are outlined below. While the specific data loading method may vary depending on the source (e.g., CSV, Parquet, Hive), the goal is always to have the target string columns available within a PySpark DataFrame

object, ready for analysis.

First, ensure the necessary libraries, particularly those concerning SQL functions and the Spark session, are imported into your Python environment.
Initialize a Spark session using the Builder pattern, allowing Spark to manage the underlying cluster resources required for distributed computation.
Load the dataset containing the two string columns that need comparison into a PySpark DataFrame.
Optionally, use the `select()` function if you wish to work only with a subset of columns, focusing exclusively on the two columns designated for comparison, thereby improving processing efficiency.

You can use the following concise syntax to compare strings between two columns in a PySpark DataFrame:

Method 1: Case-Sensitive String Comparison Using Equality

This method provides the quickest way to check for absolute equality, including character case, between two string columns. In PySpark, the direct use of the `==` operator on column objects automatically performs a distributed, row-wise comparison.

The resulting **Boolean** output must be integrated into the DataFrame structure, typically achieved using the **withColumn** transformation, which appends the comparison results as a new column.

```
df_new = df.withColumn('equal', df.team1==df.team2)
```

This specific example executes a strict comparison between the strings contained in columns team1 and team2. The resulting new column, typically named 'equal', returns either True or False, definitively indicating whether the strings are identical in content and case.

Method 2: Case-Insensitive String Comparison Using Transformation

For comparisons where the capitalization of characters should be ignored, it is necessary to apply a transformation function to both columns before executing the equality check. This normalization step is crucial for reconciling data entry inconsistencies, such as varying capitalization (e.g., 'Apple' vs. 'apple').

The built-in **lower** function (imported from `pyspark.sql.functions`) is the standard tool for

converting strings to a consistent lowercase format across the entire column before comparison, ensuring that the equality check operates only on the semantic content.

```
from pyspark.sql.functions import lower

df_new = df.withColumn('equal', lower(df.team1)==lower(df.team2))
```

This implementation performs a robust case-insensitive comparison by normalizing the strings in both the team1 and team2 columns to lowercase prior to the equality check. This approach is highly effective for reconciling data entry inconsistencies.

Preparing the Sample Data: Creating a Test DataFrame

To demonstrate these two powerful comparison methods in practice, we will utilize a small, representative DataFrame. This sample data set contains basketball team names across two columns, team1 and team2, deliberately including variations in capitalization to showcase the critical differences between case-sensitive and case-insensitive

comparisons.

The following code snippet shows the initialization of the Spark session, the definition of the data and column names, and the creation and display of the resulting DataFrame. Note the variations in rows 2 and 5, which serve as our primary test cases for case sensitivity.

```
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()

+-------+-------+
| team1| team2|
+-------+-------+
| Mavs| Mavs|
| Nets| nets|
| Lakers| Lakers|
| Kings| Jazz|
| Hawks| HAWKS|
|Wizards|Wizards|
+-------+-------+
```

This output confirms that our test data is correctly loaded and structured. The next steps will apply the string comparison methods detailed previously to this sample DataFrame, providing tangible results for interpretation of the two different equality approaches.

Example 1: Performing Case-Sensitive String Comparison

Using the strict equality method, we expect only those rows where both the content and the case of the strings are exactly identical to return `True`. Rows like 'Nets' vs 'nets' and 'Hawks' vs 'HAWKS' are anticipated to return

`False` because of the capitalization mismatch, despite having the same sequence of letters.

We apply the withColumn transformation to introduce the results of the strict comparison into a new column called `equal`. This is the default and simplest form of comparison in Spark SQL expressions.

```
#compare strings between team1 and team2 columns (case-sensitive)
df_new = df.withColumn('equal', df.team1==df.team2)

#view new DataFrame
df_new.show()

+-------+-------+-----+
| team1| team2|equal|
+-------+-------+-----+
| Mavs| Mavs| true|
| Nets| nets|false|
| Lakers| Lakers| true|
| Kings| Jazz|false|
| Hawks| HAWKS|false|
|Wizards|Wizards| true|
+-------+-------+-----+
```

As demonstrated by the output, the new column named equal accurately returns True only if the strings match perfectly, including the case of the characters between the two columns. The discrepancies in capitalization correctly yield False results, highlighting the strict nature of this method. This approach is highly useful when case preservation is a requirement, such as comparing hash values or database primary keys.

Example 2: Implementing Case-Insensitive String Comparison

When comparing strings where case differences should be ignored--a necessity in many real-world scenarios involving names, categories, or user inputs--the use of the lower function before comparison is essential. By converting both columns to a consistent case (typically lowercase) before the equality check, we standardize the comparison basis, minimizing false negatives caused by inconsistent data entry.

By applying the transformation, we ensure that the logical comparison focuses solely on the sequence of characters. In this specific example, we anticipate that the previously unsuccessful matches ('Nets' vs 'nets' and 'Hawks' vs 'HAWKS') will now correctly evaluate to

`True`.

```
from pyspark.sql.functions import lower

#compare strings between team1 and team2 columns (case-insensitive)
df_new = df.withColumn('equal', lower(df.team1)==lower(df.team2))

#view new DataFrame
df_new.show()

+-------+-------+-----+
| team1| team2|equal|
+-------+-------+-----+
| Mavs| Mavs| true|
| Nets| nets| true|
| Lakers| Lakers| true|
| Kings| Jazz|false|
| Hawks| HAWKS| true|
|Wizards|Wizards| true|
+-------+-------+-----+
```

The resulting column equal now returns True if the strings match regardless of their capitalization. This

transformation is a staple technique in data analysis pipelines for improving data quality and maximizing matches between slightly inconsistent fields.

Advanced Techniques: Conditional Comparison using WHEN and LIKE

While direct equality (`==`) is excellent for exact matches, string comparison often requires more flexible pattern matching or conditional logic, particularly when cleaning data or performing sophisticated validation. This is where the `when()` function, combined with SQL pattern matching operators like LIKE or `rlike` (for regular expressions), comes into play.

The `when()` function is critical when you need to assign custom values based on the comparison result, rather than just `True`/`False`. It evaluates conditions sequentially. If the goal is not just to determine absolute equality but to categorize the relationship between the strings (e.g., 'Exact Match', 'Partial Match', or 'No Match'), the function is indispensable.

The LIKE operator is used within PySpark to check if a string matches a specified pattern. This is particularly relevant when comparing strings that might share a

common prefix or suffix. LIKE uses standard SQL wildcards, such as % (matches any sequence of zero or more characters) and _ (matches any single character). Combining `when()` with LIKE allows for sophisticated, customized string validation rules that go beyond simple direct comparison.

Note #2: You can find the complete documentation for the PySpark withColumn function here.

Related PySpark Tutorials

The following tutorials explain how to perform other common tasks in PySpark, building upon the foundational knowledge of DataFrame manipulation demonstrated here: