How to Concatenate Columns in PySpark: A Step-by-Step Guide

Authored by

stats writer

January 1, 2026

PySpark can be used to Concatenate Columns of a DataFrame in multiple, highly optimized ways. This process is essential for data transformation, whether you are preparing fields for analysis or creating clean output data. The primary methods available leverage dedicated functions within the `pyspark.sql.functions` module, ensuring scalability across distributed clusters.

One fundamental approach uses the concat() function, which takes a list of column expressions (representing strings) as its arguments and returns a single string that is the raw concatenation of all the inputs. A second, often more useful method, is to use the concat_ws() function. This function requires a delimiter as its first argument and subsequently merges the strings, inserting the specified delimiter between each element. Examples demonstrating how to implement these highly efficient functions are provided below, using a typical data processing scenario.

Core Functions for String Concatenation in PySpark

When preparing data for analysis or reporting, combining multiple string fields into a single column is a routine requirement. In the distributed computing environment of PySpark, it is critical to use native SQL functions rather than Python UDFs to maintain performance. The `pyspark.sql.functions` library provides two specialized tools for this task, offering flexibility based on whether a separator is required.

These functions are designed to operate directly on column expressions across the cluster, guaranteeing that the operation is executed efficiently. We will explore how to apply these methods using the `withColumn` transformation, which is the standard way to add or update columns in a DataFrame.

The two core functions you can use to concatenate strings from multiple columns in PySpark are described below:

Method 1: Concatenate Columns Without a Separator using concat()

The concat() function performs a direct, raw merger of column contents. It accepts any number of column names as arguments and simply appends the string representation of each column next to the previous one. This is most useful when generating internal keys or identifiers where visual separation is not necessary.

from pyspark.sql.functions import concat

df_new = df.withColumn('team', concat(df.location, df.name))

This particular example uses the **concat** function to merge the strings found in the **location** and **name** columns into a new, singular column called **team**. Notice that the resulting strings will run

together without any intervening spaces or delimiters.

Method 2: Concatenate Columns with a Separator using concat_ws()

For creating human-readable strings, such as addresses, full names, or descriptive labels, the concat_ws() function (Concatenate With Separator) is superior. It requires the separator string as its first argument, followed by the columns to be joined. This approach drastically improves data clarity.

from pyspark.sql.functions import concat_ws

df_new = df.withColumn('team', concat_ws(' ', df.location, df.name))

This particular example uses the **concat_ws** function to merge the strings in the **location** and **name** columns into a new field called **team**, specifically utilizing a single space (' ') as the required separator between the elements.

Setting Up the Demonstration DataFrame

To effectively showcase the differences between `concat()` and `concat_ws()`, we must first establish a standard DataFrame. This dataset, representing basketball team information, will serve as the source for our column concatenation exercises.

The following code snippet initializes a Spark session and defines a sample dataset. We create a DataFrame with three columns: `location` (string), `name` (string), and `points` (integer). Note that while `points` is numerical, string concatenation functions will only operate successfully on the `location` and `name` columns, unless `points` is explicitly cast to a string type.

The following examples show how to use each method in practice with the following PySpark DataFrame:

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

# Define the raw data structure for NBA teams
data = ,
,
,
,
,

]

Define the column names for the structure
columns =

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

# View the initial DataFrame structure and contents
df.show()

```
+----------+-------+------+
| location| name|points|
+----------+-------+------+
| Dallas| Mavs| 18|
| Brooklyn| Nets| 33|
| LA| Lakers| 12|
| Boston|Celtics| 15|
| Houston|Rockets| 19|
|Washington|Wizards| 24|
| Orlando| Magic| 28|
+----------+-------+------+
```

This initial DataFrame, named `df`, is the foundation for our transformations. The goal is to combine the `location` and `name` columns into a new composite column called `team`, illustrating how the functions handle the merging process differently.

Example 1: Using concat() for Raw Concatenation

In our first practical example, we apply the `concat()` function. This function is ideal for scenarios where the resulting string must be compact and continuous, often for internal systems that do not rely on visual delimiters for parsing. We utilize the `withColumn` method to append the new `team` column to our existing DataFrame.

We use the following syntax to concatenate together the strings in the **location** and **name** columns into a new column called **team**:

from pyspark.sql.functions import concat

Concatenate strings in location and name columns without a separator

```
df_new = df.withColumn('team', concat(df.location, df.name))

# View the new DataFrame to observe the concatenated results
df_new.show()

+----------+-------+------+----------------+
| location| name|points| team|
+----------+-------+------+----------------+
| Dallas| Mavs| 18| DallasMavs|
| Brooklyn| Nets| 33| BrooklynNets|
| LA| Lakers| 12| LALakers|
| Boston|Celtics| 15| BostonCeltics|
| Houston|Rockets| 19| HoustonRockets|
|Washington|Wizards| 24|WashingtonWizards|
| Orlando| Magic| 28| OrlandoMagic|
+----------+-------+------+----------------+
```

As clearly shown by the output, the new **team** column successfully merges the contents, but without any space. For example, 'Dallas' and 'Mavs' are strictly combined to form 'DallasMavs'. This output style highlights the importance of choosing the correct concat() function based on the downstream application's needs.

It is crucial to understand the behavior of concat() regarding null values. If even one of the input columns in a given row holds a `null` value, the resulting concatenated string in the output column for that entire row will also be `null`. Data engineers must often incorporate pre-processing steps, such as imputing or replacing nulls with empty strings (`""`), to prevent cascading nullification when using this function.

Note: You can find the complete documentation for the PySpark **concat** function.

Example 2: Using concat_ws() for Delimited Concatenation

For most user-facing or analytical tasks, using a separator during concatenation is standard practice. The `concat_ws()` function provides this capability efficiently. In this demonstration, we specify a single space (' ') as the required separator to create easily readable team names.

We can use the following syntax to concatenate together the strings in the **location** and **name** columns into a new column called **team**, using a space as a separator:

from pyspark.sql.functions import concat_ws

```
# Concatenate strings in location and name columns, using space as separator
df_new = df.withColumn('team', concat_ws(' ', df.location, df.name))

# View the new DataFrame
df_new.show()

+----------+-------+------+------------------+
| location| name|points| team|
+----------+-------+------+------------------+
| Dallas| Mavs| 18| Dallas Mavs|
| Brooklyn| Nets| 33| Brooklyn Nets|
| LA| Lakers| 12| LA Lakers|
| Boston|Celtics| 15| Boston Celtics|
| Houston|Rockets| 19| Houston Rockets|
|Washington|Wizards| 24|Washington Wizards|
| Orlando| Magic| 28| Orlando Magic|
+----------+-------+------+------------------+
```

The resulting **team** column clearly benefits from the separator. 'Dallas' and 'Mavs' are now represented as 'Dallas Mavs'. This function greatly improves the clarity and usability of the resulting field, making it the preferred method for generating display values.

A significant technical advantage of the concat_ws() function is its intelligent handling of `null` inputs. When `concat_ws()` encounters a `null` value in one of the provided columns, it simply skips that element and does not insert a separator where the null occurred. This prevents the generation of unwanted extraneous delimiters (e.g., 'Part1--Part3') and avoids the common pitfall of a single null input causing the entire output string to become null, as is the case with `concat()`. This makes `concat_ws()` highly resilient to minor data quality issues.

Note: You can find the complete documentation for the PySpark **concat_ws** function.

Data Type Compatibility for Concatenation

A common pitfall when performing string operations in PySpark is attempting to mix string columns with non-string columns (e.g., integers, dates, or booleans) directly within concat() or concat_ws(). These functions are type-sensitive and are strictly optimized for string operations.

If your use case requires merging numerical data or timestamps with string fields, you must explicitly cast those columns to the `StringType()` before passing them to the concatenation function. This is achieved using the `.cast('string')` method on the column expression. For example, if we wished to include the numerical `points` column in our team string, the expression

would look like: `concat_ws(' - ', df.location, df.name, df.points.cast('string'))`. This practice is essential for preventing runtime exceptions and ensuring a smooth data pipeline.

Furthermore, for data engineers needing extremely complex, customized string transformations that might involve conditional logic or intricate formatting, using the SQL `expr()` function can offer additional power, allowing standard SQL string manipulation syntax to be applied directly within the DataFrame API. However, for 99% of concatenation requirements, the built-in `concat()` and `concat_ws()` functions provide the necessary functionality with maximum performance.

Summary of Best Practices

Mastering the distinction between concat() and concat_ws() is fundamental to efficient PySpark development. Choosing the right tool based on the required output format and null value tolerance will significantly impact the robustness of your data transformation workflow.

For Raw Merging: Use concat() when a continuous string is needed, but be sure to pre-process nulls.

For Readable Output: Use concat_ws() for human-readable output, as it includes a separator and inherently handles null values more gracefully by simply ignoring the missing elements.

Type Safety: Always remember to `cast()` non-string columns to `StringType()` before attempting any string concatenation.

By following these guidelines and utilizing the optimized SQL functions, developers can ensure that column concatenation in PySpark is both accurate and highly performant across large datasets.

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