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

Authored by

Mohammed looti

November 11, 2025

RECOMMENDED CITATION

Mohammed looti (2025). *Learning to Concatenate Columns in PySpark: A Step-by-Step Guide*. PSYCHOLOGICAL STATISTICS. Retrieved from https://statistics.arabpsychology.com/?p=16646

Introduction to Column Concatenation in PySpark

In modern big data processing pipelines, leveraging PySpark is essential for handling massive datasets efficiently. A common requirement in data preparation, normalization, and feature engineering is the combination of string data from multiple columns into a single, cohesive column. This process, known as concatenation, allows developers and data engineers to create human-readable identifiers or composite keys. PySpark provides several highly optimized functions within the `pyspark.sql.functions` module to achieve this operation seamlessly.

Specifically, when dealing with DataFrame structures, there are two primary methods available for concatenating strings from distinct columns. The choice between these methods usually depends on whether a separator is required to enhance readability or distinguish between the original column values. Understanding the nuances of each function is crucial for writing clean, performant Spark code. The two main functions we will explore are `concat` and `concat_ws` (concatenate with separator).

The following sections detail the syntax and practical implementation of both techniques, providing clear examples using the powerful withColumn transformation to append the newly generated composite field to the existing DataFrame.

Setting Up the Environment and Sample DataFrame

To demonstrate the concatenation techniques effectively, we first establish a sample DataFrame. This DataFrame simulates typical structured data, containing columns that hold location information, team names, and performance metrics. We will use the `location` and `name` columns as our input for the concatenation examples, aiming to create a new `team` column that combines these two fields.

The following initialization code sets up the Spark Session, defines the sample data structure, and creates the baseline DataFrame used throughout the subsequent examples. This is standard practice in PySpark development to ensure reproducible results.

```
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()

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

As shown in the output, our starting DataFrame `df` is ready for transformation. We will now apply the concatenation methods to combine the `location` and `name` fields.

Method 1: Direct String Concatenation using concat()

The most straightforward approach for joining strings from multiple columns is by utilizing the PySpark SQL function, concat. This function accepts a variable number of column expressions and returns a new string formed by appending the input strings end-to-end without introducing any delimiters or intervening characters. This method is ideal when the resulting field does not require human readability separation or when the strings themselves already contain the necessary structure.

To implement this, we must first import the function from `pyspark.sql.functions`. We then apply it using the withColumn method, which is the standard PySpark mechanism for adding or replacing columns in a DataFrame in a lazy and optimized manner.

from pyspark.sql.functions import concat

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

The code snippet above demonstrates the basic syntax: we instruct the DataFrame `df` to create a new column named `team` by applying the concat function to the `location` column followed by the `name` column. It is important to remember that concat offers no built-in separator, meaning the resulting strings will abut directly against one another, as shown in the execution example below.

Example 1: Demonstrating Direct Concatenation using concat()

We execute the syntax described above to combine the strings in the `location` and `name` columns into the new `team` column.

```
from pyspark.sql.functions import concat

#concatenate strings in location and name columns
df_new = df.withColumn('team', concat(df.location, df.name))

#view new DataFrame
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|
+----------+-------+------+-----------------+
```

Observe the output in the new `team` column. For instance, 'Dallas' and 'Mavs' are merged into 'DallasMavs'. While this technique is fast and simple, the lack of spacing often results in composite strings that are difficult to parse visually. This limitation leads us to consider the second, often more useful, concatenation method.

For detailed technical specifications, you can find the complete official documentation for the PySpark **concat** function on the Apache Spark website.

Method 2: Concatenation with Separator using concat_ws()

When readability is a priority, or when standard formatting is required (e.g., separating fields with a comma, hyphen, or space), the `concat_ws` (concatenate with separator) function is the preferred tool. Unlike `concat`, this function requires that the first argument specifies the delimiter string that should be placed between every subsequent concatenated column value.

The structure of `concat_ws` is highly advantageous for data cleaning tasks because it also includes implicit handling of null values. If a column passed to the function contains a null value, `concat_ws` gracefully ignores that null string, preventing the entire resulting concatenated string from becoming null (unless all input strings are null). This behavior significantly simplifies complex ETL logic that might otherwise require explicit null checks.

In this example, we will use a single space (' ') as the separator. This ensures that 'Dallas' and 'Mavs' will result in 'Dallas Mavs', vastly improving the legibility of the combined column. The implementation structure remains similar to Method 1, relying on the withColumn transformation.

```
from pyspark.sql.functions import concat_ws

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

Example 2: Executing Concatenation with Separator using concat_ws()

Executing the above syntax demonstrates the power of the separator argument. We create the new `team` column by combining the strings in the `location` and `name` columns, but this time, the output is separated by a space, resulting in clear, formatted team names.

```
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 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 results clearly show the improvement in output quality. The `team` column now contains properly spaced strings, making the data far more usable for reporting or external system integration. This highlights why `concat_ws` is often the default choice for general-purpose string combination in PySpark.

The official documentation provides comprehensive information on all parameters and usage scenarios for the PySpark **concat_ws** function.

Advanced Considerations and Summary

While `concat` and `concat_ws` handle the core task of combining columns, data professionals must also consider edge cases and performance characteristics within the Spark ecosystem. When working with large-scale data, the efficiency of these built-in functions is generally excellent, as they operate natively within the Spark engine (using Catalyst optimization).

A key difference to reiterate involves null handling:

concat: If any input column value is `NULL`, the resulting concatenated string for that row will also be `NULL`. This can lead to data loss if not explicitly managed.

concat_ws: This function ignores `NULL` values in the input columns. The resulting string is only `NULL` if the separator is `NULL`, or if all input string arguments are `NULL`. This robustness makes `concat_ws` safer for operational data pipelines where missing values are common.

Therefore, unless specific business logic demands that a row with a missing value results in a null composite string, `concat_ws` is highly recommended due to its superior handling of common data quality issues. If a specific format, such as `"column1:column2"` is needed, the separator argument can easily be set to a more complex string like `':'`.

Additional Resources for PySpark Mastery

Mastering column transformations is fundamental to effective big data processing using PySpark. Beyond concatenation, numerous other functions exist for string manipulation, conditional logic, and aggregation.

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

Transforming Columns: Learning how to use the withColumn method for data type casting and arithmetic operations.

Handling Nulls: Deep diving into functions like `coalesce` or `fillna` to manage missing data effectively before concatenation.

Conditional Logic: Implementing complex concatenation rules using `when` and `otherwise` clauses for nuanced data transformation.