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

Authored by

stats writer

February 7, 2026

Concatenation is a fundamental data manipulation technique involving the merging of two or more data fields or strings into a cohesive, single output field. In large-scale data processing environments like Apache Spark, specifically using PySpark, the ability to effectively combine columns is critical for feature engineering, creating descriptive identifiers, or preparing data for visualization. This guide delves into the primary functions available in PySpark SQL functions module for achieving robust column concatenation.

Understanding String Concatenation in Data Processing

The core method for combining arbitrary columns in a DataFrame is the `concat` function, imported from `pyspark.sql.functions`. This versatile function accepts a variable number of column inputs and merges their content sequentially. A crucial aspect of effective concatenation, especially when dealing with descriptive text fields such as names or addresses, is the inclusion of delimiters or separators to ensure readability and maintain data integrity.

For instance, consider the common scenario of merging "first_name" and "last_name". If concatenated directly, the result would be a single, unseparated string (e.g., "JohnDoe"). To introduce a necessary space, we must utilize the lit function, which stands for "literal." The lit function allows us to inject constant string values (like a space or a comma) directly into the concatenation sequence alongside the column references.

The following example illustrates how to create a new column, "full_name," by combining "first_name" and "last_name" separated by a literal space:

```
df.withColumn("full_name", concat(col("first_name"), lit(" "), col("last_name")))
```

Furthermore, the power of concat extends beyond two columns. When building a structured address or location field, you often need to combine multiple geographical components (city, state, country) and separate them using defined delimiters, such as commas and spaces. This requires inserting multiple literal strings between the column identifiers.

To create a comprehensive "location" field from "city," "state," and "country," separated by commas and spaces, the implementation becomes slightly more complex, demonstrating the ordered nature of the concat function:

```
df.withColumn("location", concat(col("city"), lit(", "), col("state"), lit(", "), col("country")))
```

This technique ensures the generated "location" column maintains a standard, readable format, such as "Austin, TX, USA." Understanding the synergy between `concat` and lit is foundational for precise string manipulation within PySpark DataFrames.

Concatenate Columns in PySpark (With Examples)

When working with string data in PySpark, there are two primary and highly effective methods for performing column concatenation. Choosing the right method depends largely on whether a specific separator is needed between the combined values.

Core PySpark Functions for Combining Columns

The two functions available within `pyspark.sql.functions` provide distinct advantages. The `concat` function offers maximum flexibility, allowing for the interleaving of literal strings and column data. The `concat_ws` (concatenate with separator) function, however, is designed specifically for efficiency when combining multiple columns using a single, consistent delimiter.

The first approach involves the direct use of `concat`. This method is straightforward for simple mergers or when custom separators are required, as demonstrated below for combining the `location` and `name` columns into a new field called `team` without any space.

Method 1: Direct Column Concatenation (No Separator)

```
from pyspark.sql.functions import concat

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

This particular example utilizes the `concat` function to merge the strings found in the `location` and `name` columns directly, resulting in a new column named `team` where the two strings abut each other.

The second, often preferred method for readable output, uses `concat_ws`. This function is optimized for inserting a single, specified separator between all the concatenated columns, significantly simplifying syntax compared to using `concat` with multiple `lit` calls.

Method 2: Concatenate Columns with a Defined Separator

```
from pyspark.sql.functions import concat_ws

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

This enhanced example employs the `concat_ws` function. It

concatenates the strings from the `location` and `name` columns into the new `team` column, but critically, it uses a specified string--in this case, a single space (`' '`)--as the separator between the inputs.

Setting Up the Environment: Creating the Source DataFrame

To demonstrate these two powerful PySpark concatenation methods practically, we must first establish a working DataFrame. This example uses data representing hypothetical sports teams, including their location, name, and points score. We initialize the Apache Spark session and define the schema before creating the structured data set.

The following block of code initializes the `SparkSession`, defines the input data (a list of lists), specifies the column names, and finally creates and displays the resulting source DataFrame (`df`) which we will use for both subsequent concatenation demonstrations.

```
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|
+----------+-------+------+

Example 1: Concatenate Columns in PySpark

Practical Application of the PySpark `concat` Function

This example demonstrates the raw concatenation of the `location` and `name` fields. Notice that because the `concat` function is used without any intermediary `lit` functions, the resulting `team` column merges the city and team names seamlessly, without a delimiter.

We use `df.withColumn()` to add the new `team` column to the existing DataFrame, utilizing the imported `concat` function.

```
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|
+----------+-------+------+----------------+
```

As observed in the output, the new `team` column successfully merges the values from the `location` and `name` columns, such as 'Dallas' and 'Mavs' becoming 'DallasMavs'. This output format is suitable for scenarios where a compact identifier is needed, but it sacrifices human readability.

Note: For detailed technical specifications and advanced usage patterns, consult the complete documentation for the **PySpark** `concat` function.

Example 2: Concatenate Columns with Separator in PySpark

Leveraging `concat_ws` for Separated Strings

When readability is paramount or the resulting string

needs to adhere to a specific format standard (e.g., separating components with hyphens, pipes, or spaces), the `concat_ws` function is the optimal tool. Unlike `concat`, where the separator must be manually inserted using `lit` for every gap, `concat_ws` takes the separator as its first argument and automatically applies it between all subsequent column arguments.

In this example, we apply `concat_ws` to combine the `location` and `name` columns, specifying a single space (`' '`) as the required delimiter. This dramatically improves the clarity of the output column `team`.

```
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 'Dallas Mavs' and 'Brooklyn Nets', demonstrating the effective use of the space separator defined in the first argument of `concat_ws`. This function is generally recommended for combining multiple columns where the separator remains uniform across all merged elements.

Note: The official documentation provides comprehensive details on using the **PySpark** `concat_ws` function, including how it handles null values (it skips them, but does not add the separator for the skipped field).

Best Practices and Performance Considerations

While both `concat` and `concat_ws` achieve column merging, choosing the correct function can impact code readability and processing performance in large-scale DataFrame operations.

Uniform Separators: Always favor `concat_ws` when merging three or more columns that require the same separator (e.g., combining 10 address fields with a pipe `|` delimiter). This is more concise and generally performs better than repeated use of `lit` with `concat`.

Custom Separators: Use `concat` in conjunction with `lit` only when different delimiters are needed within the same merged string (e.g., `col1, lit(' - '), col2, lit(' @ '), col3`).

Handling Nulls: An important distinction is how null values are handled. `concat_ws` is robust against nulls, skipping them entirely and not adding the separator, whereas `concat` will often propagate the null value to the entire resulting concatenated string if any input column contains a null for that row.

Conclusion and Further Resources

Mastering column concatenation is an essential skill for any data engineer working with PySpark. By leveraging

the specific functionalities of `concat` and `concat_ws`, developers can efficiently transform raw data fields into meaningful, structured identifiers tailored for complex analytics and downstream consumption.

For those looking to deepen their expertise in PySpark, consider exploring additional tutorials covering various common data manipulation tasks, such as handling date formats, conditional logic, and advanced filtering techniques.

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