How to Groupby and Concatenate Strings in PySpark: A Step-by-Step Guide

Authored by

stats writer

February 4, 2026

In the realm of large-scale data processing, efficiently manipulating and transforming textual information is paramount. PySpark, the Python API for Apache Spark, provides powerful tools for this purpose, most notably the combination of the `groupBy` and string concatenation functions. This powerful pairing allows data engineers and scientists to aggregate records based on shared attributes and then merge associated textual data into a single, cohesive field.

The `groupBy` function enables you to segment a DataFrame based on values in one or more columns, creating groups of related rows. Meanwhile, specialized concatenation functions facilitate the combining of strings from different rows that belong to the same group. By leveraging these features in tandem, you can significantly streamline complex tasks such as data cleansing, feature engineering, and report generation, where compiling lists of attributes per category is necessary. Mastering this technique is crucial for optimizing your PySpark data analysis pipeline and enhancing overall data manipulation capabilities.

PySpark: Use Groupby and Concatenate Strings

Understanding Grouping and Concatenation in PySpark

Data aggregation is a fundamental step in almost every data processing workflow. When working with large datasets, it is frequently necessary to summarize information or compile lists based on common identifiers. The `groupBy` operation in PySpark is designed precisely for this purpose. It transforms the structure of your data, preparing it for subsequent aggregation functions that operate on the resulting groups.

When the goal is to merge text fields associated with a group--such as combining all employee names linked to a single store ID--standard numerical aggregation functions like `sum` or `count` are insufficient. Instead, we must employ specialized aggregation functions, specifically `collect_list`, to gather all individual strings into a list structure, followed by `concat_ws`, which is designed to efficiently join array elements into a single string using a defined separator.

This combined approach ensures that the data retains its contextual integrity. For instance, if you are tracking product sales across different regions, you could use `groupBy` on the region column and then concatenate the list of unique product IDs sold there. This technique transforms high-granularity data into useful, summarized features, suitable for reporting or machine learning input, thereby drastically improving data utility and readability.

The Architecture of String Aggregation in PySpark

To successfully group rows and concatenate strings, PySpark relies on two critical functions found within the `pyspark.sql.functions` module: `collect_list` and `concat_ws`. Understanding their

roles is key to implementing the solution effectively. Since Spark processes data in a distributed manner, these functions are optimized to handle large volumes of data across multiple partitions efficiently.

The first component, `collect_list`, acts as the aggregating function during the grouping process. When applied to a column within a group, it collects all non-null values from that column into a standard array (list) type. If you were grouping employee data by store, `collect_list` would gather every employee name into a temporary list for each unique store ID.

The second component, `concat_ws` (meaning "concatenate with separator"), takes the array generated by `collect_list` and stitches its elements together into a single string. This function is indispensable because it explicitly requires a separator argument--such as a comma, space, or semicolon--which governs how the elements are joined, ensuring the final output is clean and structured. The combination of these functions within the `.agg()` method provides a concise and powerful syntax for complex string aggregations.

Implementing the Core PySpark Syntax

The standard syntax for grouping a DataFrame by one column and subsequently concatenating strings from another column involves using the `.groupby()` method followed by the `.agg()` method, which applies the necessary functions.

Here is the fundamental structure demonstrating how to group data based on the 'store' column and concatenate the list of 'employee' names using a comma and space (`, `) as the separator:

import pyspark.sql.functions as F

#group by store and concatenate list of employee names
df_new = df.groupby('store')
.agg(F.concat_ws(', ', F.collect_list(df.employee))
.alias('employee_names'))

This operation first instructs PySpark to partition the data based on identical values in the `store` column. Then, for each resultant group, it performs the aggregation: `F.collect_list(df.employee)` gathers all employee names, and `F.concat_ws(', ', ...)` joins them. Finally, `.alias('employee_names')` assigns a readable name to the newly created aggregate column.

It is important to note that the use of `collect_list` ensures that all elements are collected into an array structure before concatenation occurs. If you required uniqueness within the aggregated string, you would use `collect_set` instead of `collect_list`, although `collect_list` is the

appropriate choice when preserving all occurrences is desired, as is typical when listing names or identifiers.

Step-by-Step Example: Setting up the PySpark Environment and Initial Data

To illustrate this process clearly, we will begin by establishing a PySpark session and defining a sample DataFrame. This foundational step is crucial for running any PySpark script and ensuring that the structured data exists in memory for processing. Our dataset simulates employee records across different store locations and quarters, allowing us to perform meaningful aggregation.

The following code initializes the Spark session, defines the input data containing employee records, assigns appropriate column names (`store`, `quarter`, `employee`), and constructs the initial DataFrame named `df`. Viewing the DataFrame confirms the structure before manipulation.

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()

+-----+-------+--------+
|store|quarter|employee|
+-----+-------+--------+
A	1	Andy
A	1	Bob
A	2	Chad

B	2	Diane
B	1	Eric
B	4	Frida
C	2	Greg
C	3	Henry
+-----+-------+--------+

The objective is to consolidate the employee data such that for every unique entry in the `store` column, we generate a single row containing a list of all associated employee names. This transformation reduces the DataFrame's row count significantly while retaining all relevant textual information in an easily digestible format.

Applying Grouping and Concatenation to Employee Data

With our initial DataFrame ready, we can now execute the core operation. We specifically want to apply the `groupBy` method on the `store` column. This is the categorical key defining the groups we wish to analyze.

The aggregation is handled by chaining the `.agg()` function, where we utilize `collect_list(df.employee)` to gather names and immediately pass that array to `concat_ws`(', ', ...), specifying a comma followed by a space as the delimiter. This results in the creation of a new column, `employee_names`, which holds the aggregated string for each store.

import pyspark.sql.functions as F

#group by store and concatenate list of employee names
df_new = df.groupby('store')
.agg(F.concat_ws(', ', F.collect_list(df.employee))
.alias('employee_names'))

#view new DataFrame
df_new.show()

+-----+-----------------+
|store| employee_names|
+-----+-----------------+
A	Andy, Bob, Chad
B	Diane, Eric, Frida
C	Greg, Henry
+-----+-----------------+

This resulting DataFrame, `df_new`, is significantly simpler, condensing eight rows of initial data into three distinct records, each representing a unique store and containing a concatenated list of its employees. This structure is highly efficient for downstream reporting or analysis, as it minimizes joins required to link employees back to their respective stores.

Advanced Customization: Choosing Alternative Separators

One of the primary benefits of using the `concat_ws` function is the flexibility it offers regarding string delimiters. While a comma and space is standard for simple lists, sometimes business requirements demand a different separator, such as a hyphen, pipe symbol (`|`), or an ampersand (`&`) for a more formal connection.

If, for instance, the requirement was to present the employees joined by an ampersand, perhaps to indicate a team partnership, we simply adjust the first argument passed to `concat_ws`.

Here we update the syntax to concatenate the employee names using the `&` symbol as the separator:

import pyspark.sql.functions as F

#group by store and concatenate list of employee names
df_new = df.groupby('store')
.agg(F.concat_ws(' & ', F.collect_list(df.employee))
.alias('employee_names'))

#view new DataFrame
df_new.show()

+-----+-------------------+
|store| employee_names|
+-----+-------------------+
A	Andy & Bob & Chad
B	Diane & Eric & Frida
C	Greg & Henry
+-----+-------------------+

This customization demonstrates the ease with which presentation requirements can be met without altering the underlying grouping logic. Regardless of the separator used, the `groupBy` function ensures that the aggregation remains accurate, focusing only on records sharing the same grouping key.

Refining the Output using Aliases

A crucial step for maintaining code readability and producing professional output is assigning meaningful names to the resultant columns. When using the `.agg()` function, the resulting column name will default to a complex string based on the function used (e.g., `concat_ws(collect_list(employee))`), which is impractical for production environments.

To overcome this, we utilize the `.alias()` function immediately after defining the aggregation. As seen in the examples above, `.alias('employee_names')` explicitly sets the name of the new column to `employee_names`, ensuring clarity for future users of the DataFrame.

While the focus here was on simple string concatenation, the principles of `groupBy` and `agg` are broadly applicable. This methodology is often extended to calculate multiple metrics simultaneously within the same grouping context, such as calculating the count of records, the average numeric value, and concatenating strings, all in a single, efficient operation.

Further Exploration in PySpark Aggregation

The combination of grouping and string concatenation is just one facet of advanced data manipulation in PySpark. Data professionals often need to explore related aggregation techniques to handle various data types and scenarios.

For those interested in deepening their understanding of data aggregation and transformation, consider exploring tutorials on:

Using `collect_set` instead of `collect_list` when unique values are required in the aggregated array.
Applying Window Functions to perform rolling or relative aggregations without collapsing the original rows.
Implementing custom User Defined Functions (UDFs) for highly complex string operations that native PySpark functions cannot handle.

Mastering these techniques will significantly enhance your ability to preprocess complex datasets effectively for analytical or machine learning applications.

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