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

Authored by

stats writer

January 2, 2026

Introduction to Grouping and String Aggregation in PySpark

The ability to manipulate and reorganize large datasets efficiently is fundamental in modern data engineering. Within the realm of distributed computing, particularly when utilizing PySpark, two essential operations often work in tandem: grouping data based on specific criteria and performing aggregate functions on those groups. This technique is particularly powerful for transforming granular records into summarized, descriptive formats. By leveraging the built-in `groupBy` functionality combined with specialized string concatenation methods, data scientists can cluster related information and present it as a single, coherent string, simplifying subsequent analysis or reporting processes.

The core concept involves taking a massive dataset, like a list of sales records or employee entries, and partitioning it logically. For instance, you might want to see all employees associated with a particular store location, or all products sold on a specific date. The `groupBy` transformation handles this segmentation flawlessly, creating distinct groups based on the unique values found in the specified key columns. Once these groups are formed, standard SQL aggregation functions can be applied. However, aggregating strings requires a unique approach compared to standard numerical aggregations like sums or averages, which is where specialized functions come into play to merge individual string elements into a consolidated text field.

Mastering this combination of grouping and string merging is critical for preparing data for consumption by analytical tools or visualization platforms that often prefer consolidated fields over many individual records. The resulting concatenated string acts as a feature vector, summarizing complex relationships within the original data structure. Furthermore, this method is highly optimized within the distributed framework of Spark, ensuring that even petabyte-scale datasets can be processed quickly and reliably. Throughout this guide, we will explore the specific PySpark functions necessary to execute this powerful data transformation successfully, detailing the practical syntax and underlying logic required for clean, efficient code.

Understanding PySpark DataFrames and the groupBy Operation

At the heart of DataFrame operations in PySpark lies the concept of distributed data manipulation. A PySpark DataFrame is essentially a distributed collection of data organized into named columns, conceptually similar to a table in a relational database or a data frame in R/Pandas, but built to handle massive scale across a cluster of machines. The groupBy function is a foundational transformation used to initiate an aggregation stage. It instructs Spark to shuffle and partition the data such that rows sharing the same value in the specified column(s) are brought together onto the same partition, preparing them for a collective operation.

The syntax for using `groupBy` is remarkably straightforward, yet its distributed nature handles

immense complexity behind the scenes. When you call `df.groupBy('column_name')`, you are defining the clustering key. All subsequent aggregation functions will operate independently on these distinct groups. It is crucial to understand that calling `groupBy` itself does not immediately return the final result; instead, it returns a `GroupedData` object. This object is a prerequisite for executing an aggregation function, such as `sum()`, `count()`, or, in our specific case, a function designed for list collection and string merging.

When implementing string concatenation following a grouping operation, the choice of the aggregation function is paramount. Since strings cannot be summed or averaged in the traditional sense, we need a function that first collects all the relevant string values within a group into a manageable array or list. Once this intermediary list is created, we can apply a final function that iterates over the list elements and joins them together using a specified delimiter. This two-step approach--grouping followed by list creation and then concatenation--is the standard and most robust pattern for achieving grouped string aggregation within the PySpark environment.

The Mechanism of Aggregation: Combining Rows into Lists

After the groupBy operation has logically partitioned the DataFrame, the next step involves using an aggregation function that prepares the string data for concatenation. For this purpose, PySpark provides the highly useful function collect_list. This function takes all the values from a specific column within each defined group and gathers them into a single list structure. This list is then held within a new column in the aggregated result, effectively transforming many rows into one row per group, where the aggregated column contains an array of strings.

The `collect_list` function is preferred over `collect_set` when the preservation of duplicate values is important. If, for instance, an employee name appeared twice for a specific store due to data entry anomalies or if the underlying business logic required counting all instances, `collect_list` would maintain both occurrences. In contrast, `collect_set` would automatically remove duplicates, returning only unique values. For our goal of simply concatenating all corresponding strings, `collect_list` provides a reliable, ordered intermediary step, ensuring every element belonging to the group is captured before the final join operation.

Once the `collect_list` operation is performed, the aggregated data structure is ready for the final transformation. We move from a columnar structure where rows repeat the grouping key (e.g., 'Store A' repeated for every employee) to a summary structure where each row holds a unique grouping key (e.g., one row for 'Store A') and an array field containing all the relevant aggregated data (e.g.,). This array field, although useful on its own, is not yet in the desired single-string format. The next function in the chain, `concat_ws`, is specifically designed to take this array and format it into a displayable string, using a user-defined separator to bridge the elements.

Introducing concat_ws: Concatenating Strings with Separators

While `collect_list` provides the necessary collection of strings, the final step requires converting that list or array into a single, cohesive string output. This is achieved using the concat_ws function, which stands for "concatenate with separator." This function is purpose-built for aggregating arrays of strings into a singular scalar string value within a PySpark aggregation context. It requires two primary arguments: the delimiter (or separator) to be used between the concatenated elements, and the column containing the array of strings generated by `collect_list`.

The flexibility of concat_ws is evident in its handling of the separator. You can use any string as a delimiter--a comma, a space, a pipe symbol, an ampersand, or even a full phrase. Choosing the appropriate separator is crucial for ensuring the readability and usability of the resulting aggregated string. For reporting purposes, a simple comma and space (`', '`) is often standard, but for log parsing or specific data formats, more complex delimiters might be necessary.

The overall aggregation syntax structure often looks like this within the `.agg()` method: `F.concat_ws(separator, F.collect_list(df.target_column)).alias('new_column_name')`. The `.alias()` function is extremely important here. Since we are creating a completely new derived column based on an aggregation of multiple rows, we must provide a meaningful name for this column. Using `.alias()` ensures that the resulting DataFrame is clean and well-labeled, significantly improving code readability and making the aggregated data immediately understandable.

The standard pattern for grouping by a single column and concatenating the related strings utilizes the structure demonstrated below, integrating `concat_ws`, `collect_list`, and `alias`:

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 snippet efficiently groups all records based on the values in the **store** column and subsequently aggregates all corresponding strings from the **employee** column, joining them using a comma and space delimiter. The final output is stored in a new column named **employee_names**.

We will now walk through a complete, runnable example demonstrating this powerful syntax in action.

Practical Implementation: Setting up the PySpark Environment and Data

To effectively illustrate the combined power of groupBy and string aggregation, we must first establish a working PySpark environment and create a sample DataFrame. Our example centers around employee data, where each record specifies the store location, the operational quarter, and the employee's name. The ultimate objective is to transform this detailed record set into a summarized view where each row represents a unique store and lists all employees associated with it in a single string field.

Initializing the Spark session is the mandatory first step in any PySpark application, providing the entry point to Spark functionality. Following initialization, we define our raw data, which is structured as a list of lists, and the corresponding column schema. This clear separation of data and metadata ensures that the DataFrame creation is explicit and reproducible. The data setup ensures we have multiple entries sharing the same 'store' value (A, B, or C), providing the necessary structure for the grouping operation to yield meaningful results.

The following code block demonstrates the necessary steps to set up the environment, define the data structure, and display the initial, unaggregated DataFrame. This initial view allows us to clearly understand the input structure before we apply the transformations, confirming that the grouping keys (stores A, B, and C) have associated multiple employee entries that will need to be collected and concatenated. Note the use of `SparkSession.builder.getOrCreate()`, which either retrieves an existing session or initializes a new one, a standard practice for interactive PySpark scripting.

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 resulting DataFrame clearly shows the one-to-many relationship: multiple employees correspond to a single store. Our next step is to collapse these multiple employee rows into a single summary row per store, concatenating the names into a clean, comma-separated string. This transformation is highly efficient for data summarization tasks where detail must be preserved but structure simplified.

Applying the groupBy and concat_ws Functions (Comma Separator Example)

With the input DataFrame ready, we can now execute the core transformation. The goal is precise: group by the **store** identifier and then use the aggregation functions collect_list and concat_ws to produce the desired aggregated string. This process is initiated by calling `.groupby('store')`, which defines our aggregation boundaries, followed by the `.agg()` clause where we define the specific aggregation logic.

Inside the `.agg()` function, the nested function call is critical: `F.concat_ws(', ', F.collect_list(df.employee))`. The innermost function, `F.collect_list(df.employee)`, operates first, gathering all employee names for a given store (e.g., for Store A, it produces the list). The outer function, `F.concat_ws(', ', ...)`, then receives this list and joins its elements using the specified separator--a comma followed by a space (`', '`). Finally, the `.alias('employee_names')` clause assigns a descriptive name to this newly computed column, finalizing the transformation.

Executing this syntax transforms the eight-row input DataFrame into a three-row output DataFrame, perfectly summarizing the employee structure by store. This demonstrates how a complex, multi-row relationship can be condensed into a single descriptive field using powerful declarative functions in PySpark. Review the code execution and output below to see the results of this precise data aggregation:

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

The resulting **employee_names** column in the aggregated DataFrame now contains the concatenated strings, separated by a comma. This output is clean, concise, and ready for further analysis or integration into summary reports. For instance, we can verify the results against the original data:

Andy, Bob, and Chad all worked at store A according to the original data.
Diane, Eric, and Frida were associated with store B.
Greg and Henry were the two employees listed for store C.

This verification confirms the accuracy and effectiveness of the combined groupBy and collect_list/`concat_ws` operation.

Customizing Separators and Advanced Aggregation Techniques

One of the major strengths of the concat_ws function is its complete flexibility regarding the separator used. While commas are standard for lists, analytical requirements or aesthetic preferences might necessitate a different delimiter. By simply changing the first argument passed

to `concat_ws`, we can instantly customize the output format without altering the fundamental grouping or collection logic. This adaptability makes the function suitable for generating data outputs compliant with various file formats or specific report designs.

Consider a scenario where the output needs to be highly readable or perhaps integrated into a human-facing report where the standard comma list looks too technical. We might choose to use the ampersand symbol (`&`), potentially surrounded by spaces (`' & '`), to imply a partnership or relationship between the listed entities. This small change in the separator argument drastically alters the appearance of the resulting aggregated string, as demonstrated in the following example where we replace the comma separator with an ampersand.

Beyond simple string concatenation, this technique is easily extendable to more complex aggregation scenarios. For example, you could group by multiple columns (e.g., grouping by **store** AND **quarter**) to get an even finer granularity of aggregation. The syntax remains structurally identical; you simply pass more column names to the groupBy method. Furthermore, you can simultaneously apply multiple aggregation functions--such as concatenating employee names using `concat_ws(F.collect_list())` and calculating the count of employees using `F.count()`--all within the same `.agg()` clause. This capability for complex, simultaneous transformations is what makes PySpark such a powerful tool for large-scale data manipulation.

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
+-----+--------------------+

The resulting DataFrame, using the ampersand separator, showcases the flexibility of `concat_ws`. Remember that the collect_list function plays an indispensable role by collecting the raw strings into an intermediate array structure, making the final concatenation possible. Using the `.alias()`

function, as noted earlier, ensures the resulting column is appropriately named, preventing generic or confusing default names in the output DataFrame.

Summary and Next Steps in PySpark Aggregation

This exploration has detailed the powerful and efficient methodology for performing grouped string PySpark aggregation. By combining the distributional partitioning capabilities of groupBy with the array creation function collect_list and the flexible joining mechanism of `concat_ws`, we can transform granular, repetitive data into clean, summarized rows. This technique is invaluable for anyone working with large datasets where data summarization and preparation for reporting are key requirements.

The principles demonstrated here--defining the grouping key, collecting the aggregated elements into a list, specifying a joining separator, and assigning a clear alias--form a reusable pattern for various data manipulation tasks. Understanding how these functions interact within the distributed computing context of Spark ensures that your data processing pipelines are both effective in outcome and highly performant at scale.

To further expand your skills in PySpark data transformation, consider exploring other advanced aggregation techniques that might apply to different data types.

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