Learning PySpark: A Tutorial on Data Grouping and String Concatenation

Authored by

Mohammed looti

November 11, 2025

Introduction to Complex Data Aggregation in PySpark

In the world of big data processing, particularly when utilizing PySpark, data engineers frequently encounter the need to summarize vast amounts of information based on shared attributes. This process, known as data aggregation, involves consolidating rows within a DataFrame to generate meaningful, high-level summaries. A particularly powerful and common technique is the combination of grouping data by one categorical column and then merging the string values from a related column into a single, cohesive text field.

This specialized form of aggregation is indispensable for creating streamlined reports, generating unique composite keys, or compiling complete lists of associated entities. For instance, you might need to list all employee names tied to a specific store location or consolidate product identifiers linked to a single order ID. Achieving this complex transformation efficiently requires leveraging specific, highly optimized functions available within the pyspark.sql.functions module, allowing for clean execution in a distributed environment.

Understanding how to group transactional or relational data and transform associated arrays of strings into a single, delimited output is fundamental for effective data preparation in PySpark workflows.

The Foundational Functions for String Aggregation Syntax

To effectively group rows in a DataFrame and subsequently concatenate strings, we must utilize a specific combination of three essential functions: groupBy(), `collect_list`, and concat_ws(). The typical workflow involves calling the groupBy() method on the column intended for categorization, followed immediately by the `agg` (aggregate) function, which houses the logic for string merging.

The structure below illustrates the standard, idiomatic approach to this aggregation task. In this specific example, the data is grouped by the **store** column, and all corresponding string entries found in the **employee** column are combined into one field, separated by a comma and a space:

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'))
```

The process executes in a distinct sequence: first, `collect_list` operates within each group (defined by the store) to gather all relevant strings (employee names) into an array. Second, concat_ws() (Concatenate With Separator) takes this resulting array and efficiently merges all its

elements into a single string using the specified delimiter (`', '`). Finally, the `alias` function ensures the resulting aggregated column is clearly labeled as **employee_names**.

Setting Up the PySpark Environment and Sample Data

To provide a practical demonstration of this powerful aggregation technique, we must first establish a functioning PySpark environment and generate a representative sample DataFrame. This sample dataset is structured to mimic real-world employee records, including the store location, the quarter they worked, and their individual name. We initiate this process by importing necessary modules, starting the `SparkSession`, and converting our raw data list into a structured, relational format.

Our initial dataset intentionally features multiple entries for the same store (A, B, or C), creating the perfect scenario for aggregation. The core objective of this demonstration is to transform this detailed, row-level structure into a streamlined format where each store appears uniquely, accompanied by a comprehensive list of all associated employees. This transformation is a cornerstone of data preparation, particularly in distributed computing environments where data consolidation is essential before reporting.

The following code block defines the raw data, creates the DataFrame named `df`, and displays the initial, detailed structure, setting the stage for the subsequent aggregation step:

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

Executing the Grouping and String Concatenation Logic

With the sample data prepared, we now proceed to execute the core aggregation operation. Our objective remains consistent: group the data by the **store** column and concatenate the corresponding values found in the **employee** column. This requires importing the necessary functions from the pyspark.sql.functions module and then applying the chained aggregation methods to the initial DataFrame, `df`.

The outcome of this operation is the creation of a new DataFrame, designated as `df_new`, which presents a clean, high-level summary of the employee roster organized by store location. By inspecting the results of `df_new.show()`, we can visually confirm that the row-level details have been successfully condensed into concise, store-level summaries, demonstrating the efficiency and correctness of the groupBy() technique.

Below is the execution of the aggregation syntax and the resulting summarized output:

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 final **employee_names** column contains all names associated with a given store, delimited by `', '`. This new column clearly and efficiently summarizes the employee structure across the different locations:

For store A, the team consists of Andy, Bob, and Chad.
Store B is staffed by Diane, Eric, and Frida.
The team at store C includes Greg and Henry.

Advanced Customization: Modifying the Concatenation Separator

One of the significant advantages of using the concat_ws() function is its inherent flexibility in defining the separator. While the comma-and-space sequence (`', '`) is the conventional choice for combining list elements, real-world business requirements often demand different delimiters, such as a pipe (`|`), a dash (`-`), or a specific semantic symbol.

If, for example, the reporting standard required using the ampersand symbol (`&`) to visually represent team relationships instead of a comma, we only need to modify the first argument supplied to the concat_ws() function. This simple customization capability ensures that the resulting data structure perfectly aligns with specific visualization, integration, or downstream data consumption needs without requiring complex post-processing steps.

The following example illustrates how to redefine the separator string to use `' & '` for concatenating the employee names, resulting in an immediate change to the visual output while preserving the underlying grouping and aggregation logic:

import pyspark.sql.functions as F

```
# Group by store and concatenate list of employee names using '&'
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|
+-----+-------------------+
```

Deep Dive into PySpark Aggregation Primitives

To truly master this efficient aggregation pattern, it is vital to fully appreciate the specific responsibilities of the non-standard aggregation primitives employed in the `agg` step: `collect_list`, concat_ws(), and `alias`. These functions move beyond simple mathematical operations to handle complex data type transformations necessary for string manipulation.

`collect_list`: This function is arguably the most critical component. Unlike conventional aggregation functions (such as `sum` or `count`) which return a single scalar value, string concatenation requires an input collection. `collect_list` systematically transforms all values from the target column (e.g., **employee**) belonging to a specific group key (e.g., **store**) into a temporary array or list of strings. Importantly, `collect_list` preserves duplicate values. Its sibling function, `collect_set`, performs the same collection but automatically removes any duplicates encountered within the group.

`concat_ws()`: Short for "concatenate with separator," this function is specifically engineered to consume an array of strings (the direct output from `collect_list`) and merge them into a single, cohesive string output. Its syntax mandates that the separator string be specified as the very first argument, followed by the array column itself. Leveraging this built-in function is significantly more performant and reliable than attempting to implement manual string manipulations using custom User Defined Functions (UDFs) in a distributed environment.

`alias`: While the aggregation operation would technically run without it, using `alias` is essential for producing professional, readable, and maintainable code. It grants the developer the ability to assign a precise and descriptive name to the new column generated by the aggregation process (in our case, **employee_names**), ensuring that the resulting DataFrame structure is self-documenting and easy for subsequent users or processes to interpret.

Conclusion: Mastering Distributed String Aggregation

The combined use of groupBy(), `collect_list`, and concat_ws() constitutes the most robust and idiomatic pattern for handling sophisticated string aggregation tasks within PySpark. This technique is highly optimized to run efficiently on large datasets across a cluster, offering superior performance compared to traditional iterative or row-based processing methods. Achieving fluency in this specific pattern is a critical skill for any data professional involved in large-scale data preparation, feature engineering, and reporting pipelines.

To further enhance your understanding of PySpark and other essential data manipulation capabilities, we recommend exploring additional tutorials that detail how to accomplish other common distributed computing tasks.