How to Get the Last Row of a PySpark DataFrame

Authored by

stats writer

February 5, 2026

Introduction: Retrieving the Last Record from a PySpark DataFrame

Introduction: Challenges of Retrieving the Last Row in PySpark

Working with large datasets often requires precise data manipulation, and a common task in data engineering is isolating the final record within a distributed dataset. In PySpark, retrieving the last row from a DataFrame presents a unique challenge because DataFrames are inherently distributed and lack a guaranteed natural order unless explicitly defined.

One common, though potentially less efficient, approach involves using the **`orderBy()`** function to sort the entire DataFrame in descending order based on a specific index or timestamp column. Once sorted, applying the **`head()`** function (or `limit(1)`) retrieves the first entry, which corresponds to the last row in the sorted sequence. While functional, this method incurs the significant overhead of a full sort operation across all partitions.

Fortunately, PySpark offers a more robust and idiomatic solution for reliably identifying the last row, particularly when the dataset structure doesn't include a pre-existing sequential index. This superior method leverages built-in functions to assign a unique, increasing ID, allowing us to accurately pinpoint the final record without unnecessary resource expenditure. We will focus on implementing this optimized technique.

The Optimized Technique: Leveraging Monotonically Increasing IDs

To accurately determine the final row in a distributed environment, we must first impose an artificial order. The recommended technique involves utilizing the **`monotonically_increasing_id`** function. This function generates guaranteed unique and non-decreasing 64-bit integer IDs for each row across all partitions. Although these IDs are increasing, they are not necessarily consecutive, as the function calculates IDs based on the partition index and the record index within that partition. Crucially, however, the last row processed by the system will inherently possess the largest ID.

Once we have assigned these unique IDs, the retrieval task simplifies into a maximum selection problem. By grouping the row data with its corresponding ID using the **`struct`** function, we can then use the **`max`** aggregate function. Since the comparison of structures in PySpark is performed element-wise, finding the maximum of the structured data automatically selects the row associated with the highest ID value.

This approach is significantly more efficient than a global sort because **`monotonically_increasing_id()`** avoids full shuffling, and the subsequent use of **`max()`** is a well-optimized aggregation operation. The entire process consists of adding the ID column, finding the maximum structured row, and then cleanly dropping the temporary ID column to return the

original row structure.

PySpark Syntax for Last Row Retrieval

The following syntax encapsulates the steps described above, providing a clean and efficient way to extract the last row from any DataFrame. This method requires importing necessary functions from `pyspark.sql.functions`, which are essential for defining the temporary ID and performing the aggregation.

from pyspark.sql.functions import*#get last row of DataFrame
last_row = df.withColumn('id', monotonically_increasing_id())
.select(max(struct('id', *df.columns))
.alias('x')).select(col('x.*')).drop('id'))

Understanding the flow of this chained operation is key to effective PySpark manipulation. We start with the original DataFrame (`df`), enrich it with the unique ID, perform the core aggregation using **`max()`** on the structured data, and conclude by cleaning up the resulting schema to match the input schema. This structure ensures readability and maintainability, which are crucial in production environments.

Practical Example: Defining the Sample DataFrame

To illustrate this functionality in practice, let us define a sample DataFrame containing simulated data about basketball players. This example demonstrates how the solution works regardless of the data types or the complexity of the schema. We begin by initializing the Spark Session, defining our data structure, and viewing the resulting DataFrame structure.

The sample data includes columns for `team`, `conference`, `points`, and `assists`. For this demonstration, the "last row" is defined purely by the order in which the data was input or processed by the system before we apply our retrieval logic--in this case, the row representing Team C.

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()

```
+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
| A| East| 11| 4|
| A| East| 8| 9|
| A| East| 10| 3|
| B| West| 6| 12|
| B| West| 6| 4|
| C| East| 5| 2|
+----+----------+------+-------+
```

Executing the Retrieval Process

Our objective is clearly defined: retrieve the row corresponding to Team C, which is the final record in the defined dataset. Utilizing the previously introduced ID-based syntax, we can execute the retrieval logic. This step demonstrates the practical application of the functions **`monotonically_increasing_id`**, **`struct`**, and **`max`** working together to isolate the target row.

The code block below applies the chain of transformations to the sample DataFrame `df`. The result, stored in `last_row`, is a new DataFrame containing only the single, desired record.

from pyspark.sql.functions import*#get last row of DataFrame
last_row = df.withColumn('id', monotonically_increasing_id())
.select(max(struct('id', *df.columns))
.alias('x')).select(col('x.*')).drop('id'))

#view last row
last_row.show()

```
+----+----------+------+-------+
```

```
|team|conference|points|assists|
+----+----------+------+-------+
| C| East| 5| 2|
+----+----------+------+-------+
```

As demonstrated by the output, we have successfully isolated and extracted the final row corresponding to team C. This confirms that the complex chain of operations correctly identifies the record with the maximum synthetic ID, translating directly to the last row of the input data sequence.

Deconstructing the Retrieval Logic

Understanding the exact mechanics of how this syntax achieves the result is vital for any PySpark developer. The solution relies on three primary logical steps, which collectively bypass the need for a costly global sort operation. This technique is highly reliable for unstructured DataFrames where a natural ordering column might not exist or be trustworthy.

Here is a breakdown of the function chain:

Step 1: Assigning the ID Column: We initiated the process by invoking the **`monotonically_increasing_id`** function. This added a new column, temporarily named `id`, which contains unique, non-decreasing values. This ID serves as our temporary index, ensuring that the last row inserted or processed has the highest corresponding value.

Step 2: Aggregation using MAX and STRUCT: Next, we used the **`max`** function in conjunction with **`struct`**. The `struct` function combines the `id` column with all other original columns. When `max` is applied to this structured column, PySpark performs the comparison based on the columns in order. Since `id` is the first field in the structure, the row with the maximum ID is selected, effectively isolating the last row of the original DataFrame.

Step 3: Cleanup: Lastly, the temporary column alias `x` (representing the structure) is expanded back into individual columns using `select(col('x.*'))`, and the temporary `id` column is removed using `drop('id')`. This ensures the output DataFrame schema perfectly matches the input schema, yielding a clean result.

Conclusion and Resources

The methodology of combining **`monotonically_increasing_id`** with the **`max(struct(...))`** pattern provides a powerful and reliable mechanism for retrieving the last record in a PySpark environment. This technique is preferred over global sorting methods when dealing with massive,

distributed datasets where performance is a critical factor.

For developers seeking deeper technical reference, the complete documentation for the **monotonically_increasing_id** function provides further insight into its characteristics and usage constraints within the Spark ecosystem.

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