PySpark Tutorial: How to Get the Last Row of a DataFrame

Authored by

Mohammed looti

November 11, 2025

Welcome to this comprehensive guide on manipulating data efficiently within the PySpark DataFrame environment. Working with large-scale data using Apache Spark, a powerful engine designed for distributed data processing, introduces complexities that are absent in single-node tools like pandas or traditional SQL databases. One of the most common yet counter-intuitive challenges involves isolating the final record from a dataset. In conventional, ordered systems, retrieving the last row is trivial, often relying on an implicit index. However, a PySpark DataFrame is inherently distributed across a cluster and, critically, lacks guaranteed global ordering unless that order is explicitly enforced through costly operations. This distribution means that attempting to collect the data directly results in a non-deterministic sequence. Nevertheless, in many critical operational scenarios--such as analyzing records based on ingestion time, monitoring streaming data pipelines, or performing audit checks--accurately identifying and isolating the record that was logically processed last is essential.

To successfully pinpoint and extract this final row, we must first introduce a stable, sequential ordering mechanism that spans all partitions of the PySpark DataFrame. This mandatory ordering is achieved not by a full, resource-intensive global sort, but by leveraging specialized, built-in functions designed to generate unique, increasing identifiers across the entire dataset. Once this reliable, globally consistent index is established, the task simplifies dramatically: the " last row" is simply defined as the record associated with the maximum generated index value. This technique offers a robust and consistent solution, ensuring accuracy regardless of how the underlying data is partitioned or shuffled across the computing cluster.

This article will introduce an efficient and reliable methodology to extract the last record from any given PySpark DataFrame. We utilize a sophisticated combination of uniqueness assignment and aggregation techniques available through the `pyspark.sql.functions` module. The method showcased below is particularly optimized for performance because it skillfully avoids expensive global sort operations, relying instead on the swift generation of a unique ID followed by an efficient maximal selection procedure. This approach ensures minimal data shuffling and maximum computational speed. You can utilize the following concise syntax structure to efficiently retrieve the last row from a target PySpark DataFrame:

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'))

The subsequent sections will meticulously dissect this powerful, single-line code structure, explaining the precise purpose of each function call, detailing the underlying distributed logic, and

demonstrating its practical application using a concrete, real-world example to fully illustrate its effectiveness and performance benefits.

The Fundamental Challenge of Ordering in PySpark

To master complex data operations in Apache Spark, it is absolutely fundamental to grasp its distributed architecture. Conceptually, a PySpark DataFrame behaves like a table, but its data is physically partitioned and stored across multiple worker nodes within a cluster. When data is ingested or transformed, Spark's execution engine optimizes for parallelism by processing these partitions independently. This design choice, while boosting performance dramatically, means there is no inherent guarantee that the rows maintain any specific ingestion or logical order across the cluster. If an operation like `df.collect()` is invoked without a preceding global sort, the resulting list's order is highly non-deterministic and is likely to fluctuate across different execution runs due to varying resource availability and processing speeds on the worker nodes.

This non-deterministic behavior is a key trade-off for high performance; it eliminates the overhead of constant synchronization and coordination across the cluster nodes that would be necessary to maintain a strict global order. For many analytical tasks, this arbitrary ordering is acceptable. However, for use cases requiring precise sequential identification--such as finding the single latest record, determining the earliest entry, or calculating accurate running totals based on time--this lack of inherent, stable ordering presents a significant obstacle. Traditional solutions relying on implicit row numbers or simple indexing that work perfectly in single-machine environments are rendered insufficient because Spark's execution engine prioritizes parallel computation, meaning records may be outputted in arbitrary order across partitions.

To overcome this critical limitation, we must artificially introduce a unique and sequential index that is guaranteed to span all partitions consistently. This index then serves as the definitive source of truth for defining the desired sequential order, allowing us to accurately identify the maximal element, which corresponds precisely to the "last row" relative to the sequence we established. The methodology we choose must prioritize computational efficiency. Naive techniques, such as applying a Window function combined with `row_number()`, often necessitate an expensive global `orderBy` operation. This global sort forces a massive data shuffle across the entire cluster, severely impacting performance for datasets of any substantial size. Our preferred strategy, detailed below, utilizes a specialized function that generates identifiers without needing an immediate full global sort, offering a superior performance profile when the sole objective is to find the single record at the conceptual end of the sequence.

The Technical Solution: Leveraging Monotonically Increasing IDs

The cornerstone of our optimized solution is the monotonically_increasing_id function, which is

conveniently available within the `pyspark.sql.functions` module. This function generates unique 64-bit integer IDs for every row in the DataFrame. The critical guarantees provided by this function are twofold: first, the IDs are unique across the entire DataFrame; and second, they are continuously non-decreasing as rows are processed within any given partition. Furthermore, the IDs generated by partitions processed later are guaranteed to be strictly larger than those generated by earlier partitions. These guarantees ensure that, across the entire distributed DataFrame, the IDs increase globally, allowing us to confidently use them as a stable proxy for the row index relative to the data ingestion order.

It is important to understand the nuance of monotonically_increasing_id: while the IDs are monotonically increasing, they are not necessarily contiguous, meaning gaps may exist, particularly between partition boundaries. However, this non-contiguity is irrelevant to our goal. Since IDs generated by subsequent partitions are always larger, the row corresponding to the logically "last" operation will always possess the absolute maximum ID value within the resulting column. This characteristic makes the function the perfect, highly efficient tool for identifying the final record without incurring the massive performance penalty of forcing a full global sort, which is a key benefit when dealing with petabyte-scale data processing.

Once we have successfully assigned this unique identifier column, typically named 'id', the next step involves a strategic aggregation operation. A common mistake is to simply find the maximum 'id' and then filter the DataFrame based on that value, which necessitates two separate passes over the data and introduces potential data synchronization issues. To avoid this inefficiency, we utilize the struct function. This function cleverly combines the 'id' column along with all other data columns into a single, complex column. By applying the max function to this resulting struct column, PySpark intelligently uses the leading element (the 'id') for comparison and then returns the entire row associated with that maximal 'id' value. This achieves the desired result--retrieving the full last row--in a single, highly optimized aggregation step.

Practical Implementation Example: Setting up the DataFrame

To demonstrate this sophisticated technique in a concrete manner, let us first establish a sample PySpark DataFrame. Our example will focus on a small dataset representing basketball player statistics, including details such as team affiliation, conference, points scored, and assists recorded. Before any data manipulation can occur, the environment must be initialized by creating a SparkSession, which acts as the foundational entry point to all Spark functionality, enabling us to define and interact with our distributed data structures.

We begin by defining the raw data as a standard Python list of lists, where each inner list precisely represents one record. The corresponding column names are also explicitly defined to ensure clarity. Using the `spark.createDataFrame()` method, we transform this local Python structure into

a distributed PySpark DataFrame named `df`. This setup is crucial for simulating a real-world scenario where data is ingested sequentially, and the order of the records in the source defines which row is conceptually considered the "last" record for our subsequent analysis when we impose the unique ID.

The following Python code initializes the distributed environment, defines the sample data structure, creates the distributed DataFrame, and immediately displays the resulting dataset using the `df.show()` command. It is essential to note that the record associated with Team 'C' (5 points, 2 assists) is the final entry in our manually defined source dataset, and this is precisely the record we intend to isolate using the optimized PySpark syntax in the subsequent steps. This initial view confirms the target output before we apply the complex transformation logic.

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 Optimized Retrieval Operation

With our sample data prepared and loaded into the distributed DataFrame `df`, we can now apply the optimized PySpark syntax specifically designed for robust last row retrieval. This entire operation is executed as a single, carefully chained sequence encompassing transformations and a final action. The high efficiency of this technique stems directly from the minimal data shuffling required; the process focuses only on creating the unique ID and then performing a quick aggregation based on that ID. This methodology is demonstrably faster than methods that require sorting the entire DataFrame globally and then selecting the top or bottom record, which would force a highly resource-intensive reorganization of data across the entire cluster.

We start by ensuring that all necessary functions--specifically monotonically_increasing_id, the max function, and the struct function--are correctly imported from the `pyspark.sql.functions` module. The crucial insight during this execution is the use of the Python splat operator (`*df.columns`) within the struct function call. This dynamic inclusion ensures that every original column in the DataFrame, alongside the newly created 'id' column, is bundled together within the aggregation structure. This combined structure enables simultaneous comparison and retrieval of the full record associated with the maximum ID.

Executing the following sequence on our sample DataFrame `df` successfully isolates the final record, which corresponds to the entry for Team C. The resulting DataFrame, named `last_row`, contains only this single record. This outcome confirms that we have accurately extracted the desired data point without resorting to manual or implicit indexing, and without incurring the significant performance cost of a full global data sort across the cluster. The precise single-record output demonstrates the accuracy and efficiency inherent in the combination of unique ID generation and maximal selection techniques.

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 clearly demonstrated, we have successfully extracted only the last row from the PySpark DataFrame using a methodology that is both robust and inherently cluster-friendly, ensuring high performance even with massive datasets.

Deconstructing the Workflow: A Step-by-Step Analysis

To fully appreciate the efficiency and technical elegance of this solution, it is highly beneficial to break down the complex, chained operation into its discrete, logical steps. This methodology relies entirely on leveraging built-in features of Apache Spark that are designed to handle distributed computing optimally, guaranteeing that we circumvent the common pitfalls associated with naive index manipulation in a distributed environment. The core procedural goal is twofold: first, to assign a unique, globally increasing identifier; and second, to utilize that identifier as the definitive basis for a maximal selection operation.

The operation begins with the addition of the temporary 'id' column using the command `.withColumn('id', monotonically_increasing_id())`. This initial step is absolutely crucial because it provides the necessary, stable ordering context that the distributed PySpark DataFrame inherently lacks. Following this, the struct function performs the essential bundling operation, combining the newly created 'id' column and all original data fields into a single, complex column. This bundling allows the subsequent aggregation step to treat the entire row data as one logical unit that is inextricably tied to the 'id' for comparison purposes, accomplished via `max(struct('id', *df.columns))`.

Finally, the application of the max function to the structured column ensures that only the record containing the highest 'id' value--which is guaranteed to be the last record in the sequence--is retained. The subsequent selection operations (`.select(col('x.*'))`) and the final `.drop('id')` command serve purely for data cleansing and presentation. These steps flatten the structured column back into its original individual columns and remove the temporary index, thereby returning the DataFrame to its original schema structure, containing only the desired last row. Here is a concise summary of this functional flow:

First, we use the **monotonically_increasing_id** function to add a new column called **id** that contains monotonically increasing values. This action establishes a stable and reliable global ordering proxy.

Next, we employed the **max function**, applying it to a structured column created by the **struct**

function. This operation selects the entire row associated with the largest **id** value, which is guaranteed to be the last row in the logical sequence.

Lastly, we meticulously dropped the temporary **id** column from the DataFrame using `.drop('id')`, effectively restoring the original schema while preserving the extracted last record.

The end result is that we were able to isolate only the last row from the distributed DataFrame with maximum efficiency and reliability.

Note: Comprehensive documentation for the **monotonically_increasing_id** function, including detailed explanations of its internal workings and guarantees, can be found by referring directly to the official Apache Spark documentation.

Additional Resources for PySpark Mastery

Mastering complex PySpark DataFrame manipulation requires a deep familiarity with a wide array of functions and the distinct paradigms of distributed programming. Understanding precisely how to manage non-deterministic ordering, execute complex aggregations, and manipulate schemas efficiently is key to becoming truly proficient in large-scale data processing. We strongly encourage further exploration of related technical topics, particularly those concerning advanced window functions, effective data partitioning strategies, and optimized aggregation techniques. All these topics build upon the fundamental concepts introduced in this article regarding the establishment of stable indexing within challenging distributed data environments.

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