How to Retrieve the Last Row of a PySpark DataFrame Easily

Authored by

stats writer

January 2, 2026

Working with large datasets often requires precise data retrieval, and isolating specific records, such as the final entry, is a common requirement. When dealing with a PySpark DataFrame, obtaining the last row is not as straightforward as with local data structures like Pandas DataFrames or standard Python lists, primarily due to the distributed nature of Apache Spark. While a quick, but potentially inefficient, approach involves using the take() method combined with the DataFrame count (`df.take(df.count())`), this method forces the computation of the entire dataset size and risks collecting substantial data into the driver program, which can lead to performance bottlenecks and OutOfMemory errors on truly massive datasets.

A more robust and scalable solution involves leveraging Spark's built-in functions to assign a unique, order-preserving identifier to each row. This allows us to treat the DataFrame structure sequentially and then easily filter for the highest identifier, thus retrieving the record that was last generated or ingested into the current DataFrame state. This advanced technique ensures that the operation remains performant even when working within a highly distributed computing environment, maintaining the core philosophy of PySpark DataFrame operations.

The Robust Solution: Leveraging Monotonic IDs

To safely and efficiently extract the last row in a distributed PySpark DataFrame, we must generate a column that captures the relative ordering of the rows as they exist immediately before the operation. The function `monotonically_increasing_id()` is perfectly suited for this task. By assigning a unique, non-decreasing ID to every record, we transform the challenge of identifying the "last" row into a simple aggregation problem: finding the row associated with the maximum ID value.

This method is highly preferred over indexing or collecting methods because it utilizes Spark's parallel processing capabilities. Instead of materializing the entire dataset to determine the length or index, we introduce a temporary key, perform a distributed maximum aggregation, and then select the corresponding record. This pattern is essential for maintaining stability and performance when processing petabyte-scale data where traditional sequential access methods fail. Remember that without a specific sorting operation (`orderBy`), the 'last row' generally refers to the row that happens to be positioned last based on the execution plan or ingestion order.

The core syntax employed for this task involves chaining several critical Apache Spark functions: `withColumn`, `monotonically_increasing_id`, `struct`, `max`, and `drop`. Mastering this combination allows developers to perform complex row-level selections efficiently in a distributed environment. This is the syntax that provides the most optimized way to identify and retrieve the final 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'))
```

Detailed Syntax Breakdown for Retrieving the Last Record

Understanding the components of the powerful one-liner provided above is crucial for its proper implementation and debugging. The operation works by embedding all necessary column information alongside a generated identifier, finding the maximum based on that identifier, and then unpacking the result. Each step ensures that the operation is executed in a parallel fashion across the cluster.

The process starts by invoking `df.withColumn('id', monotonically_increasing_id())`. This adds a temporary column named 'id' populated by the monotonically_increasing_id function. This function ensures that the resulting IDs are unique and non-decreasing, fulfilling our need for an ordered index to identify the final row. While the generated IDs are not necessarily contiguous, their monotonicity guarantees that the last row will possess the highest assigned value.

The most complex part of the syntax is the aggregation step: `select(max(struct('id', *df.columns)).alias('x'))`. Here, we use the struct function to combine the newly created 'id' column along with all existing columns (`*df.columns`) into a single complex column called 'x'. By applying the max aggregation function to this structured column, Spark implicitly uses the first field in the structure--the 'id'--for comparison. Because the 'id' is monotonic, maximizing the structured column effectively selects the entire row associated with the highest ID value.

Finally, we clean up the output using `select(col('x.*')).drop('id')`. The expression `col('x.*')` unpacks the structured column 'x' back into its original component columns (team, conference, points, assists, etc.), and the `drop('id')` function removes the temporary index column, leaving us with a final DataFrame containing only the desired last row and its original schema. This highly optimized approach avoids the inefficiency of collecting large datasets to the driver node, which is paramount in big data processing.

Setting Up the Practical PySpark Example

To demonstrate the effectiveness of using monotonic IDs for last row retrieval, let us apply this method to a concrete dataset. Suppose we are working with a PySpark DataFrame containing statistics about basketball players, where the order in which the rows appear is significant--perhaps representing chronological additions or a default ingestion order. We need to identify the very last recorded player entry.

The initial setup involves importing the necessary libraries, initializing the Spark Session, defining the raw data, and creating the DataFrame. It is essential to ensure that the Apache Spark environment is correctly configured before proceeding. The data below contains information regarding the team, conference, points scored, and assists recorded for several players. This DataFrame setup establishes the initial state upon which our last-row retrieval logic will operate.

We use the `SparkSession.builder.getOrCreate()` command to initialize the connection to our Spark cluster, followed by defining the list of data records and the corresponding column names. The `spark.createDataFrame(data, columns)` call then converts this localized Python structure into a resilient, distributed PySpark DataFrame, ready for scalable manipulation. Viewing the DataFrame using `df.show()` confirms the successful creation and the initial state of the data:

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 Last Row Extraction Logic

With the sample DataFrame initialized, our goal is to isolate the final entry, which in this case corresponds to Team 'C' with 5 points and 2 assists. While this specific row is visible in the output of `df.show()`, a production scenario requires programmatic extraction that is resilient to scaling. This is where the specialized aggregation technique proves its value. We must import the necessary functions, particularly `monotonically_increasing_id`, `max`, and `struct`, from `pyspark.sql.functions`.

The execution involves applying the transformation logic defined earlier directly to the DataFrame `df`. The result is stored in a new DataFrame called `last_row`, which, due to Spark's lazy evaluation, only holds the execution plan until an action, such as `show()`, is called. This approach ensures minimal resource utilization until the result is explicitly needed. It is important to note that this method relies on the order established when the DataFrame was materialized, meaning if the data were explicitly sorted before this operation, the "last row" would be the last element according to that sort key.

The code below demonstrates the seamless integration of the monotonic ID logic, performing the ID assignment, structured aggregation, selection of the maximum based on the ID, and finally, the clean-up of the temporary columns. The execution confirms that the desired record is successfully isolated and returned as a single-row DataFrame, showcasing the efficiency of this distributed selection technique:

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|
+----+----------+------+-------+

Step-by-Step Logic Confirmation

The successful extraction confirms the utility of this method for handling ordered retrieval in distributed environments. To solidify the understanding, it is beneficial to recap the sequence of operations that allows Apache Spark to identify the last record without requiring a complete data shuffle or collection into the driver program. This sequence is fundamental to optimized PySpark development.

The entire operation hinges on three primary steps executed within the distributed framework:

Step 1: Generating the Positional Key. We initiate the process by invoking the `withColumn` method paired with monotonically_increasing_id. This step adds a temporary 'id' column. This column acts as a positional surrogate key, ensuring that even if the underlying data partitions are reordered, we have a way to consistently identify the record that appeared last in the sequence before the current operation began. Although Spark does not guarantee the exact ordering of rows within partitions, the IDs ensure that the highest ID corresponds to the latest generated row key.

Step 2: Aggregation via Struct and Max. The subsequent stage involves using the struct function to group the 'id' and all other columns into a single composite type. We then apply the max aggregation. Crucially, when `max` is applied to a structured column, it prioritizes the maximum value of the first field--our 'id'. By maximizing based on the 'id', Spark efficiently finds the complete row corresponding to the highest positional identifier across all partitions, effectively isolating the last record in the DataFrame sequence.

Step 3: Cleanup and Final Projection. The final transformation uses the alias 'x' to reference the aggregated structured column. We use `col('x.*')` to project all fields contained within 'x' back into individual columns, restoring the original DataFrame schema. The temporary 'id' column, having served its purpose, is removed using the `drop('id')` function, resulting in a clean DataFrame containing only the single, desired last row, perfectly matching the schema of the source DataFrame.

The net result of this multi-step process is a highly efficient, distributed method for achieving a task that is traditionally simple in local computing environments but complex in distributed frameworks like Spark. This technique avoids the heavy performance penalty associated with approaches that require sorting the entire dataset or collecting it to the driver, making it suitable for truly massive data volumes.

Alternative Approaches and Performance Considerations

While the monotonic ID technique is highly recommended for its efficiency and stability, other methods exist, though they often come with significant caveats, especially concerning performance

in production environments. Understanding these trade-offs is crucial for any expert Apache Spark developer.

One common, simpler alternative mentioned initially is using the take() method combined with `count()`: `df.take(df.count())`. This approach is deceptive. The `df.count()` operation requires Spark to process the entire dataset just to determine the total number of rows (an action that often triggers a full job). Subsequently, `df.take(N)` collects the first N rows into the driver program. If N is equal to the total count of the DataFrame, this means collecting the entire dataset, which will crash the driver for large DataFrames. While acceptable for small, development-level DataFrames, this is strictly prohibited for large-scale production use cases.

Another approach involves sorting: if the "last row" is defined by a specific timestamp or sequence column, one might use `df.orderBy(col("timestamp").desc()).limit(1)`. This method guarantees the retrieval of the latest record based on a specific business logic. However, `orderBy` triggers a massive data shuffle across the cluster, which is computationally expensive, especially on unsorted or non-indexed large datasets. If the data is inherently ordered (or order does not strictly matter and we just need the last row generated internally), the monotonic ID method remains superior because it relies on partition-local generation followed by a minimal aggregation, minimizing expensive shuffles.

The choice of method must always balance correctness (does "last row" mean sequentially last or last by a specific key?) and efficiency. For simply finding the row that was physically last in the current, unordered sequence, the monotonically_increasing_id and max combination provides the best blend of simplicity and distributed performance.

Further Resources and Conclusion

The use of internal Spark functions like monotonically_increasing_id is a hallmark of efficient PySpark DataFrame manipulation. By understanding how to generate positional keys and leverage composite type aggregation using struct, developers can confidently handle tasks that require sequence-dependent retrieval across immense datasets.

For those interested in delving deeper into the specifics of the functions utilized in this robust solution, the official Apache Spark documentation provides comprehensive details. Specifically, the documentation for `monotonically_increasing_id` explains the guarantees and limitations of the ID generation process, which is critical when relying on this function for ordering or uniqueness constraints. Remember that maximizing performance in distributed computing requires moving away from iterative or sequential paradigms and embracing techniques that fully exploit parallel execution.

Documentation Note: You can find the complete documentation for the

monotonically_increasing_id function directly in the official PySpark API documentation, detailing its behavior and performance characteristics.

Related PySpark Tutorials

Mastering PySpark DataFrame operations extends beyond simple row retrieval. For those seeking to expand their knowledge of common distributed data tasks, the following list includes related tutorials covering essential manipulations, transformations, and actions frequently performed in big data workflows:

How to efficiently join multiple DataFrames using broadcast hints.

Techniques for window functions and calculating running totals.

Strategies for handling null values and schema evolution.

Optimizing data partitioning and caching strategies for iterative workloads.

Utilizing the max and other aggregation functions over grouped data.

These resources will provide foundational knowledge for performing other common tasks in PySpark, building upon the principles of distributed computation learned here.