Select Top N Rows in PySpark DataFrame (With Examples)

Authored by

stats writer

November 16, 2025

Introduction: Why Select Top N Rows in PySpark?

In the realm of big data processing, working with massive datasets stored in a **DataFrame** is a common task. Often, data scientists and engineers need to quickly inspect a subset of the data for debugging, quality checks, or simply to get a feel for the data structure and values. While the standard `show()` operation displays a limited set of rows (usually 20 by default), sometimes we specifically need to retrieve the absolute top N rows for further manipulation or focused analysis. **PySpark** offers highly efficient methods to achieve this, primarily through two distinct functions: `take()` and `limit()`. Understanding the fundamental differences between these two functions is crucial, as their choice significantly impacts execution strategy and the resulting data type, especially when dealing with distributed clusters.

Selecting the top N rows is not merely about visualization; it's about controlling data transfer across the distributed cluster. Since Apache Spark is inherently designed for parallel processing across multiple nodes, any action that brings data back to the local machine--the driver--must be managed carefully. When retrieving a small subset, it is vital to use the method that minimizes network overhead and prevents accidentally collecting too much data onto the driver node, which could lead to severe memory constraints or out-of-memory errors if mismanaged. This detailed guide provides a comprehensive examination of both `take()` and `limit()`, highlighting their unique features, implementation, and optimal use cases within a PySpark environment.

We will first establish the theoretical difference between these methods--one being an immediate action and the other a lazy transformation--and then follow up with practical, reproducible examples using a sample DataFrame setup. This approach will equip developers with the necessary knowledge to select the correct function based on whether their goal is local data inspection or continued distributed processing.

Overview of PySpark Methods for Row Selection

When aiming to retrieve the first N records from a **DataFrame**, PySpark provides two primary, optimized approaches: `take()` and `limit()`. While they appear to serve the same immediate purpose of subsetting data, they differ fundamentally in output type and their relationship to Spark's execution model. This distinction is paramount for maintaining robust performance and ensuring that downstream operations function correctly based on the expected data structure, whether local or distributed.

The first method, `take(N)`, is categorized as an **action**. Actions immediately trigger the execution of the necessary computation across the cluster and result in the data being collected and returned to the local Python environment (the driver program). Consequently, the output is not a distributed Spark object but a standard Python list of **Row** objects. This is ideal for quick, small-scale

integrations with non-Spark tools or for immediate debugging where the resulting data must be processed locally.

The second method, `limit(N)`, is classified as a **transformation**. Transformation operations are lazily evaluated; they do not trigger execution immediately. Instead, `limit()` returns a new, smaller DataFrame reference where a physical restriction is applied to the row count. The data remains distributed across the cluster until a subsequent action is explicitly called. This design makes `limit()` the superior choice for creating optimized subsets of data that require further large-scale distributed computations, preserving the efficiency of the Spark cluster.

Method 1: Utilizing the take() Action for Local Collection

The **`take()`** method provides the most direct means of accessing a specified number of rows from the beginning of a DataFrame. When this method is invoked, Spark initiates a job that reads data from the partitions, retrieves N rows, serializes them, and transfers them across the network back to the driver machine. This makes `take()` an immediate operation that forces computation.

df.take(10)

As a direct consequence of being an action, `take()` returns an **array** (a Python list) of the top 10 rows. These rows are now local to the Python interpreter running the Spark application. While convenient for quick checks, analysts must strictly control the argument N. If N is set too high--for instance, retrieving millions of records--the operation risks overwhelming the driver's memory resources, leading to the application crashing. Thus, `take()` should be reserved for scenarios where N is small, ensuring minimal impact on the cluster's stability and network bandwidth.

Method 2: Utilizing the limit() Transformation for Distributed Subsetting

In contrast, the **`limit()`** method provides a way to reduce the effective size of the DataFrame without collecting the data locally. Because it is a transformation, calling `limit(N)` merely updates the lineage graph of the DataFrame, instructing Spark's optimizer that any subsequent action should only consider the first N records available.

df.limit(10).show()

This method returns a new **DataFrame** object that contains the definition for the top 10 rows. This is highly advantageous because the data remains distributed and ready for further transformations (like filtering, joining, or complex aggregations) that leverage the parallelism of the Spark cluster. By using `limit()`, we are ensuring that subsequent computations operate only on the restricted dataset, optimizing performance without transferring data back to the driver. The `show()` call in the

example acts as the necessary action to visualize the result, but the underlying data structure remains a Spark DataFrame, suitable for large-scale operations.

Setting Up the PySpark Environment and Sample Data

To facilitate clear demonstration of these methods, we must first initialize the necessary Spark environment and construct a sample DataFrame. This preparatory step involves setting up the **SparkSession**, which serves as the fundamental entry point for all PySpark operations, and then defining a small dataset that allows us to easily verify the row selection results.

The following code block handles the imports, SparkSession creation, data definition, and DataFrame instantiation. The data structure is defined simply as a list of lists, followed by the definition of column names. The `createDataFrame` method binds these elements together into a distributed data structure. This small, clean dataset is essential for confirming that both `take()` and `limit()` accurately retrieve the intended subset of records.

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

# Define sample data structure
data = ,
,
,
,
]

Define descriptive column names
columns =

# Create DataFrame using the defined data and schema
df = spark.createDataFrame(data, columns)

# Display the initial DataFrame structure and content
df.show()

+----+----------+------+
|team|conference|points|
+----+----------+------+
A	East	11
A	East	8
A	East	10

B	West	6
B	West	6
C	East	5
+----+----------+------+

This base DataFrame, consisting of six rows, will now be used to demonstrate how to efficiently extract the top three records using both the action and the transformation methods available in PySpark.

Example 1: Selecting Top N Rows Using take()

We begin by applying the `take()` method to select the top 3 rows from our sample DataFrame. Because `take()` is an action, it executes immediately and returns the result directly to the driver machine as a list. This approach is most beneficial when the resulting data needs to be integrated into standard Python data structures or processed by non-Spark libraries, such as NumPy or Pandas (after conversion).

The following syntax executes the action, retrieving the specified three records. The output clearly shows the transformation from a distributed Spark object into a local Python list of `Row` objects. This confirmation of the output type is crucial for subsequent programming logic, as the data is now subject to Python's memory management and execution constraints, rather than Spark's distributed architecture.

 # Select top 3 rows from DataFrame
df.take(3)

This method successfully returns a local array containing the data for the top three rows of the DataFrame. Developers should be highly cognizant of the potential memory impact when using `take()`; even a relatively large value for N (e.g., 500,000 rows) can potentially consume gigabytes of driver memory if the rows are wide (contain many columns), leading to unexpected application instability. Therefore, `take()` must be used judiciously, typically only for small-scale sampling or verification purposes.

Example 2: Selecting Top N Rows Using limit()

Next, we apply the `limit()` method to achieve the same result of selecting the top three rows, but we maintain the distributed nature of the data. Since `limit()` is a transformation, we must chain an action, such as `show()`, to force the evaluation and display the results. This ensures that the data remains within the Spark ecosystem, ready for subsequent distributed operations.

The following code snippet demonstrates the proper implementation of this sequence. The result of `df.limit(3)` is a restricted DataFrame, and `.show()` then displays the output. Notice how the output format remains the structured tabular format associated with a DataFrame, underscoring the distinction from the list of rows returned by `take()`.

 # Select top 3 rows from DataFrame
df.limit(3).show()

```
+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|  A|      East|    11|      4|
|  A|      East|     8|      9|
|  A|      East|    10|      3|
+----+----------+------+-------+
```

This method is generally preferred in production pipelines because it scales effectively. The limitation is integrated into Spark's query plan, meaning that Spark executes the minimum work required to satisfy the limit N, potentially reading data only from the necessary partitions and then efficiently stopping the computation. This ensures high performance and resource conservation, especially when dealing with terabytes of data where minimizing data movement is critical.

Advanced Use Case: Combining limit() with Column Selection

The true power of `limit()` lies in its ability to be seamlessly integrated into a chain of transformations. A highly optimized and frequently utilized pattern involves combining `limit()` with the `select()` method. By selecting only the required columns before applying the row limit, we achieve both horizontal (column) and vertical (row) data reduction. This minimizes the data size being processed at an earlier stage in the pipeline, leading to faster execution times and reduced memory footprint across the executors.

In the example below, we restrict the DataFrame to just the `team` and `points` columns, and then apply the row limit of 3. This is an efficient way to sample the data for specific variables without loading the entire row width.

 # Select top 3 rows from DataFrame only for 'team' and 'points' columns
df.select('team', 'points').limit(3).show()

```
+----+------+
|team|points|
+----+------+
```

```
| A| 11|
| A| 8|
| A| 10|
+----+------+
```

Notice that only the top 3 rows for the **team** and **points** columns are shown in the resulting DataFrame. This chained execution demonstrates a best practice in PySpark development: always filter and select the necessary data early in the processing chain to minimize the volume of data Spark needs to manage throughout the subsequent transformations.

Conclusion and Best Practices

In summary, PySpark provides two robust methods for fetching the top N rows, each suited for a different requirement in the big data lifecycle. The critical distinction lies in their classification: `take()` is an immediate action that collects data locally, suitable only for small N values and quick inspection, while `limit()` is a lazy transformation that maintains data distribution, making it ideal for constructing efficient pipelines that require continued distributed processing on a restricted subset.

As a rule of thumb for large-scale production environments, `limit()` is the recommended default choice because it inherently aligns with Spark's lazy evaluation and distributed execution principles, offering superior scalability and cluster stability. Use `take()` sparingly--only when integration with local Python structures is unavoidable and you are absolutely certain that N is small enough to be safely handled by the driver node's memory. Mastering the appropriate use of these two functions is a foundational element of effective and high-performing PySpark development.