Select Top N Rows in PySpark DataFrame (With Examples)

Authored by

Mohammed looti

November 11, 2025

Introduction: Mastering Data Sampling in PySpark

When interacting with massive, distributed datasets managed by PySpark, data inspection becomes a critical, initial step. Whether you are debugging complex transformations, validating a schema, or performing rapid exploratory data analysis, you frequently need to isolate and examine a small subset of the records. Unlike traditional SQL environments where a simple `TOP` or `LIMIT` clause provides immediate results, the distributed nature of PySpark necessitates specialized methods.

This guide focuses on the two primary mechanisms for retrieving the first N rows from a DataFrame: the **take()** action and the **limit()** transformation. While both achieve the goal of selecting a specific number of records, their underlying execution models--and, consequently, their impact on cluster performance and driver memory--are fundamentally different. Understanding these distinctions is essential for writing efficient and scalable Spark applications.

We will thoroughly investigate the syntax, return types, and operational behavior of **take()** and **limit()**. The correct choice between them hinges entirely on your next step: do you need to materialize the subset of data locally on the driver program for immediate, non-distributed processing, or do you intend to maintain the result within the distributed cluster environment for subsequent large-scale transformations?

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

The **take()** function is categorized as an Action within the PySpark framework. Actions are distinguished by their behavior: when **take()** is called, it immediately triggers the execution of all preceding lazy transformations defined on the DataFrame lineage. Crucially, this action computes the specified number of rows (N) and forces the data to be collected back into the memory of the driver program.

The syntax for executing this action is straightforward, requiring only the number of rows desired as an argument:

df.take(10)

When executed, **take()** returns a standard Python array (specifically, a list of `Row` objects) containing the top N records. Because this method materializes the data locally, it is the most suitable choice for quick checks, logging, or when N is guaranteed to be a very small number (e.g., less than a hundred). However, developers must exercise significant caution: using **take()** with an excessively large N value can easily exhaust the memory resources of the driver machine, potentially leading to an `OutOfMemoryError` and crashing the application.

Method 2: Employing the limit() Transformation for Distributed Processing

In stark contrast to **take()**, the **limit()** function operates as a Transformation. Transformations are inherently lazy: they do not trigger immediate computation but instead define a new DataFrame based on the operation applied. The **limit()** transformation effectively marks the dataset, specifying that only the first N records should be retained when an Action is subsequently called upon the resulting structure.

To view the results of a **limit()** operation, it must be chained with an action, such as `.show()`:

df.limit(10).show()

The crucial distinction here is the return type: **limit()** returns a new, truncated DataFrame. This guarantees that the data remains distributed across the cluster executors, rather than being pulled into the driver program. This characteristic makes **limit()** the preferred approach when you need to apply further distributed operations, such as filtering, joining, or complex aggregations, on the limited subset of data in a scalable manner.

Practical Demonstration: Setting up the Sample PySpark DataFrame

To clearly illustrate the distinct outputs and execution behaviors of both **take()** and **limit()**, we will first establish a reproducible sample DataFrame. This setup requires initializing a SparkSession, which manages the connection to Spark functionality, and then defining a small dataset of team performance metrics.

The following code block defines our sample data, specifies column names, and instantiates the DataFrame named `df`, concluding with a display of the initial contents:

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|
+----+----------+------+
| A| East| 11|
| A| East| 8|
| A| East| 10|
| B| West| 6|
| B| West| 6|
| C| East| 5|
+----+----------+------+
```

It is important to note that in an unsorted DataFrame, the concept of "top N" refers only to the first N records encountered during the execution plan, which is generally deterministic but not based on any specific metric. If your requirement is to find the records with the highest points, for example, you must apply the `.orderBy()` transformation immediately before invoking **take()** or **limit()** to ensure the results are logically sorted.

Example 1: Retrieving Data Locally Using take()

We now demonstrate the use of the **take()** action to retrieve the first three records from our initialized `df`. This method provides the fastest way to pull a small, representative sample of data directly into the local Python environment for immediate handling, bypassing the need for further distributed operations.

```
#select top 3 rows from DataFrame
df.take(3)
```

The output clearly confirms that **take()** returns a standard Python array containing `Row` objects. These records are now disconnected from the distributed cluster environment, residing entirely within the memory space of the driver program. While excellent for quick checks or printing, this subset cannot be easily re-inserted into the Spark execution plan for further distributed computation without explicit conversion.

Example 2: Maintaining Distribution Using limit()

Next, we apply the **limit()** transformation. As **limit()** returns a new, lazily evaluated DataFrame, we must chain an Action, such as `.show()`, to force the evaluation across the cluster and display the resulting three records.

#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 output shows that **limit()** successfully produces a new, structurally intact DataFrame containing only the top 3 rows. The primary advantage of **limit()** lies in its seamless integration into complex data pipelines. Because it is a Transformation, it can be chained with other operations like `.filter()` or `.groupBy()`, ensuring that all processing remains scalable and distributed across the cluster executors.

For instance, we can combine **limit()** with `.select()` to narrow down both the number of rows and the columns displayed, providing a highly refined view of the data without sacrificing distributed efficiency:

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

This example demonstrates the powerful compositional nature of PySpark transformations, allowing developers to build sophisticated queries where data reduction (via **limit()**) occurs as part

of the overall distributed execution plan.

Key Differences Between take() and limit()

The strategic decision between using **take()** and **limit()** centers on managing data location: whether the subset needs to be processed locally on the driver or maintained distributively across the cluster executors. Understanding the core differences summarized below is vital for optimizing PySpark code:

Operational Type: take() is an **Action**, meaning it forces immediate evaluation and computation of the entire lineage. **limit()** is a **Transformation**, which is executed lazily only when a subsequent action is triggered.
Output Format: take() returns a native Python array (a list of `Row` objects). **limit()** returns a new **DataFrame** object.
Memory Implications: take() transfers all N rows back to the driver program memory, posing a significant risk of memory overflow if N is large. **limit()** retains the data distributed across the cluster, imposing minimal strain on the driver's memory.
Ideal Use Case: Use **take()** for diagnostic purposes, such as quickly retrieving the first few records (e.g., N < 100) for inspection or when the results must be processed using standard Python libraries. Use **limit()** when N is large or when the limited dataset is intended to be the subject of further, complex distributed transformations.

Additional Resources for PySpark Mastery

For developers committed to mastering distributed data processing, building upon this foundational knowledge of actions and transformations is key. The efficient selection and manipulation of data subsets demonstrated here are prerequisites for many common and essential tasks in PySpark.

To continue deepening your expertise, we recommend exploring tutorials that cover related tasks, such as distributed sorting, complex aggregations, and data partitioning strategies, all of which rely heavily on the correct application of transformations and actions.