How to Select the Top N Rows in a PySpark DataFrame

Authored by

stats writer

February 9, 2026

Introduction to Data Sampling and Inspection in PySpark

In the expansive realm of **Big Data**, the ability to efficiently inspect and manipulate subsets of data is a fundamental skill for any data engineer or scientist. When working with **PySpark**, the Python API for **Apache Spark**, users often find themselves needing to extract a specific number of records from a massive dataset to verify results or perform exploratory data analysis. The distributed nature of Spark requires a nuanced understanding of how data is retrieved from various clusters and brought into a readable format. Selecting the top N rows is not merely a matter of convenience; it is a critical step in debugging complex transformation pipelines and ensuring data quality across distributed systems.

The **DataFrame** abstraction in PySpark provides several built-in functions to facilitate the selection of top records. Unlike traditional **relational databases** where a simple "LIMIT" clause in **SQL** might suffice, PySpark offers different methods that behave uniquely depending on whether you want to return data to the local machine or maintain it as a distributed object. Understanding the distinction between returning a local list of rows and creating a new, smaller DataFrame is essential for writing optimized code that does not overwhelm the **Driver Node** memory.

In this comprehensive guide, we will explore the primary methodologies for selecting the top N rows in a **PySpark DataFrame**. We will delve into the technical nuances of the **take()** and **limit()** functions, providing clear examples and best practices for their implementation. By the end of this article, you will have a thorough understanding of how to manage row selection efficiently, balancing the needs for performance and data visibility within your Spark applications. Whether you are dealing with millions or billions of records, these techniques will serve as the cornerstone of your data manipulation toolkit.

The Core Mechanics of the PySpark DataFrame API

At its heart, **Apache Spark** operates on the principle of **Distributed Computing**. This means that data is partitioned across multiple nodes in a cluster, and operations are performed in parallel. When you interact with a **DataFrame**, you are essentially working with an immutable, distributed collection of data organized into named columns. The **PySpark** API acts as a bridge, allowing Python developers to leverage the power of Spark's JVM-based engine through familiar syntax and high-level abstractions.

One of the most important concepts to grasp in Spark is **Lazy Evaluation**. This means that Spark does not execute transformations immediately; instead, it builds a logical execution plan called a Directed Acyclic Graph (DAG). Actual execution only occurs when an "action" is called. Methods like **take()** are considered actions because they trigger the computation and return results to the driver program. Conversely, **limit()** is a transformation that returns a new DataFrame, allowing

Spark to optimize the execution plan further before any data is actually moved or computed.

To select the top N rows effectively, one must consider the **Serialization** and network overhead involved. Selecting a few rows for inspection is inexpensive, but attempting to retrieve a large percentage of a massive dataset to the driver can cause **Out of Memory (OOM)** errors. Therefore, choosing between different row selection methods requires an understanding of where the data will reside--either as a collection in the driver's memory or as a smaller partition within the distributed cluster executors.

Exploring the take() Method for Immediate Data Retrieval

The **take()** method is one of the most direct ways to retrieve a specific number of rows from a **PySpark DataFrame**. When you call **df.take(N)**, Spark scans the partitions of the DataFrame and pulls the first N records it encounters. These records are then serialized and sent over the network to the **Driver Node**, where they are consolidated into a standard Python list of Row objects. This makes **take()** an ideal choice for quick inspections or when you need to pass data into local Python libraries like **Pandas** or Matplotlib.

A significant advantage of **take()** is its efficiency in terms of execution time for small samples. Spark does not necessarily need to scan the entire dataset to fulfill a **take(N)** request; it only processes enough partitions to find the requested number of rows. This can save significant computational resources compared to actions that require a full shuffle or global scan. However, it is crucial to remember that the order of rows returned by **take()** is not guaranteed unless the DataFrame has been explicitly sorted using an **orderBy()** or **sort()** transformation beforehand.

From a **Memory Management** perspective, developers must exercise caution when using **take()** with large values of N. Since the resulting list is stored in the driver's memory, requesting an excessively large number of rows can lead to a crash. It is generally recommended to use **take()** for small samples (e.g., N < 1000) intended for manual verification or lightweight processing. For larger subsets that require further distributed processing, other methods like **limit()** are far more appropriate and scalable.

Leveraging the limit() Method for Scalable Data Transformation

The **limit()** method offers a fundamentally different approach to row selection compared to **take()**. While **take()** is an action that returns a local Python object, **limit(N)** is a transformation that returns a new **DataFrame** containing at most N rows. This distinction is vital because the resulting DataFrame remains distributed across the cluster. This allows the developer to continue performing **PySpark** operations, such as joins, aggregations, or filtering, on the reduced dataset without ever pulling it to the local driver node.

In practice, **limit()** is frequently used in **ETL (Extract, Transform, Load)** pipelines where only a sample of the data is needed for downstream processing. Because it returns a DataFrame, it benefits from the **Spark Catalyst Optimizer**. The optimizer can push the limit operation down to the data source, significantly reducing the amount of data read from disk. For example, when reading from a **Parquet** file, Spark can stop reading once the limit threshold is met, leading to massive performance gains in large-scale environments.

To view the results of a **limit()** operation, it is often paired with the **show()** action. Running **df.limit(10).show()** effectively truncates the dataset to 10 rows and then prints them in a formatted table to the console. This combination provides the best of both worlds: the scalability of a distributed transformation and the visibility of an action. It is the preferred method for developers who want to inspect data within the Spark environment without converting it into local Python types until absolutely necessary.

Critical Differences Between Local Collection and Distributed Transformation

Understanding the architectural difference between **take()** and **limit()** is key to mastering **Apache Spark**. The **take()** method results in a **Collection** in the driver's memory. This is a synchronous operation that blocks further execution until the data is retrieved. Once the data is in the driver, it is no longer under the management of Spark's distributed engine; it is a standard Python list. This is useful for integration with the **Python** ecosystem but lacks the parallel processing capabilities of Spark.

On the other hand, **limit()** maintains the data within the **Distributed Computing** framework. The rows selected by **limit()** reside on the executors, and any subsequent operations on that limited DataFrame will still be executed in parallel across the cluster. This is essential for maintaining performance when the "subset" of data is still relatively large. For instance, selecting the "top" 1,000,000 rows of a billion-row dataset is a task perfectly suited for **limit()**, but likely impossible for **take()** on a standard driver node.

Furthermore, the **API** design reflects these use cases. **take()** returns a List of Row objects, which allows for index-based access in Python (e.g., `results`). **limit()** returns a DataFrame object, which supports Spark-specific methods like `select()`, `where()`, and `groupBy()`. Choosing the right tool depends on whether your next step involves native Python processing or further Spark-based transformations. Mixing these up can lead to inefficient code and unnecessary data movement across the **Network**.

Practical Demonstration with Code Examples

To illustrate these concepts, let us look at a practical implementation using a **PySpark DataFrame**. First, we must initialize a **SparkSession**, which serves as the entry point to Spark functionality.

The following example demonstrates how to create a simple DataFrame containing team names, conferences, and points, which we will use to test our row selection methods.

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

# Define raw data
data = ,
,
,
,
,
]

Define schema column names
columns =

# Create DataFrame using data and column names
df = spark.createDataFrame(data, columns)

# Display the full DataFrame
df.show()

+----+----------+------+
|team|conference|points|
+----+----------+------+
A	East	11
A	East	8
A	East	10
B	West	6
B	West	6
C	East	5
+----+----------+------+

Now, let us apply the **take()** method to retrieve a specific number of rows. As discussed, this will return an array of Row objects to the driver. This is particularly useful when you need to programmatically access specific fields in the first few records using Python logic. The output shows the structured Row format, including the column names and their associated values for each record retrieved.

 # Select the top 3 rows from the DataFrame using take()

top_rows_list = df.take(3)

# Print the resulting Python list
print(top_rows_list)

Alternatively, if we wish to keep the data in a DataFrame format, we use the **limit()** method. This is the preferred approach for visual inspection and for creating smaller subsets of data for further analysis. By calling **show()** after **limit()**, we can see the data in a clean, tabular format. We can also chain other operations, such as selecting specific columns, before applying the limit to further refine our output.

 # Select the top 3 rows using limit() and display them
df.limit(3).show()

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

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

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


Performance Considerations and Memory Management

When working with **Apache Spark**, performance is often the primary concern. Selecting the top N rows might seem like a simple task, but in a distributed environment, it involves coordination between the driver and multiple executors. One of the common pitfalls is assuming that the "top" rows will be the same every time you run the command. Because Spark processes data in parallel

across partitions, the order is non-deterministic unless you include a **sort()** operation. For consistent results, always use `df.orderBy("column").limit(N)`.

Another critical aspect is **Memory Management**. Every time you use **take()** or **collect()**, you are moving data from the distributed cluster to a single machine (the driver). If N is large, this can cause the driver to exceed its allocated **JVM** heap space. It is a best practice to keep the number of rows returned to the driver as small as possible. If you need to process a large subset of data, it is much more efficient to write the limited DataFrame to a storage system like **Amazon S3** or **HDFS** rather than bringing it to the driver.

Lastly, consider the impact on the **Network** bandwidth. Moving data from executors to the driver involves **Serialization** and network transfer. While selecting 10 rows is negligible, selecting 100,000 rows can introduce significant latency. Using **limit()** avoids this transfer entirely until an action like **show()** or **write()** is called, and even then, **show()** only transfers a small amount of data to satisfy the display requirements. Always favor transformations over actions when building production-grade data pipelines.

Supplementary Methods: head(), first(), and show()

Beyond **take()** and **limit()**, the **PySpark** API provides other convenience methods for row selection. The **head(N)** function is essentially an alias for **take(N)**; it returns a list of N rows to the driver. It is often preferred by users coming from the **Pandas** library due to the familiar naming convention. Similarly, the **first()** method is a specialized version of **take(1)**, returning only the very first row of the DataFrame. These methods are excellent for checking schema adherence or inspecting the first few values of a transformation.

The **show(N)** method is arguably the most common tool used during development. Unlike **take()**, it does not return a value to the Python environment; instead, it prints the first N rows directly to the console output. By default, **show()** displays 20 rows and truncates long strings, but these parameters can be adjusted (e.g., `df.show(5, truncate=False)`). It is important to note that **show()** is an action, meaning it triggers computation. It is purely for human consumption and cannot be used to pass data to subsequent functions.

In summary, the choice of method depends on your specific objective. Use **limit()** when you need a smaller **DataFrame** for further Spark work. Use **take()** or **head()** when you need a local Python list for processing outside of Spark. Use **first()** for a single record check, and use **show()** for quick visual debugging. By selecting the most appropriate method for the task at hand, you ensure that your PySpark applications remain performant, readable, and robust against memory issues.