How to Display Full Column Content in PySpark

Authored by

stats writer

February 4, 2026

When working with large datasets in Apache Spark, specifically through its Python interface, PySpark DataFrame operations, developers frequently encounter challenges with data visualization, particularly when dealing with columns containing extensive text or complex structured data. By default, PySpark implements a character limit for column display during output viewing, a feature designed to prevent overwhelming the console or notebook output with overly wide results. However, for effective Data analysis and debugging, seeing the complete, untruncated content of a column is often essential for verifying data integrity and understanding subtle differences in record entries.

The core mechanism for managing how data is displayed in PySpark is the `show()` action, which is integral to the PySpark DataFrame API. To overcome the default truncation behavior and ensure the full content of any column is visible, you must explicitly override this setting. The most straightforward approach involves utilizing the `truncate` parameter within the show() function, setting it to either `False` or `0`. Additionally, while less common for quick inspection, alternative methods like combining `select()` and `collect()` allow for retrieving column values as a native Python list, thus bypassing Spark's display constraints entirely.

This guide delves into these methods, providing precise, valid code examples demonstrating how to reliably force a PySpark DataFrame to present all column content without abbreviation. Mastering these display controls is a fundamental skill for anyone performing serious data manipulation and quality checks using Apache Spark environments, ensuring that critical data remains visible and accessible during the development lifecycle.

PySpark: Show Full Column Content

Understanding Data Truncation in PySpark

The default behavior of the show() function in PySpark is designed for efficiency and readability within standard terminal environments. By default, when a PySpark DataFrame is printed, column values exceeding 20 characters are truncated. This is a sensible default for general usage, as it prevents excessively wide tables that are difficult to scan quickly. However, this feature becomes a hindrance when the actual length or content structure of a column is critical for verification.

When truncation occurs, PySpark replaces the remaining characters with an ellipsis (`...`), signaling that the displayed data is incomplete. This can lead to misleading interpretations, especially if the differences between two records lie just beyond the 20th character. For developers working on detailed text processing, natural language processing (NLP) tasks, or deep Data analysis, it is imperative to override this limit to ensure data integrity checks are performed against the complete underlying data.

Fortunately, Apache Spark provides explicit controls to manage this display behavior directly within the function call itself, offering temporary solutions without altering global settings. There are two primary parameters that achieve the goal of full column visibility, both focusing on the `truncate` argument of the show() function.

The Primary Solution: Utilizing the `show()` Function

The most common and immediate solution to display all column content is by manipulating the `truncate` parameter within the show() function. This method is preferred because it is non-persistent; it only affects the output of that specific function call, leaving global Spark configurations untouched. This ensures that subsequent display calls, perhaps by other users or other parts of the script where brevity is necessary, remain unaffected by the change.

There are two acceptable values for the `truncate` parameter that will disable the column width limit: setting it to a boolean `False` or setting it to an integer `0`. Both syntaxes are valid and yield the exact same untruncated result, though using `False` is often considered clearer in terms of intent, explicitly indicating that truncation should be disabled. The underlying implementation of the show() function recognizes both inputs as a command to display the entire string content of every column.

Before demonstrating the full display methods, let us first establish a sample PySpark DataFrame where truncation is evident. This setup uses standard PySpark initialization methods and creates a small, representative dataset where one column intentionally exceeds the 20-character default width, simulating real-world data storage scenarios.

Below is the setup code for our example DataFrame:

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 with default truncation
df.show()

+-----+--------------------+-----+
|store| employees|sales|
+-----+--------------------+-----+
| A|Andy Bob Chad Dou...| 136|
| B| Frank Henry| 223|
| C|Ian John Ken Liam...| 450|
| D| Oscar Prim| 290|
| E| Quentin Ross Sarah| 189|
+-----+--------------------+-----+
```

As clearly demonstrated in the output above, rows in the `employees` column, such as the entries for 'A' and 'C', are cut off and appended with an ellipsis (`...`). This truncation occurs because the default display width limit of 20 characters has been exceeded. This is precisely the issue we aim to resolve using the `truncate` parameter.

Detailed Implementation: Using `truncate=False`

The most readable method for disabling truncation is by passing the boolean value `False` to the `truncate` parameter within the show() function. When this argument is set, PySpark bypasses the standard width check and dynamically adjusts the column display width to accommodate the longest string present in that column across all retrieved rows. This ensures complete visibility of all textual data.

Using `truncate=False` is particularly useful during exploratory Data analysis or when debugging complex ETL pipelines where verifying the exact string contents after transformations is paramount. It guarantees that the visual representation of the DataFrame accurately reflects the underlying data stored in memory, providing confidence in subsequent processing steps that rely on text integrity.

We apply this method to our sample DataFrame `df`:

```
#view dataframe with full column content using truncate=False
df.show(truncate=False)

+-----+--------------------------+-----+
|store|employees |sales|
+-----+--------------------------+-----+
|A |Andy Bob Chad Doug Eric |136 |
```

```
|B |Frank Henry |223 |
|C |Ian John Ken Liam Mike Noah|450 |
|D |Oscar Prim |290 |
|E |Quentin Ross Sarah |189 |
+-----+--------------------------+-----+
```

Observing the new output, the `employees` column now displays its full content without any ellipsis or missing characters. The column width has automatically expanded to fit the longest string ("Ian John Ken Liam Mike Noah"). This method provides a clear, transient solution for inspecting data without permanently altering the environment.

The Alternative Syntax: Setting `truncate=0`

An equally effective, though perhaps slightly less intuitive, method is setting the `truncate` parameter to the integer value `0`. In the context of PySpark's display parameters, setting `truncate` to `0` is an instruction interpreted as "set the maximum display width to zero," which effectively means "do not truncate at all." While the resulting output is identical to using `truncate=False`, some developers prefer this integer syntax as it aligns with configurations in other programming or database environments where a zero value often signifies 'no limit'.

This alternative syntax serves as a robust backup and allows for flexibility in coding styles. Regardless of whether `False` or `0` is chosen, the fundamental result--full column visibility--remains the same. It is important to remember that using `show()`, even with `truncate=False`, still only displays the first 20 rows of the PySpark DataFrame by default. If you need to see more rows, you must also specify the `n` parameter (e.g., `df.show(n=100, truncate=False)`).

We implement the `truncate=0` method on the same DataFrame:

```
#view dataframe with full column content using truncate=0
df.show(truncate=0)
```

```
+-----+--------------------------+-----+
|store|employees |sales|
+-----+--------------------------+-----+
|A |Andy Bob Chad Doug Eric |136 |
|B |Frank Henry |223 |
|C |Ian John Ken Liam Mike Noah|450 |
|D |Oscar Prim |290 |
|E |Quentin Ross Sarah |189 |
+-----+--------------------------+-----+
```

As verified by the output, the full contents of the `employees` column are displayed, confirming that `truncate=0` is an interchangeable command with `truncate=False` for disabling output abbreviation. This consistency across different input types provides reliability in data inspection workflows.

Advanced Display Control: Configuring Spark Properties

While the `truncate` parameter in the show() function offers a temporary solution, in continuous integration environments or long-running interactive sessions where full visibility is consistently required, developers may prefer to set a global configuration. This ensures that every subsequent call to `df.show()` (without specific parameters) adheres to the new, expanded display width.

This global setting is managed through Spark Configuration properties. The relevant property is `spark.sql.repl.outputMaxStringLength`. By default, this property is set to 20, aligning with the default truncation length. By setting this property to a very high number (e.g., 1000) or, counter-intuitively, to `0` (which is sometimes interpreted by Spark as infinite length, similar to the `truncate=0` behavior), you can globally modify the default behavior.

To configure this globally within a PySpark session, you would use the following command structure:

```
spark.conf.set("spark.sql.repl.outputMaxStringLength", 1000)
# Or, to effectively disable truncation entirely:
spark.conf.set("spark.sql.repl.outputMaxStringLength", 0)
```

It is important to understand the trade-offs of using global configuration. While it simplifies subsequent code, it can potentially lead to extremely wide, unreadable output if the dataset contains truly massive strings. Therefore, relying on the explicit `truncate=False` parameter within the show() function remains the recommended best practice for isolated data visualization tasks, reserving global configuration for specific environments where untruncated output is the standard operational requirement.

Alternative Viewing Methods: `select()` and `collect()`

Beyond the table-formatting provided by `show()`, another powerful technique for viewing full column content, especially for inspection or debugging, is converting the specific column into a native Python object. This approach completely bypasses Spark's internal formatting rules, presenting the data exactly as it exists in memory within the context of standard Python data structures.

This technique involves chaining the `select()` and `collect()` actions. First, `select()` is used to isolate the column of interest (e.g., 'employees'). Then, the `collect()` action is called. `collect()` retrieves all elements of the PySpark DataFrame (or the selected column's contents) and returns them as a list of `Row` objects to the driver program. If only one column is selected, the list usually contains the string values directly, which are guaranteed not to be truncated.

Consider the practical application using our example. If we only needed to verify the employee names list, we could execute:

df.select("employees").collect()
 # Output is a list of Row objects:
#

If a simpler list of strings is desired, a list comprehension can be applied immediately after `collect()`:

for row in df.select("employees").collect()]
 # Output:
#

This method is highly effective for detailed inspection and integration with other Python libraries but comes with a major caveat: `collect()` moves all data to the driver node. If the column contains billions of rows, this operation can cause the driver program to run out of memory, resulting in an Out of Memory (OOM) error. Therefore, `collect()` should be used judiciously, typically only on small subsets of data or when confirming the contents of a small, pre-filtered result set.

Summary of Best Practices for Data Visualization

Effective data visualization in Apache Spark requires a balanced approach, considering both the need for detailed inspection and the constraints of memory and screen space. Developers should adopt a hierarchy of methods based on the context of their operation. For quick, localized checks, the explicit use of the `truncate` parameter is undeniably the best method.

A recommended set of practices for handling data display includes:

Default Quick Check: Use `df.show()` for a high-level view of structure and basic contents, accepting the default truncation.
Detailed Inspection (Transient): Use `df.show(truncate=False)` or `df.show(truncate=0)` when verifying long string content or debugging data transformations, ensuring that the full length is captured.

Debugging Small Data (Python Integration): For scenarios involving small result sets or when integrating with non-Spark Python functions, use `df.select(...).collect()` to retrieve data into native Python lists.
Environment Configuration (Persistent): Only use `spark.conf.set("spark.sql.repl.outputMaxStringLength", N)` if the entire development or analysis environment requires consistently untruncated output, understanding the potential impact on visual clutter.

By judiciously applying these techniques, data scientists and engineers can maintain high visibility into their data quality and processing results, optimizing the iterative process inherent in modern Data analysis pipelines running on distributed systems like Apache Spark.

Note: You can find the complete documentation for the PySpark show() function on the official Apache Spark API documentation page.

Further Learning in PySpark

The ability to control output display is just one fundamental aspect of working efficiently with PySpark. Mastering other common tasks is essential for leveraging the full power of distributed processing. Below are examples of other critical skills and functions commonly required in data workflows:

How to perform efficient joins across large datasets.
Techniques for handling missing values (nulls) using functions like `fillna()` or `na.drop()`.
Applying User Defined Functions (UDFs) for custom column transformations.
Optimizing data reading and writing operations using formats like Parquet or ORC.

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