How to Add Row Numbers as a New Column in PySpark

Authored by

stats writer

February 7, 2026

Generating sequential row numbers is a fundamental operation in traditional database management systems and data processing environments. However, implementing straightforward indexing can be complex within distributed computing frameworks like PySpark. Unlike pandas or SQL databases where physical order is often guaranteed, PySpark operates on distributed, immutable data structures known as DataFrames, meaning rows inherently lack a fixed, inherent physical order across partitions. Therefore, creating a stable, deterministic row index requires explicit use of window functions or other specialized tools provided by the Spark API.

The necessity for an index column often arises when users need to perform row-wise analysis, sequence operations, or simply track the original ingestion order of the data. While it is possible to use the highly performant function monotonically_increasing_id() for generating unique IDs, the most reliable and common method for deterministic row numbering, starting precisely from 1, involves leveraging the powerful row_number() function in conjunction with a specialized Window specification. This technique ensures that even in a highly parallel execution environment, each record receives a unique and sequential integer ID, crucial for subsequent data transformations and reporting.

PySpark: Add New Column with Row Numbers

Method 1: Utilizing the `row_number()` Function with Window Specifications

The standard, highly recommended approach for generating sequential, gap-free row numbers in a DataFrame is through the use of the row_number() window function. This function assigns a sequential rank starting from 1 to each row within a partition or the entire dataset, depending on the defined Window specification. Since row numbering requires a defined order to be stable, we must provide an ordering clause, even if the resulting index sequence itself is arbitrary relative to the data values.

To implement this across the entire DataFrame without partitioning (meaning we want a single, continuous sequence of numbers), we define a Window object that orders the data based on an arbitrary, yet stable, column or value. Since we are interested in indexing the existing rows regardless of their content values, we must force a global sort operation. The following syntax demonstrates the precise requirement to achieve this global row indexing:

```
from pyspark.sql.functions import row_number,lit
from pyspark.sql.window import Window

# Define a global Window specification.
# We use lit('A') to create a dummy column for orderBy, forcing a single global order across the entire DataFrame.
```

```
w = Window().orderBy(lit('A'))
df = df.withColumn('id', row_number().over(w))
```

This implementation effectively adds a new column, designated here as **id**, which will contain row numbers ranging sequentially from 1 up to N, where N is the total count of rows in the DataFrame. It is important to note that the use of **orderBy(lit('A'))** is a performance-intensive operation in Spark because it necessitates shuffling all data onto a single partition prior to applying the ordering logic, which is necessary to guarantee a continuous, non-gapped sequence. Users must be aware of the computational cost involved when performing global numbering on extremely large datasets.

Detailed Breakdown of the Window Components

The Window function in PySpark is central to performing calculations across sets of rows that are related to the current row. For row numbering, we leverage two main concepts within the window specification: partitioning and ordering. When we instantiate **Window()** without any arguments, we implicitly define a single partition encompassing the entire DataFrame, which is essential for global numbering.

The second critical component is the **orderBy()** clause. Since the row_number() function requires a deterministic order to assign sequential ranks, we must specify how the rows are sorted before indexing occurs. In the previous example, we used **lit('A')**, which stands for "literal 'A'". This creates a temporary, constant column used solely for the sorting operation. Because every row has the same sorting value ('A'), Spark must consolidate the data to apply this ordering, ensuring the generated sequence is continuous and not split across partitions, thereby achieving the desired effect of a global row index.

Alternatively, if the user wishes to assign row numbers based on a meaningful criterion--for instance, indexing students by their scores--the **orderBy()** function would take the relevant column name (e.g., **orderBy('score', ascending=False)**). However, when the goal is purely to index the rows as they currently exist in memory or storage without regard to specific column values, the arbitrary literal approach is adopted as a technical necessity to satisfy the requirements of the Window function syntax.

Practical Example: Setting Up the Initial PySpark DataFrame

To illustrate the implementation of row numbering, let us first establish a sample DataFrame. This dataset represents performance metrics for different teams and conferences, which will serve as our input data structure before the indexing process begins. The setup requires initializing a **SparkSession** and defining the schema and data contents, ensuring a stable environment for demonstration.

The following code block demonstrates the necessary steps to define and visualize the initial data structure within a PySpark environment. We import the necessary components, define our data array and column names, and then construct the DataFrame:

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

# Define the sample input data
data = ,
,
,
,
,
]

Define column names
columns =

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

# Display the 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
+----+----------+------+

As evident from the output, the initial DataFrame possesses three core columns. Currently, there is no explicit index column that uniquely identifies the sequential position of each record. Our objective is to augment this structure by introducing a new column that achieves precisely this purpose, allowing us to reference rows by their numerical sequence.

Implementing the Global Row Numbering Logic

Having established our initial DataFrame, we proceed with applying the window function logic detailed earlier. This step requires the import of the necessary functions--**row_number** and **lit** from **pyspark.sql.functions**, and **Window** from **pyspark.sql.window**--to define the mechanism that generates the sequential indices. We will use the **withColumn** transformation to append this newly calculated index column to our existing dataset.

Execution of the following syntax will transform the DataFrame by calculating the global row number based on the imposed arbitrary ordering. This ensures every row, regardless of the physical distribution across the cluster, is assigned a unique, gap-free integer starting from 1. We then visualize the updated DataFrame to confirm the successful creation and population of the new index column.

```
from pyspark.sql.functions import row_number,lit
from pyspark.sql.window import Window

# Define the arbitrary global order Window
w = Window().orderBy(lit('A'))
df = df.withColumn('id', row_number().over(w))

# View the updated DataFrame
df.show()
```

```
+----+----------+------+---+
|team|conference|points| id|
+----+----------+------+---+
|  A|      East|    11|  1|
|  A|      East|     8|  2|
|  A|      East|    10|  3|
|  B|      West|     6|  4|
|  B|      West|     6|  5|
|  C|      East|     5|  6|
+----+----------+------+---+
```

The resulting output clearly demonstrates the addition of the **id** column, successfully populated with sequential row numbers from 1 to 6, corresponding to the total number of records in the dataset. This new index column can now be utilized for filtering, joining, or other analytical tasks where a guaranteed sequential order is required. It is important to remember that the order in which the rows appear might differ slightly on repeated runs unless a specific, stable ordering column (like a timestamp or a primary key) is used in the **orderBy()** clause instead of the arbitrary

lit('A').

Post-Processing: Reordering Columns for Improved Readability

While the new **id** column successfully contains the desired row numbers, it often appears as the final column in the schema, which can be less intuitive for users who expect index columns to lead the dataset. To improve the usability and readability of the resulting DataFrame, it is common practice to explicitly reorder the columns using the **select()** method, moving the identifier column to the front.

The **select()** transformation allows precise specification of the output schema order. By listing **id** first, followed by the original columns, we ensure that the index is immediately visible. This step does not alter the data itself but significantly enhances the presentation, making the DataFrame easier to interpret and analyze. This process is demonstrated below:

 # Move 'id' column to the front of the DataFrame schema
df = df.select('id', 'team', 'conference', 'points')

# View the updated DataFrame with reordered columns
df.show()

```
+---+----+----------+------+
| id|team|conference|points|
+---+----+----------+------+
| 1| A| East| 11|
| 2| A| East| 8|
| 3| A| East| 10|
| 4| B| West| 6|
| 5| B| West| 6|
| 6| C| East| 5|
+---+----+----------+------+
```

The final output confirms that the **id** column is now correctly positioned as the leading column, providing a clear and immediate reference point for each row. Furthermore, a crucial technical detail regarding the use of the literal value must be highlighted: the syntax **lit('A')** within the **orderBy()** clause serves purely as a placeholder to satisfy the mandatory ordering requirement of the Window function. The specific content of the literal (e.g., 'A', 'Z', or 1) is entirely arbitrary; replacing 'A' with any other constant value will yield the same functional result--a globally indexed DataFrame.

Alternative Method: Utilizing `monotonically_increasing_id()`

While row_number() is ideal for guaranteed sequential indices starting from 1, PySpark offers an alternative, less computationally expensive function for generating unique identifiers: monotonically_increasing_id(). This function is often preferred when absolute sequential order (1, 2, 3...) is not required, but rather a unique, non-decreasing identifier for every row is sufficient.

The primary advantage of monotonically_increasing_id() is that it avoids a full dataset shuffle, unlike the global ordering required by row_number(). Instead, it generates IDs based on the partition index and the record index within that partition. This results in IDs that are unique and increasing, but they are not necessarily contiguous or guaranteed to start at zero or one, and the resulting ID values can be very large **long** integers.

If performance is critical and a perfect sequence (1, 2, 3...) is not mandated, this function provides a strong option. It ensures that any subsequent rows added to the DataFrame will receive a higher ID than existing rows, maintaining the monotonicity principle. However, users must understand the trade-off: speed and efficiency gain in exchange for losing the strict sequential numbering that mirrors human-readable indexes. The implementation simply involves using **withColumn** and calling the function directly:

from pyspark.sql.functions import monotonically_increasing_id

df_monotonic = df.withColumn('monotonic_id', monotonically_increasing_id())

df_monotonic.show()

Example Output (IDs vary based on partition configuration):
+----+----------+------+-------------+
|team|conference|points|monotonic_id|
+----+----------+------+-------------+
| A| East| 11| 0|
| A| East| 8| 1|
| A| East| 10| 2|
| B| West| 6| 8589934592|
| B| West| 6| 8589934593|
| C| East| 5| 8589934594|
+----+----------+------+-------------+

Key Considerations for Production Environments

When applying row numbering techniques in large-scale production PySpark jobs, the choice

between row_number() and monotonically_increasing_id() becomes a crucial design decision governed by performance and requirement constraints. The global ordering required by the Window function forces a costly operation where all data must be collected and sorted, potentially creating a bottleneck if the DataFrame exceeds the memory capacity of a single executor.

For scenarios where row numbering must restart sequentially within logical groups (e.g., assigning ranks within each 'team' or 'conference'), the power of the Window function is fully realized by adding a **partitionBy()** clause. This localizes the numbering operation to smaller chunks of data, significantly reducing the required shuffle magnitude and improving scalability. If global, consecutive numbering is truly non-negotiable for a massive dataset, engineers should consider temporary repartitioning strategies to manage the shuffle load, although this remains an expensive procedure.

Conversely, if the primary objective is generating a unique key for tracking or internal joins--and the aesthetic requirement of starting exactly at 1 is secondary--the monotonically_increasing_id() function is overwhelmingly the more performant choice. It operates locally within partitions, providing high throughput for indexing operations on petabyte-scale data where a global sort is impractical or impossible. Understanding these operational differences ensures that the correct tool is selected based on the functional requirements and available cluster resources.

Summary of Row Numbering Techniques

In summary, adding a sequential row number column in PySpark requires careful architectural consideration due to the distributed nature of the DataFrame. The method utilizing row_number() combined with a globally ordered Window specification is the definitive technique for achieving a continuous, 1-to-N index, as demonstrated in the primary example.

The key steps involve:

Defining a **Window** specification that spans the entire DataFrame (no **partitionBy**).
Using **orderBy(lit(constant_value))** to satisfy the requirement for deterministic ordering, even if the order itself is arbitrary.
Applying **row_number().over(w)** to generate the index.

Conversely, for high-performance use cases where uniqueness and monotonicity are prioritized over starting exactly at 1 and ensuring continuity, the monotonically_increasing_id() function offers a highly scalable, shuffle-free alternative. Developers should select the method that best balances data requirements (strict sequence vs. unique ID) and performance objectives (shuffle avoidance vs. global order guarantee).

Conclusion and Further Exploration

Mastering window functions is essential for advanced data manipulation in PySpark, and row numbering serves as an excellent foundational example. Whether you require a traditional index starting at one for reporting purposes or a robust, unique identifier for distributed processing, PySpark provides the tools necessary to solve these indexing challenges efficiently.

To further enhance your proficiency in PySpark, consider exploring other common analytical tasks that leverage these powerful functionalities:

Calculating running totals or cumulative sums.
Ranking data based on specific criteria using **rank()** or **dense_rank()**.
Performing lead and lag analysis to compare values across preceding or succeeding rows.

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