How to Add Row Numbers as a New Column in PySpark DataFrames

Authored by

stats writer

January 20, 2026

The Need for Row Numbers in PySpark

Generating sequential row numbers is a fundamental operation in data processing, often required for tasks like creating primary keys, sampling records, or performing time-series analysis. When working within the distributed computing framework of PySpark, this task requires specialized techniques due to the parallel nature of DataFrame processing. Standard pandas indexing methods are not applicable in this environment, necessitating the use of Window functions. This article provides a definitive, step-by-step guide on how to efficiently and reliably add a new column containing row numbers to any PySpark DataFrame.

The core challenge in PySpark is ensuring a deterministic order when assigning sequential identifiers across multiple partitions. Unlike traditional relational databases where implicit ordering often exists, Spark distributes data across a cluster. To impose a specific order for numbering, we must explicitly define a window specification. The mechanism involves leveraging the built-in functions `row_number()` and `Window()` from the `pyspark.sql` library. Understanding how these components interact is essential for generating reproducible and accurate row IDs.

Below, we will first introduce the precise syntax required for this operation, followed by a comprehensive practical example demonstrating the entire workflow, from DataFrame initialization to final output manipulation. We will also delve into the technical rationale behind using the `lit()` function in this context, which is key to forcing a global, non-partitioned ordering.

The Essential Syntax for Row Numbering

To successfully add a new column with globally sequential row numbers in PySpark, you must utilize a combination of functions designed for complex analytical operations. Specifically, this involves importing `row_number` and `lit` from `pyspark.sql.functions`, and `Window` from `pyspark.sql.window`. These tools allow us to define how the numbering should occur across the entire dataset, regardless of how the data is partitioned across the cluster.

The following syntax represents the standard and most robust method for generating sequential row IDs starting from 1 up to N (the total number of rows in the DataFrame). It defines a special window specification that covers the entire dataset, then applies the `row_number()` function over that specification. This ensures that every record receives a unique, sequential identifier.

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

#add column called 'id' that contains row numbers from 1 to n
w = Window().orderBy(lit('A'))
df = df.withColumn('id', row_number().over(w))
```

This implementation effectively adds a new column named **id** to your DataFrame. The values within this column will range sequentially from 1 up to the total count of rows (N). The successful execution of this code snippet depends entirely on the correct definition of the Window function, which we will explore in detail next.

Deep Dive into Window Functions and Ordering

The magic behind generating global row numbers lies in the use of the Window function. A Window specification defines a set of rows on which a function (like `row_number`) operates. For global numbering, the window must encompass all rows without any partitioning, yet it still requires a mandatory ordering component.

The line `w = Window().orderBy(lit('A'))` is crucial. Since we are interested in a global sequence and not a sequence partitioned by a specific column (like 'team' or 'conference'), we leave the `partitionBy()` clause empty. However, the `orderBy()` clause is mandatory for `row_number()` to work, as ranking functions inherently require a defined order to assign ranks or sequences.

The ingenious use of `lit('A')` solves the problem of arbitrary ordering. The `lit` function creates a literal column where every row holds the same constant value ('A' in this case). When Spark performs `orderBy()` on a column where all values are identical, the resultant ordering is determined by the underlying physical storage order of the data on the cluster. While this method does not guarantee the **exact** same order between two different Spark job runs (as Spark's internal partitioning can change), it is the standard technique for assigning a unique sequence across a non-partitioned set, satisfying the requirement of the `row_number` function.

Practical Demonstration: Setting up the DataFrame

To solidify our understanding, let us walk through a complete, hands-on example. We will begin by initializing a Spark session and constructing a sample DataFrame that represents typical structured data.

The example dataset contains information about various teams, their conference, and their scores (points). This DataFrame serves as the basis for demonstrating how the row numbering technique works in a real-world context. Creating a Spark session is the prerequisite for any PySpark operation, and defining the schema (column names) ensures clarity.

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

As shown in the output above, the initial DataFrame contains three columns. Our objective is to now augment this structure by introducing a fourth column that serves as a sequential index.

Implementing Row Numbering (Step-by-Step Execution)

With the DataFrame initialized, we can now apply the previously discussed logic utilizing Window functions. This process involves defining the window specification, applying the `row_number()` operation, and using `withColumn` to inject the result back into the DataFrame under the desired name, which we have chosen to be 'id'.

The code block below demonstrates the exact implementation. Note the mandatory imports required at the beginning of the script. After defining the window `w` using `Window().orderBy(lit('A'))`, the `withColumn` transformation triggers the calculation across the distributed dataset.

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

#add column called 'id' that contains row numbers from 1 to n
w = Window().orderBy(lit('A'))
df = df.withColumn('id', row_number().over(w))

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

Notice that the new **id** column contains row numbers ranging from 1 to 6. This successfully solves the problem of adding a global row index in PySpark using robust Window function techniques.

Refining the Output: Reordering Columns

If you'd like, you can reorder the columns so that the **id** column appears at the front, which is common practice when generating index columns. The `select()` method handles this rearrangement easily by specifying the desired order of columns.

```
#move 'id' column to front
df = df.select('id', 'team', 'conference', 'points')

#view updated DataFrame
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 sequential `id` column is now positioned at the beginning of the DataFrame, providing a clean and easily accessible index for the dataset.

Important Considerations: The Role of lit('A')

Note: The syntax **lit('A')** is simply used as an arbitrary value to satisfy the ordering requirement of the Window function. You can replace 'A' with anything you'd like (e.g., `lit(1)`) and the code will still work, provided the value remains constant across all rows. This forcing of a non-deterministic global order is essential when you need contiguous sequential numbering (1, 2, 3...) rather than just unique IDs.

Alternatives: monotonically_increasing_id vs. row_number

When seeking to add unique identifiers to a PySpark DataFrame, two main functions are often considered: `row_number()` (as detailed above) and `monotonically_increasing_id()`. While both generate unique IDs, they serve different use cases and have distinct performance implications driven by Spark's architecture.

The `monotonically_increasing_id()` function generates 64-bit integer IDs that are guaranteed to be unique and generally increasing across partitions, but not necessarily contiguous or sequential. This method is highly optimized because it avoids a global shuffle (a costly operation where data is redistributed across the cluster). It is ideal for simply generating a unique key when the exact sequential order is not required (i.e., you just need a primary key).

Conversely, `row_number()`, when used over a non-partitioned window, guarantees a strict sequence (1, 2, 3, ... N). Achieving this strict sequence requires Spark to collect and order all data globally (the global shuffle), which is computationally expensive for extremely large datasets. Therefore, if strict sequential numbering is essential, `row_number()` is the correct choice, but users must be mindful of the performance cost associated with the operation.

In summary, choosing between the two depends on your requirement:

If you need sequential IDs (1, 2, 3, ... N), use `row_number().over(Window().orderBy(lit('A')))`.
If you only need unique, non-sequential IDs with high performance, use

`monotonically_increasing_id()`.

Conclusion: Mastering PySpark Indexing

Adding sequential row numbers to a PySpark DataFrame is a common requirement that necessitates leveraging the powerful capabilities of Window functions. By coupling `row_number()` with a globally defined window ordered by a literal column (such as `lit('A')`), we can reliably generate a unique, sequential index suitable for various data engineering and analytical tasks.

While this technique guarantees contiguity (1, 2, 3, ...), always remember the trade-off inherent in distributed computing: demanding a global order introduces a computational burden (the global shuffle). Data scientists should always assess whether a strictly sequential index is necessary or if a more performant, unique but non-contiguous ID (like those provided by `monotonically_increasing_id`) would suffice for the specific application.

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