How to Add a Random Number Column to Your PySpark DataFrame

Authored by

stats writer

January 18, 2026

Introduction to Generating Random Data in PySpark

Integrating PySpark DataFrame manipulation with statistical tasks often requires the ability to introduce non-deterministic or **randomized data**. Generating random numbers is a fundamental requirement in scenarios ranging from simulation modeling and statistical sampling to creating dummy data for testing purposes. PySpark, through its rich library of built-in SQL functions, provides highly efficient and scalable mechanisms to achieve this across distributed datasets. This guide details the expert methods available for adding a new column populated with randomized values, covering both floating-point decimals and whole integers, ensuring full compatibility with large-scale data processing workflows inherent to **Apache Spark**.

The flexibility of PySpark lies in its integrated use of the `pyspark.sql.functions` module, which allows users to apply sophisticated mathematical operations directly onto columns without needing to rely on expensive user-defined functions (UDFs) that often hurt performance in distributed environments. By leveraging optimized functions like `rand` and `round`, we ensure that the generation process is executed efficiently across all nodes of the cluster. Understanding how to properly configure these functions, particularly how to manage the range and reproducibility of the generated values, is critical for data integrity and consistent analysis in distributed systems.

We will explore two primary techniques to populate a new column with random values. The first method focuses on generating **random decimal numbers** (floats), typically normalized between 0 and 1, which can then be scaled to any desired range. The second method builds upon the first, incorporating the `round` function to convert these continuous random values into discrete **random integers**. Both methods prioritize the use of the rand() function, which is the cornerstone for non-deterministic number generation within the PySpark environment.

Prerequisites: Setting up the PySpark Environment and Sample Data

Before proceeding with the column generation techniques, it is essential to establish a working PySpark session and define a sample PySpark DataFrame. The DataFrame serves as the target structure where the new randomized column will be appended. We initiate the `SparkSession` which acts as the entry point to programming Spark with the DataFrame API. This setup is standard practice for any PySpark operation and ensures that computational resources are correctly allocated across the cluster.

For demonstration purposes, we will create a simple DataFrame containing fictional team names and their corresponding scores. This structure provides a clear context for observing how the new random column integrates with existing structured data. The following code snippet demonstrates the necessary imports and the creation of our sample dataset, which we will refer to as `df` throughout the subsequent examples. This foundational step guarantees that all code examples

are immediately runnable and reproducible.

```
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|points|
+-------+------+
| Mavs| 18|
| Nets| 33|
| Lakers| 12|
| Kings| 15|
| Hawks| 19|
|Wizards| 24|
| Magic| 28|
| Jazz| 40|
|Thunder| 24|
| Spurs| 13|
+-------+------+
```

The resulting DataFrame, `df`, is now ready for modification. The process of adding a new column in PySpark is typically executed using the `withColumn` transformation. This method, common in SQL operations within Spark, creates a new column derived from an existing one or based on a calculated expression, preserving the immutability of the original DataFrame by returning a new one. In our case, the expression involves calling the `rand()` function, ensuring a distinct random value is generated for every row in the partition.

Method 1: Generating Random Decimal Numbers (Floats)

The most straightforward approach for introducing randomization is by generating floating-point decimal numbers. This method utilizes the core `rand()` function imported from `pyspark.sql.functions`. By default, the output of rand() function is uniformly distributed between 0 (inclusive) and 1 (exclusive). This normalized range provides a baseline which can be easily scaled to meet specific data requirements, such as generating random percentages or probabilities in modeling exercises.

To generate random numbers within a specific upper bound, such as 100, we simply multiply the output of the `rand()` function by the maximum desired value. For instance, multiplying by 100 will scale the output range from $[0, 1)$ to $[0, 100)$. The transformation is applied using `withColumn`, where we specify the name of the new column (e.g., 'rand') and define the mathematical expression that calculates its values. This methodology is efficient because the distribution of random values is calculated row-by-row in a distributed manner, adhering to Spark's principle of lazy execution.

The following snippet illustrates the essential components required for generating random decimal numbers. We import `rand` and then apply the column transformation. Note the inclusion of a `seed` parameter within the function call, which, as we will discuss in detail later, is crucial for ensuring the reproducibility of the random sequence, a necessary feature for debugging and analytical verification in production environments.

from pyspark.sql.functions import rand

#create new column named 'rand' that contains random floats between 0 and 100
df.withColumn('rand', rand(seed=23)*100).show()

Deep Dive into the rand() Function and Seeding

The `rand()` function in PySpark utilizes a powerful, distributed pseudo-random number generator (PRNG). Unlike true randomness, PRNGs rely on a deterministic algorithm to produce sequences of numbers that appear statistically random. This is fundamental in computational environments

where predictability is often prioritized over true entropy, especially in scenarios involving statistical simulations or machine learning where identical data splits are required across multiple runs for comparison.

A key component of any PRNG is the initial value, or the seed. When a specific seed value is supplied to `rand(seed=N)`, the function guarantees that the sequence of generated random numbers will be identical every single time the code is executed on the same data partitions. This feature of repeatability is immensely valuable in data engineering and data science workflows. If the seed is omitted, PySpark defaults to using a system-dependent value (often based on the current system time), resulting in a unique, non-reproducible sequence of numbers for each execution, which is suitable for tasks like live simulation but problematic for analytical reproducibility.

The mathematical principle behind scaling the output is straightforward: since `rand()` produces values R such that $0 le R < 1$, multiplying R by a constant M scales the resulting values to $0 le R cdot M < M$. In our examples, setting $M=100$ ensures that the generated numbers fall within the desired range of $[0, 100)$. This scaling factor effectively determines the maximum limit of the random sequence, while the minimum limit remains 0 unless an offset is added to shift the entire range, such as generating numbers between 10 and 50.

Example 1: Implementing Random Float Generation

Let's apply Method 1 to our sample DataFrame to observe the results. We aim to add a column named 'rand' containing decimal values between 0 and 100. By setting the seed to 23, we ensure that if this code were to be run again later, the exact same random sequence would be generated alongside the existing team and points data, preserving the integrity of any previous analysis based on this randomized column.

The `withColumn` transformation, followed by the `show()` action, executes the distributed generation process and displays the resulting DataFrame. Notice how the new column 'rand' is appended to the right side of the original data, and each entry is a unique floating-point value. This detailed output confirms the successful application of the scaling factor (multiplying by 100) and the random distribution generated by the rand() function.

from pyspark.sql.functions import rand

#create new column named 'rand' that contains random floats between 0 and 100
df.withColumn('rand', rand(seed=23)*100).show()

+-------+------+------------------+
| team|points| rand|

```
+-------+------+------------------+
| Mavs| 18| 93.88044512577216|
| Nets| 33|39.432553969527554|
| Lakers| 12|23.260361399084918|
| Kings| 15| 2.339183228862929|
| Hawks| 19| 82.53753350983487|
|Wizards| 24| 88.94415403143505|
| Magic| 28| 80.81524027081029|
| Jazz| 40| 59.56629641640896|
|Thunder| 24| 27.62195585886885|
| Spurs| 13| 70.43214981152886|
+-------+------+------------------+
```

Observation confirms that all values in the new `rand` column are decimal numbers that fall strictly within the range $[0, 100)$. This method is highly effective for tasks requiring precision, such as Monte Carlo simulations or assigning weights where fractional values are acceptable or necessary. Should a different range be required (e.g., from 50 to 150), the formula would be slightly adjusted by multiplying by the desired range size ($150 - 50 = 100$) and then adding the minimum offset (50) to the entire expression.

Method 2: Generating Random Integers

In many use cases, particularly when assigning discrete indices, categories, or simulating event counts, random integers are preferred over continuous floating-point numbers. Generating random integers in PySpark requires a small but significant modification to Method 1: the inclusion of the `round()` function. The `round()` function truncates the fractional component of a number based on standard mathematical rounding rules, converting the continuous float into a discrete value.

To ensure we generate random integers between 0 and 100, we first apply the same scaling technique using `rand() * 100`, which provides the desired range of floating-point numbers. We then wrap this entire expression within the `round()` function. The syntax for rounding in PySpark requires specifying the number of decimal places to retain; setting this parameter to 0 effectively rounds the result to the nearest whole number. This combined operation is executed efficiently across the cluster nodes.

This two-step process--generating a scaled float and then rounding it--is crucial for maintaining high performance within the distributed computing framework. We avoid traditional Python methods for floor or ceiling operations that might require UDFs, relying instead on highly optimized built-in SQL functions. The resulting values will be of type double but will represent whole numbers (e.g., 94.0 instead of 93.88...). For strict compatibility with integer fields, explicit casting may be

required after the rounding step.

from pyspark.sql.functions import rand, round

#create new column named 'rand' that contains random integers between 0 and 100
df.withColumn('rand', round(rand(seed=23)*100, 0)).show()

Example 2: Implementing Random Integer Generation

Using the combined power of `rand()` and `round()`, we execute the generation of random integers on our PySpark DataFrame. Just as in Example 1, we utilize the `withColumn` transformation and maintain the same seed value (23). This ensures that the base sequence of pseudo-random numbers is identical to the previous example, allowing us to directly compare the effect of the rounding operation and confirm deterministic results.

The output clearly demonstrates how the floating-point values from Example 1 have been converted to their nearest whole number. For instance, the original float 93.88... is rounded up to 94.0, and 39.43... is rounded down to 39.0. This systematic conversion ensures that the resulting dataset now contains discrete random values within the specified range of 0 to 100.

from pyspark.sql.functions import rand, round

#create new column named 'rand' that contains random integers between 0 and 100
df.withColumn('rand', round(rand(seed=23)*100, 0)).show()

```
+-------+------+----+
| team|points|rand|
+-------+------+----+
| Mavs| 18|94.0|
| Nets| 33|39.0|
| Lakers| 12|23.0|
| Kings| 15| 2.0|
| Hawks| 19|83.0|
|Wizards| 24|89.0|
| Magic| 28|81.0|
| Jazz| 40|60.0|
|Thunder| 24|28.0|
| Spurs| 13|70.0|
+-------+------+----+
```

It is important to acknowledge the data type of the new column. While the values appear as integers, PySpark's `round()` function, when used without explicit type casting, returns a `DoubleType` with a zero decimal place representation (e.g., 94.0). If a strict `IntegerType` is required for subsequent operations (like indexing or matching database schema requirements), an explicit cast should be applied immediately after the rounding operation using the `.cast("int")` method. This guarantees precise data type compliance while maintaining the desired range and distribution.

Conclusion and Further PySpark Operations

In summary, PySpark offers highly efficient, distributed methods for creating new columns populated with random numbers. Whether the requirement is for continuous decimal values or discrete integers, the combination of the `rand()` function, optional scaling, and the `round()` function provides a flexible and scalable solution. Mastery of the seed parameter is essential for ensuring that these randomized operations are fully reproducible, a non-negotiable requirement for professional data analysis and engineering workflows.

The two primary methods discussed--generating decimals using `rand() * Max` and generating integers using `round(rand() * Max, 0)`--are foundational techniques that leverage PySpark's optimized SQL functions. These methods exemplify the power of the DataFrame API to handle complex mathematical operations across massive datasets without the performance degradation typically associated with traditional row-by-row processing in conventional programming languages.

For data professionals looking to expand their capabilities, the next steps involve exploring related PySpark functionality, such as generating Gaussian (normal) distributed random numbers using `randn()`, or applying conditional logic during randomization using `when()` and `otherwise()` clauses to introduce randomness only when specific conditions are met. These advanced techniques build directly upon the principles demonstrated here, allowing for even more sophisticated data simulations and manipulations within the **Apache Spark ecosystem**.