How to Randomly Sample Rows from a PySpark DataFrame

Authored by

stats writer

January 3, 2026

In the realm of big data processing, efficiently generating a random sample of data is often necessary for tasks like testing, exploratory data analysis, or model training on subsets. In PySpark, the powerful **`sample()` function** provides a streamlined mechanism to achieve this goal, allowing users to randomly select rows from a distributed DataFrame. This function is defined by its use of a `fraction` parameter, which specifies the approximate proportion of rows to be returned, and an optional `seed` parameter crucial for ensuring the reproducibility of results.

Understanding the nuances of the **`sample()` function** is vital for reliable data processing. It also offers the `withReplacement` parameter; when this boolean is set to true, it permits the same row to be selected and returned multiple times in the resulting sample, a process known as sampling with replacement. Regardless of the parameters chosen, the function always returns a new DataFrame containing the randomly sampled subset of rows.

Deep Dive into the PySpark sample() Function

To select a random subset of data, you utilize the **`sample` function** directly on a PySpark DataFrame object. This method is fundamental for data preparation pipelines where only a representative fraction of the massive dataset is required for specific computations or debugging.

Syntax and Essential Parameters Explained

The **`sample()` function** signature in PySpark is straightforward yet flexible, allowing control over the sampling mechanism through three primary arguments:

```
sample(withReplacement=None, fraction=None, seed=None)
```

These parameters dictate how the random selection process is executed:

`withReplacement`: This boolean argument determines the sampling methodology. Setting it to `True` allows a single row to be selected multiple times (sampling with replacement). The default setting is `False` (sampling without replacement).
`fraction`: This required floating-point number specifies the expected proportion of rows from the original DataFrame to be included in the resulting sample. It must be a value between 0.0 and 1.0, inclusive.
`seed`: An optional integer value that defines the starting point for the random number generator. Utilizing a consistent seed parameter is critical for generating identical samples across subsequent runs, ensuring that tests and analyses are reproducible.

Ensuring Reproducibility and Handling Sample Size

A key feature for rigorous data science workflows is the ability to reproduce results. If you are sharing code or debugging models, you must ensure that your random sample remains the same every time the code is executed. This reproducibility is achieved by setting the **seed** parameter to a specific, constant integer value. Omitting the seed or setting it dynamically (e.g., based on time) will result in a different sample set each time the script runs.

It is important to understand the probabilistic nature of the sample() function. The value provided for the **fraction** argument is an approximation. Due to the distributed nature of PySpark and how partitions are handled, the resulting sample size is not mathematically guaranteed to be that exact fraction of the total rows of the original DataFrame, although it will be very close, especially for larger datasets.

Practical Application: Setting up the PySpark Environment

The following comprehensive example demonstrates how to utilize the sample() function in practice. We begin by initializing a SparkSession and creating a small demonstration DataFrame containing hypothetical information about various basketball players.

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

# Define raw data containing team names and points scored.
data = ,
,
,
,
,
,
,
,
,
]

Define column schema.
columns =

# Create the PySpark DataFrame.
df = spark.createDataFrame(data, columns)

# Display the initial DataFrame structure and content.
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 contains 10 rows. For the purpose of this demonstration, we will attempt to select a random sample comprising approximately **30%** (a fraction of 0.3) of the total dataset.

Example 1: Sampling Without Replacement (Default Behavior)

The most common use case involves selecting a subset where each row from the source data appears at most once. This is known as sampling without replacement, and it is the default behavior when the `withReplacement` parameter is either set to `False` or simply omitted.

We utilize the following syntax to create a sample DataFrame, ensuring that no rows are duplicated in the output:

```
# Select a random sample equivalent to 30% of the DataFrame size (without replacement).
df_sample = df.sample(withReplacement=False, fraction=0.3)

# View the resulting random sample.
df_sample.show()
```

```
+-----+------+
| team|points|
+-----+------+
| Mavs| 18|
| Nets| 33|
|Kings| 15|
+-----+------+
```

In this specific run, the resulting DataFrame randomly selected 3 out of the 10 rows from the original DataFrame, which aligns perfectly with the requested 30% fraction. Since we explicitly specified **`withReplacement=False`**, this guarantees that each row from the original DataFrame occurs uniquely within the random sample.

Example 2: Enabling Sampling With Replacement

There are scenarios, particularly in statistical methods like bootstrapping, where it is necessary for a single observation to be included multiple times in the sample. To enable this functionality, we set the **`withReplacement`** parameter to `True` when calling the sample() function.

This modification changes the underlying probability distribution of the selection process, allowing for repetition of rows based on the defined `fraction`.

 # Select a random sample (with replacement allowed) of 30% of rows in DataFrame.
df_sample = df.sample(withReplacement=True, fraction=0.3)

# View the resulting random sample.
df_sample.show()

```
+-----+------+
| team|points|
+-----+------+
|Magic| 28|
|Spurs| 13|
|Magic| 28|
+-----+------+
```

Note that the team name **Magic** occurred twice in the random sample since we used sampling with replacement in this example. This confirms that rows can be selected more than once when `withReplacement=True`.

Summary and Further Resources

The PySpark **`sample()` function** is an indispensable tool for data engineers and data scientists working with large-scale datasets, providing a fast and efficient way to create representative subsets for testing and analysis. Mastering the use of the `fraction`, `withReplacement`, and particularly the seed parameter ensures that your sampling methodologies are both statistically sound and reproducible in a distributed computing environment like PySpark.

For a comprehensive understanding of all available parameters, constraints, and optimization

details related to random selection in Spark, you can consult the complete official documentation for the PySpark `sample` function.

Related PySpark Tutorials

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