# Learning Random Row Sampling Techniques in PySpark DataFrames for Data Analysis

Authored by

**Mohammed looti**

November 11, 2025

The rapid growth of data necessitates sophisticated tools for efficient analysis. When dealing with large-scale datasets, such as those typically handled by PySpark, processing the entire population can be computationally prohibitive and time-consuming. Consequently, a core skill for any data professional is the ability to extract a statistically robust and representative subset of the data. This technique, known as random sampling, is crucial for accelerating processes like rapid prototyping, initial model training, and quality assurance checks, all without incurring the enormous overhead associated with querying terabytes of information. Within the distributed environment of Apache Spark, selecting an unbiased random sample of rows from a DataFrame is elegantly managed through the powerful and highly configurable `sample` function. Mastering the intricacies of this method is essential for big data engineers and analysts striving to harness the full capabilities of the Spark ecosystem.

## Deep Dive into the PySpark Sample Function Syntax

The primary mechanism for generating a subset of data in PySpark is the invocation of the `sample` method, which is applied directly to the target DataFrame object. This function is engineered to provide substantial flexibility in the random selection process, enabling users to precisely define the statistical nature of the sample. Specifically, users can dictate whether the sampling occurs with or without replacement, specify the relative size of the desired subset, and establish a fixed initialization point for the underlying randomness. This exhaustive control ensures that the resulting sample is statistically sound and perfectly tailored for the intended downstream analysis, whether that involves complex machine learning workflows or simple, efficient exploratory data analysis (EDA).

To perform rigorous data operations, explicitly defining the parameters of the `sample` function is highly recommended, even though all arguments are technically optional. These parameters govern the entire execution flow, influencing both the statistical properties and the reproducibility of the output. The function adheres to the following fundamental signature, which serves as the blueprint for controlling the randomization process:

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

These three arguments allow developers to fine-tune the output to meet specific operational or statistical requirements. For instance, careful adjustment of these values can guarantee that every selected row is unique, or conversely, ensure that the exact same random sample is generated whenever the code is executed, which is paramount for debugging and auditability. The combination of these settings fundamentally defines the type of statistical sampling applied to the distributed DataFrame structure.

## Controlling Variability: Replacement, Fraction, and Seed

The three core arguments within the **`sample`** function dictate both the statistical methodology and the deterministic characteristics of the sampling operation. The first critical argument is **`withReplacement`**, a boolean flag that defaults to `False`. This setting determines whether an observation, after being selected for the sample, is immediately returned to the original population pool before the next observation is drawn. Setting this flag to `True` permits the possibility of duplicate rows appearing in the final result, a feature that is essential for advanced statistical techniques suching as bootstrapping. Conversely, maintaining the default setting of `False` guarantees that every row chosen is unique, perfectly mimicking a simple random selection process without any duplication.

The second essential parameter is **`fraction`**, which is defined as a floating-point value between 0.0 and 1.0. This value specifies the approximate proportion of the original DataFrame rows that should be included in the final sampled subset. For example, if a developer specifies `0.25`, the function attempts to sample about 25% of the total rows. It is vital to understand the nuances of this parameter within the Spark distributed architecture: due to the underlying probabilistic nature of the sampling mechanism, the resulting sample size is not mathematically guaranteed to be exactly the requested fraction of the total rows. Instead, it represents the expected mean value. While the actual count may exhibit minor variance, particularly in extremely small datasets, for the vast majority of big data use cases, the final fraction will be very closely aligned with the requested value, offering a highly reliable approximation of the desired sample size.

Finally, the **`seed`** parameter, an optional integer, specifies the starting value for the internal random number generator utilized during the selection process. The implementation of a seed is arguably the most critical step for ensuring reproducibility across multiple executions. In any formal or production data pipeline, the ability to regenerate an identical sample set is non-negotiable for validation and auditing purposes. By assigning a specific integer value (such as `42` or `101`) to the seed, you ensure that running the sampling logic repeatedly will consistently yield the exact same set of rows. If the **`seed`** is omitted, PySpark defaults to a non-deterministic seed, usually derived from system time, which means a completely different sample will be generated upon every execution of the code.

**`withReplacement`**: A boolean flag determining whether sampling should be performed with replacement (`True`) or without (`False`). The default is `False`, ensuring all sampled rows are unique and independently selected.
**`fraction`**: A float between 0 and 1 indicating the desired proportional size of the final sample relative to the original DataFrame.
**`seed`**: An optional integer used to initialize the random number generator, guaranteeing the same sequence of random numbers and thus ensuring identical sample selection across multiple runs for

consistency.

## Practical Implementation: Setting Up the Baseline DataFrame

To effectively demonstrate the functionality and precise behavior of the `sample` method, we must first establish a foundational DataFrame within the PySpark environment. This setup phase requires initializing a Spark session and loading or creating sample data. For this illustration, we will utilize a small dataset containing information about various basketball teams and their recent scoring metrics. This initial step meticulously simulates a typical real-world scenario where raw data is ingested and structured within the distributed computing framework of Spark, providing a clear context for our subsequent random sampling operations.

The following code block outlines the standard and necessary procedure for constructing a small, manageable DataFrame from in-memory data. We clearly define the structured data rows and assign descriptive column names before using the `createDataFrame` method. This method is responsible for materializing the structured data into Spark's distributed memory architecture. This controlled setup yields a clear baseline dataset of 10 distinct rows, which will serve as the population from which we draw our subsequent random selections, allowing for easy verification of the sampling results.

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

With the base DataFrame named `df` successfully initialized, we are now ready to implement and test various sampling methodologies. For the subsequent demonstrations, let us assume our primary analytical objective is to quickly examine a subset of this data--specifically, approximately **30%** of the total rows--in order to gain a rapid preliminary understanding of the scoring distribution before committing significant resources to a full-scale, cluster-wide analysis. This selection of 30% serves as an industry-standard balance between maximizing speed and ensuring the resulting sample remains highly representative of the overall population structure.

## Case Study 1: Implementing Sampling Without Replacement (Unique Selections)

Sampling without replacement represents the most frequently used approach for generating a random sample when the overriding goal is to guarantee that the resultant subset comprises entirely unique observations drawn from the source population. Because the function defaults to **`withReplacement=False`**, explicitly setting this parameter is often performed not out of necessity, but to enhance the clarity and overall robustness of production-level code, thereby ensuring that statistical independence is rigorously maintained among the selected rows. This methodology is preferred in scenarios where observation uniqueness is mandatory, such as generating training and testing datasets for machine learning models.

In our practical example, our goal is to extract approximately 30% of the dataset, which translates to selecting three unique rows from the original ten teams. By setting **`fraction=0.3`** and retaining

the default `withReplacement=False`, we explicitly command PySpark to execute a standard, non-duplicating random selection. This approach ensures that each team appears at most once in the resultant subset, maintaining the integrity required for most traditional statistical analyses.

The following demonstration illustrates the execution of this non-replacement sampling strategy, followed by a display of the resulting data subset. Note the explicit setting of the parameters for maximum clarity. The output confirms the expected behavior for a small, controlled dataset, yielding exactly three distinct rows:

```
#select random sample of 30% of rows in DataFrame
df_sample = df.sample(withReplacement=False, fraction=0.3)

#view random sample
df_sample.show()

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

The resulting DataFrame, labeled `df_sample`, contains three distinct rows (Mavs, Nets, and Kings, based on this specific random execution) randomly chosen from the original population. The use of `withReplacement=False` rigorously guarantees that these observations are unique selections from the pool of 10 teams. This robust methodology ensures that each data point contributes singularly to any subsequent statistical analysis performed on the sample.

## Case Study 2: Implementing Sampling With Replacement (Bootstrapping)

In sharp contrast to the default non-replacement method, sampling with replacement is activated by setting the parameter `withReplacement=True`. This statistically powerful technique introduces the possibility--indeed, the likelihood--of selecting and including the identical row multiple times within a single sample generation. While the inclusion of duplicates may initially appear counterintuitive for standard exploratory data analysis, this method is fundamentally vital for advanced statistical procedures, most notably bootstrapping, where the objective is to accurately estimate the variability, confidence intervals, or distribution of a statistic by repeatedly drawing randomized samples from the original observed data.

When `withReplacement=True` is specified, the probability assigned to selecting any particular row

remains constant throughout the entire selection sequence. This mechanism ensures that a row selected early in the process is immediately returned to the pool, making it available for subsequent selection draws. For handling massive datasets, this method plays a crucial role in simulating the statistical process of drawing observations from an effectively infinite population, which forms the core assumption underlying many established statistical theorems and models.

To clearly demonstrate this behavior, we execute the sampling logic once more, maintaining the requested `fraction=0.3` but crucially changing the replacement status. It is important to remember that when replacement is enabled, the fraction often refers more closely to the expected number of total draws relative to the population size, potentially leading to slight variations in the resulting count compared to non-replacement sampling. The defining feature, however, is the composition of the resulting sample, as illustrated in the code below:

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

#view random sample
df_sample.show()

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

Upon reviewing the output, the outcome of using replacement is immediately evident: the team name **Magic** appears twice within the resulting three-row sample. This duplication is the direct, intended consequence of setting `withReplacement=True`. For specialized statistical modeling and hypothesis testing, the capacity to generate such samples is foundational for accurately establishing robust confidence intervals and performing precise variance estimation. Data professionals must therefore make a deliberate and informed choice regarding the replacement strategy based strictly on the specific statistical objectives of their project.

## Ensuring Reproducibility with the Random Seed

In distributed computing and data engineering, the concept of reproducibility is not a mere preference--it is a mandatory requirement for reliability, testing, and compliance. The `seed` parameter within the `sample` function serves as the sole mechanism for achieving deterministic

sampling results in PySpark. This integer value initializes the pseudo-random number generator, ensuring that the sequence of random numbers used for row selection remains identical across every execution, regardless of the cluster state or execution time.

When preparing data for model training or validating complex analytical results, it is essential that the sample used for analysis can be recreated exactly by others or by automated testing frameworks. If the **`seed`** is omitted, PySpark utilizes a time-dependent, non-deterministic value, leading to a new, unique sample every single time the code runs. This non-reproducible behavior is highly detrimental in production environments. By contrast, specifying a fixed integer seed, such as `42`, guarantees consistency.

Consider a scenario where we sample 50% of our data without replacement, guaranteeing that the result is always the same subset of teams:

**#sample 50% of rows with a fixed seed**
**df_reproducible_sample = df.sample(withReplacement=False, fraction=0.5, seed=42)**

#view reproducible sample
df_reproducible_sample.show()

+-------+------+
| team|points|
+-------+------+
| Nets| 33|
| Lakers| 12|
| Hawks| 19|
|Wizards| 24|
| Spurs| 13|
+-------+------+

If this code is executed thousands of times on different machines or at different moments in time, the output will consistently be the same five rows: Nets, Lakers, Hawks, Wizards, and Spurs. This deterministic behavior, enforced by the seed, moves the sampling operation from a random process to a controlled, repeatable operation, which is a fundamental requirement for reliable data pipelines and rigorous scientific computation using distributed frameworks.

## Limitations and Advanced Sampling Techniques

While the basic **`sample`** function provides excellent, efficient probabilistic sampling, users must remain cognizant of certain operational limitations and be prepared to employ advanced techniques when requirements dictate more complex selection criteria. The primary limitation, as

discussed, is the non-guarantee of an exact sample size. The `fraction` parameter provides an expected mean approximation. In situations where an exact count is non-negotiable--for instance, requiring precisely 10,000 rows for regulatory reporting--a slightly more resource-intensive, two-step process is required. This typically involves adding a temporary column populated with random numbers, ordering the DataFrame by this random column, and subsequently using the `limit()` action to truncate the result to the desired precise number of rows.

Another crucial consideration involves the assumption of simple random sampling. The standard `sample` function performs uniform selection across the entire DataFrame. However, many real-world datasets exhibit stratification, meaning they contain subpopulations that must be represented proportionally in the sample. If the analytical goal requires preserving the distribution of a categorical variable (e.g., ensuring the sample contains 30% of rows from each geographical region), simple random selection is insufficient. In these cases, PySpark offers a dedicated function: `sampleBy`. This method facilitates stratified sampling, ensuring that the selected subset accurately reflects the known proportions of various groups within the original population, thereby providing a superior basis for inference when population heterogeneity exists.

Adhering to best practices--especially the consistent use of the `seed` parameter--is essential for transforming a quick exploratory step into a foundational, trustworthy component of a production data pipeline. The decision between using `sample`, `sampleBy`, or a custom limit operation should always be driven by the balance between statistical rigor, computational cost, and the specific reproducibility requirements of the project.

## Additional Resources for PySpark Mastery

For developers and data scientists seeking to explore the full technical specifications, constraints, and performance implications of the `sample` function, the official documentation for PySpark remains the most comprehensive and authoritative reference.

The following tutorials explain how to perform other common data manipulation and analysis tasks within the PySpark environment: