How to Randomly Sample Rows in PySpark DataFrames

Authored by

stats writer

February 4, 2026

Welcome to this detailed guide on implementing statistically robust sampling techniques within the Apache PySpark environment. When dealing with massive datasets--a common scenario in big data analytics--it is often impractical or computationally expensive to process the entire dataset for initial analysis, model prototyping, or quality checks. Selecting a manageable, representative subset of data is therefore essential.

The standard mechanism for achieving this data reduction in Spark is the use of the sample method, which is a core function of the Spark DataFrame API. This functionality provides powerful control over how data is randomly extracted, allowing users to specify parameters critical for statistical validity, such as the fraction of data to be included and whether observations should be sampled with or without replacement. By mastering this method, data professionals can significantly accelerate their workflow while maintaining the integrity required for accurate data science applications.

This article will provide a comprehensive, step-by-step walkthrough of the sample transformation, detailing its parameters, illustrating practical examples, and discussing the nuances necessary for generating statistically sound, reproducible samples.

Select Random Sample of Rows in PySpark

The Necessity of Random Sampling in Big Data

In the realm of big data, working directly with datasets that contain billions of records can be prohibitively slow and resource-intensive. Random sampling serves as a fundamental technique to mitigate these computational challenges. By selecting a representative subset of the data, analysts can quickly perform exploratory data analysis (EDA), validate assumptions, and rapidly iterate on machine learning model development before committing to full-scale processing.

The integrity of the resulting sample is paramount. A truly random sample ensures that the statistical properties of the original population are maintained within the smaller subset. If the sample is biased or non-representative, any conclusions drawn from it will be flawed when extrapolated back to the entire dataset. Therefore, the implementation of controlled sampling techniques, such as those offered by the PySpark DataFrame API, is a critical skill for any big data practitioner.

PySpark provides built-in mechanisms that handle the complexity of distributed sampling, ensuring that the randomness is applied correctly across partitions, which is a non-trivial requirement given Spark's architecture. The primary tool we utilize for this purpose is the dedicated sample function, which we will now explore in detail.

Understanding the PySpark `sample` Method

The sample method is the go-to utility for extracting a random subset of rows from a PySpark DataFrame. It is a powerful transformation that generates a new DataFrame containing the sampled records. Unlike actions that trigger computation immediately, the sampling process is part of Spark's lazy execution model, meaning the sampling logic is executed only when an action (like `show()` or `count()`) is called on the resulting sampled DataFrame.

The flexibility of the sample method stems from its ability to handle both sampling with and without replacement, catering to different statistical requirements. Sampling without replacement is generally preferred for basic data exploration as it guarantees uniqueness of records within the sample, preventing redundancy and ensuring the sample is diverse. Sampling with replacement, conversely, is often used in techniques like bootstrapping, where statistical resampling requires the possibility of selecting the same record multiple times.

Effective use of this function requires understanding its key parameters, which control the statistical behavior and reproducibility of the operation. By carefully tuning these parameters, users can ensure their extracted samples are statistically sound and meet the specific needs of their analytic tasks, whether it be simple debugging or complex model training.

Syntax and Core Parameters of `sample()`

The core structure of the sample function is straightforward yet highly functional, accepting three crucial parameters that define the sampling process. The syntax is attached directly to the DataFrame object itself, making it intuitive to use within a processing pipeline.

The function uses the following standard signature in PySpark:

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

The meaning and usage of each parameter are defined as follows:

`withReplacement`: This Boolean parameter determines whether the sampling process is conducted with or without replacement. If set to `True`, a row that has already been selected for the sample can be selected again, potentially leading to duplicate entries in the output DataFrame. The default behavior, `False`, ensures that each row from the original DataFrame appears at most once in the sample. This choice has significant statistical implications regarding independence between selections.

`fraction`: This floating-point value specifies the expected proportion of rows to include in the sample. It must be a value between 0.0 and 1.0. For instance, setting `fraction=0.3` implies that approximately 30% of the original rows are targeted for inclusion in the sample.

seed: An integer value used to initialize the random number generator. Providing a specific seed is essential for ensuring that the sampling operation is deterministic, meaning that running the code multiple times will yield the exact same random sample.

Understanding the interplay of these parameters is key to generating a representative and reproducible subset of data tailored for specific analytical needs.

The Role of the `seed` Parameter for Reproducibility

In data science, reproducibility is a core principle. When dealing with random processes, such as sampling, ensuring that others (or your future self) can replicate your exact results is vital for validation and auditing. This is precisely where the seed parameter plays its critical role.

By providing an integer value to the seed argument, you initialize the pseudo-random number generator that PySpark uses internally to decide which rows to select. If the same DataFrame and the same seed are used, the resulting sampled DataFrame will be identical every single time, regardless of the cluster configuration or execution timing. This ensures consistency across different runs and environments.

It is strongly recommended to always set the seed parameter when generating samples intended for model training or critical analysis. If the seed is omitted (or set to `None`), Spark uses a time-based or internal system-derived value, making the sample truly random but inherently non-reproducible. For development and debugging, non-reproducible randomness can introduce unpredictable variations, making bug tracking difficult. Therefore, adopting a consistent, fixed seed is best practice.

Example Setup: Creating a PySpark DataFrame

To demonstrate the practical application of the sample function, let us first establish a sample DataFrame. This DataFrame will contain simple information about basketball teams and their recent game scores. This setup mirrors a common scenario where initial data loading is followed by a sampling phase to create a manageable subset for analysis.

We begin by initializing the Spark Session, defining our data structure (a list of tuples), and specifying the column headers. This process ensures we have a concrete dataset upon which we can perform our sampling operations and observe the results directly. The example is small to ensure clarity, but the sampling principle scales perfectly to terabyte-sized data volumes handled by PySpark.

The code below sets up the environment and creates the base DataFrame:

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

As evident from the output, the original DataFrame, `df`, contains 10 records. Our goal in the following sections will be to extract a random subset using the sample method, targeting a fraction

of 30% of the total rows.

Implementing Sampling Without Replacement (Default Behavior)

The most common scenario in data preparation is to select a subset where every observation is unique. This is achieved by setting the `withReplacement` parameter to `False`, which is also the default behavior if the parameter is omitted. This mode adheres to standard simple random sampling principles.

We aim to select approximately 30% of the rows (3 rows out of 10) from our established DataFrame. We will explicitly set `withReplacement=False` to ensure no duplicates, although this step is redundant given the default setting, it serves to increase clarity and intent in the code.

The following code snippet demonstrates the execution of the sampling process:

#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, `df_sample`, successfully selected 3 unique rows from the original 10, achieving the targeted fraction of 0.3 (30%). Because we specified **`withReplacement=False`**, we are guaranteed that each row in the original DataFrame appears at most once in the output sample. This type of sampling is ideal for initial data profiling and creating training/testing splits for machine learning models, where unique data points are required to prevent data leakage or biased evaluation.

Implementing Sampling With Replacement (Exploring Duplicates)

In contrast to sampling without replacement, setting `withReplacement` to `True` allows the possibility of a single row being selected multiple times. This technique, often referred to as Bernoulli sampling, is foundational to statistical procedures like bootstrapping or Monte Carlo

simulations where repeated observations are necessary to estimate population characteristics or variability.

When sampling with replacement, the size of the resulting sample is often less predictable, particularly for smaller fractions, as the selection process is truly independent for each potential inclusion. We will now run the same fraction (0.3) but enable replacement:

#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|
+-----+------+

In this execution using sampling with replacement, we observe that the team name **Magic** (with 28 points) occurred twice in the resulting sample of three rows. This demonstrates the core characteristic of this sampling method: the ability for a single record to be selected, returned to the population pool, and then potentially selected again. It is vital for analysts to choose between sampling with or without replacement based on the specific statistical demands of their project. If independence of observations is a requirement, sampling without replacement should be strictly enforced.

Important Considerations Regarding Sample Fraction Accuracy

A critical point often overlooked by new users of the PySpark sample function relates to the accuracy of the `fraction` argument. The value provided for `fraction` (e.g., 0.3) represents the **expected** proportion of rows to be included, not a guaranteed exact percentage.

PySpark performs row-level sampling using probability. Each row is independently evaluated against the specified fraction. For example, if `fraction=0.3`, each row has a 30% probability of being included in the sample. Due to this probabilistic nature, especially when working with smaller DataFrames, the actual resulting sample size may deviate noticeably from the exact calculated fraction.

For large DataFrames, however, the Law of Large Numbers dictates that the observed sample fraction will converge closely to the specified `fraction`. If an exact sample size is absolutely required (e.g., exactly 10,000 rows), the sample method should be combined with an action like `limit()`, or alternative methods like `take()` (if the dataset is small enough to fit into the driver memory) or using specialized sampling techniques like stratified sampling (often requiring custom implementation using window functions or grouping and then applying `sample()` per group) may be more appropriate.

Conclusion and Further Resources

The sample method within PySpark is an indispensable tool for data professionals navigating the complexities of big data processing. It provides a highly efficient and statistically sound way to reduce data dimensionality for faster prototyping, testing, and exploratory analysis.

By effectively managing the parameters--specifically **`withReplacement`** to control for uniqueness and **`seed`** to ensure reproducibility--analysts can generate high-quality, representative samples. Remember that the **`fraction`** parameter defines a probability expectation rather than a strict count guarantee. For advanced scenarios requiring stratified samples or precise counts, additional logic layered on top of the base sampling function might be necessary.

For comprehensive details on all accepted arguments and potential use cases, the official documentation for the PySpark **sample** function is the definitive resource.

Related PySpark Tutorials

For those looking to deepen their expertise in PySpark, exploring other common tasks is crucial. Related tutorials often focus on data manipulation, aggregation, and machine learning pipeline construction:

How to Filter Rows in a PySpark DataFrame based on complex conditions.
Techniques for performing SQL-like Joins across multiple PySpark DataFrames.
Methods for grouping data and calculating aggregate statistics using window functions.
Implementing custom User Defined Functions (UDFs) for specialized column transformations.