How to Round PySpark Column Values to 2 Decimal Places

Authored by

stats writer

January 2, 2026

Introduction: The Necessity of Precision in Data Processing

When working with large-scale datasets using PySpark, data precision is often a critical factor. Raw data, especially metrics derived from calculations or sensors, frequently contain numerous decimal places that are statistically irrelevant or unnecessary for reporting purposes. These highly precise numbers can clutter analysis, make visualizations difficult to interpret, and potentially consume excessive storage space or processing power during subsequent operations. Therefore, mastering techniques for data normalization and simplification, such as rounding column values, is fundamental to effective data engineering, ensuring both performance and clarity in analytical results.

The process of rounding numeric values ensures consistency and readability, particularly when dealing with financial metrics, averages, or statistical scores where convention dictates a specific level of precision, such as two decimal places. In the context of the Apache Spark ecosystem, PySpark provides robust, highly optimized functions within its SQL module to handle these mathematical transformations across distributed DataFrame structures efficiently. Understanding how to apply the round() function correctly is vital for any professional utilizing this powerful distributed computing framework for data cleansing and preparation.

This detailed guide focuses specifically on how to efficiently round numeric columns within a DataFrame to exactly two decimal places using the built-in round() function. We will explore the required imports, the primary syntax involving the withColumn() transformation, and walk through a practical, comprehensive example demonstrating the entire workflow from data creation to final rounded output visualization. This skill is indispensable for preparing data for downstream applications like machine learning models or BI dashboards.

Understanding the PySpark round() Function and Parameters

The core utility for achieving numerical rounding in PySpark is the round() function, which must be imported from the `pyspark.sql.functions` module. This function adheres to standard mathematical rounding principles, taking a numeric column expression as input and returning the value rounded to the nearest specified precision. Because it is part of the SQL functions library, it is optimized for parallel execution, ensuring that rounding operations are executed swiftly across the cluster nodes, which is crucial when dealing with petabyte-scale datasets typical in Spark environments.

The round() function requires two essential arguments when utilized for column transformations. The first parameter specifies the target column whose values are to be rounded. This column reference can be passed directly using the DataFrame attribute notation (e.g., `df.column_name`) or as a string literal of the column name. The second parameter is an integer that dictates the

precision--the exact number of decimal places to which the rounding should occur. To meet the common requirement of financial or standardized reporting, where data often needs to be accurate to pennies or percentages, this integer argument should be set to `2`.

It is important for users to understand that the PySpark variant of the round() function is designed for vectorized operations. Unlike the standard Python built-in `round()`, which handles single scalar values, the PySpark function applies the rounding logic simultaneously to all elements within the specified column across the distributed partitions of the DataFrame. Leveraging this dedicated SQL function is fundamental to capitalizing on Spark's parallel processing architecture and avoiding slow, inefficient User Defined Functions (UDFs) for basic mathematical transformations.

Core Syntax: Creating a New Rounded Column using withColumn()

The industry standard and most robust method for applying transformations like rounding, especially when the goal is to create a modified version of the data while preserving the original column, is by employing the withColumn() DataFrame method. The nature of withColumn() ensures immutability; it returns a completely new DataFrame incorporating the transformation, leaving the initial source data intact. This feature is vital for complex ETL (Extract, Transform, Load) pipelines where traceability and debugging are necessary.

To successfully execute this operation, the developer must first ensure the necessary function is imported: `from pyspark.sql.functions import round`. Following the import, the syntax is highly declarative. You call withColumn() on the existing DataFrame, providing two main arguments: the desired name for the new column (e.g., `'points2'`) and the column expression that performs the transformation (e.g., `round(df.points, 2)`). This pattern is clean, readable, and highly optimized by the Spark Catalyst Optimizer.

The resulting syntax below creates a new column, allowing for direct comparison between the original, high-precision values and their rounded counterparts. If you wished to overwrite the original column instead, you would simply use the original column's name (e.g., `'points'`) as the first parameter in the withColumn() call.

You can use the following syntax to round the values in a column of a PySpark DataFrame to 2 decimal places:

from pyspark.sql.functions import round

#create new column that rounds values in points column to 2 decimal places
df_new = df.withColumn('points2', round(df.points, 2))

This particular example creates a new column named **points2** that rounds each of the values in

the **points** column of the DataFrame to 2 decimal places. The column reference `df.points` indicates the source column, and the integer `2` dictates the precision level.

The following example provides a complete, working demonstration of how to apply this syntax in a real-world PySpark session, from data initialization through transformation.

Setting Up the PySpark Environment and Sample Data

Before executing the rounding logic, it is necessary to establish the operational PySpark environment and define a sample dataset. This step involves importing the `SparkSession` class and initializing an instance, which serves as the entry point to all PySpark functionality. We utilize the common `SparkSession.builder.getOrCreate()` pattern to manage the cluster connection.

For demonstration purposes, we define a small, representative dataset simulating basketball team performance statistics. This data is structured as a list of rows, where one column, 'points', intentionally contains high-precision floating-point values--more than the two decimal places we aim for--to effectively showcase the rounding transformation. This simulates the typical scenario where raw data from a source system requires normalization.

Finally, the raw data structure is converted into a distributed PySpark DataFrame using `spark.createDataFrame()`. Explicitly defining the column names ensures clarity and prepares the DataFrame, `df`, for the subsequent transformation step, allowing us to inspect the initial state of the data before rounding is applied.

Initializing the DataFrame for Transformation

Suppose we have the following PySpark DataFrame that contains information about points scored by various basketball players, where the 'points' column contains high-precision values:

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	18.3494
Nets	33.5541
Lakers	12.6711
Kings	15.6588
Hawks	19.3215
Wizards	24.0399
Magic	28.6843
Jazz	40.0001
Thunder	24.2365
Spurs	13.9446
+-------+-------+

The output above confirms that the original **points** column contains precision extending to four decimal places. Our objective is to apply the round() function to this column, preserving only the first two decimal places while adhering to standard mathematical rounding rules. This transformation is necessary to prepare the data for final presentation or further analytical models that require fixed precision input.

Executing the Rounding Logic and Viewing Results

The final step in the primary workflow involves executing the rounding transformation using the withColumn() method. We map the result of `round(df.points, 2)` to a new column named `points2`. This explicit naming convention is highly beneficial as it allows for immediate verification against the source data, ensuring the transformation pipeline is transparent and auditable.

As soon as the transformation is defined, the Spark Catalyst Optimizer optimizes the execution

plan. When the action `df_new.show()` is called, the rounding logic is executed across all distributed partitions, and the resulting DataFrame, `df_new`, is displayed. This output clearly demonstrates the efficiency of PySpark in handling numerical accuracy requirements on a large scale.

The result set below confirms that the new **points2** column successfully holds the rounded values, demonstrating the practical application of the two-parameter round() function within a standard PySpark data processing environment.

Suppose we would like to round each of the values in the **points** column to 2 decimal places. We can use the following syntax to do so:

from pyspark.sql.functions import round

#create new column that rounds values in points column to 2 decimal places
df_new = df.withColumn('points2', round(df.points, 2))

#view new DataFrame
df_new.show()

+-------+-------+-------+
| team| points|points2|
+-------+-------+-------+
Mavs	18.3494	18.35
Nets	33.5541	33.55
Lakers	12.6711	12.67
Kings	15.6588	15.66
Hawks	19.3215	19.32
Wizards	24.0399	24.04
Magic	28.6843	28.68
Jazz	40.0001	40.0
Thunder	24.2365	24.24
Spurs	13.9446	13.94
+-------+-------+-------+

Verification of Rounding Accuracy

A careful inspection of the new column, **points2**, reveals that the PySpark round() function has correctly applied the half-up rounding rule. This verification step is critical, especially when dealing with financial data where rounding errors can lead to significant discrepancies. We observe how the third decimal places dictates whether the second decimal place is retained or incremented.

Notice that the new column named **points2** contains each of the values from the points column rounded to 2 decimal places.

For example:

The value **18.3494** was rounded to **18.35** because the digit 9 (at the third decimal place) triggered an upward round.
The value **33.5541** was rounded to **33.55** because the digit 4 (at the third decimal place) resulted in a round-down (or truncation to two places).
The value **12.6711** was rounded to **12.67**, again showing the round-down effect when the third decimal digit is less than 5.
For values like **40.0001**, the result is displayed as **40.0** in the `show()` output. This is a common display convention in Spark, but the underlying data type holds the value with the requested precision (i.e., it is semantically 40.00).

And so on. This consistent behavior across the distributed dataset confirms the reliability of the PySpark implementation for numerical precision control.

Alternative Method: Rounding and Display using select()

While integrating rounded data permanently using withColumn() is standard practice, there are situations, particularly during exploratory data analysis (EDA), where a developer only needs to view or quickly output the rounded values without altering the existing DataFrame schema. In these cases, utilizing the `select()` transformation method offers a faster, temporary solution.

The `select()` transformation constructs a new temporary DataFrame based solely on the specified expressions. By wrapping the round() function within `select()` and optionally using `alias()` to provide a meaningful name for the derived column, the desired result can be obtained immediately. This method is particularly useful for debugging or quick sanity checks on numerical stability.

To demonstrate this, consider the original data. If we only wanted to see the rounded 'points' column, we could execute the following simplified code snippet, which is the original introductory example provided for PySpark rounding: `df.select(round("x", 2).alias("x")).show()`. This pattern is concise and highly efficient for direct output needs.

Summary of Rounding Techniques and Further Resources

Effectively controlling numerical precision via rounding is a cornerstone of data quality assurance in PySpark. We have established that the `pyspark.sql.functions.round(column, scale)` function is the definitive tool for this job, offering optimized, distributed computation capabilities.

The primary workflow involves using withColumn() to persist the two-decimal-place rounding into a new column, maintaining data integrity and traceability.

Mastery of this fundamental transformation opens the door to more sophisticated data preparation tasks. Depending on specific domain requirements (e.g., financial or scientific), you might need to explore related PySpark functions such as `ceil()` for rounding up or `floor()` for rounding down. These are also available within `pyspark.sql.functions` and adhere to the same vectorized execution principles, ensuring scalability.

For developers seeking comprehensive details on all available mathematical and numerical transformations, consulting the official PySpark documentation is essential. The documentation provides deep insights into data type handling, function behavior in edge cases (like handling negative numbers or ties), and performance considerations.

Note: You can find the complete documentation for the PySpark **round** function here.

The following tutorials explain how to perform other common tasks in PySpark, building upon the foundational knowledge of column manipulation demonstrated here: