Count Null Values in PySpark (With Examples)

Authored by

stats writer

November 16, 2025

Handling missing data is a critical step in any data processing pipeline. In the context of large-scale distributed computing using PySpark, identifying and quantifying these missing entries--commonly represented as Null Values--is essential before performing transformations or analysis. A DataFrame in PySpark often contains thousands or millions of rows, making manual inspection impossible. Therefore, utilizing efficient PySpark functions to aggregate null counts is necessary for maintaining data quality.

This comprehensive guide explores two primary, highly efficient methods for counting nulls within a PySpark DataFrame. The choice between these methods depends largely on whether you need a quick count for a specific column or a detailed summary across the entire dataset. Both techniques leverage the optimized distributed capabilities of the Spark engine, ensuring performance even with massive datasets.

We will first introduce the methodology for isolating and counting nulls within a single, targeted column. Following that, we will demonstrate a more advanced, programmatic approach using conditional aggregation to generate a summarized report of null counts for every column simultaneously. Understanding these techniques is fundamental for any data engineer or data scientist working with PySpark.

Method 1: Counting Null Values in a Single Column

When the objective is to quickly assess the completeness of a single feature or column, this method provides the fastest and most readable solution. It relies on chaining two fundamental DataFrame operations: filtering and aggregation. Specifically, we use the `.where()` transformation combined with the `.isNull()` condition, followed by the `.count()` action.

The `.where()` function (or `.filter()`) is instrumental in isolating specific rows that meet a defined criteria. In this case, the criteria is defined by the column attribute `.isNull()`, which returns a Boolean value indicating whether the data point is null. After the transformation filters the DataFrame down to only those rows containing a null value in the specified column, the subsequent `.count()` action simply returns the number of rows remaining in that filtered subset, effectively giving us the total null count for that column.

#count number of null values in 'points' column
df.where(df.points.isNull()).count()

Method 2: Counting Null Values Across All Columns Efficiently

For large-scale data quality checks, assessing one column at a time is inefficient. PySpark offers a powerful, concise method using conditional aggregation combined with list comprehension to

generate a comprehensive null count report across all columns simultaneously. This technique is highly efficient because it processes the data column-wise in a single pass.

This method requires importing several functions from the pyspark.sql.functions module, specifically `when`, `count`, and `col`. The core logic hinges on the `when()` function, which acts like an SQL CASE statement. We check if the column value is null using `col(c).isNull()`. If it is null, we count it; otherwise, we ignore it. By wrapping this conditional logic within the `count()` function and applying it iteratively across all column names using Python's list comprehension, we construct a new summary row where each column represents its total null count.

from pyspark.sql.functions import when, count, col

#count number of null values in each column of DataFrame
df.select().show()

Setting Up the Sample PySpark DataFrame

To illustrate these two methods in a practical context, we will first define and create a sample DataFrame. This dataset contains information related to basketball players, including their team designation, assists count, and points scored. Importantly, we have intentionally introduced null values in both the `assists` and `points` columns to demonstrate how to accurately detect them using PySpark commands.

We begin by initiating a SparkSession, which is the entry point for using Spark functionality. Once the session is active, we define our raw data as a list of tuples and specify the column schema. Finally, the `createDataFrame` method converts this raw Python data structure into a distributed, schema-aware PySpark DataFrame named `df`, allowing us to proceed with the null counting examples.

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|assists|points|
+----+-------+------+
A	null	11
A	4	8
A	2	22
A	10	null
B	8	null
B	11	14
B	14	13
B	6	7
C	2	8
C	2	5
+----+-------+------+

Example 1: Counting Nulls in the 'Points' Column

We now apply the first methodology to determine the null count specifically within the **points** column. This is a common requirement when focusing on the quality of a key metric. By chaining the `.where()` transformation with the `.isNull()` condition, we instruct Spark to filter the dataset, retaining only the rows where the value in the `points` column is null. The final `.count()` action triggers the execution and returns the resultant total as an integer.

This approach is computationally inexpensive when analyzing a single column, as the query execution plan (the physical plan) is straightforward: filter and aggregate. Observe the output of the following command, which confirms the number of missing point entries in our sample dataset:

#count number of null values in 'points' column
df.where(df.points.isNull()).count()

2

The resulting value, **2**, indicates that there are exactly two rows in our DataFrame where the `points` score is recorded as a null value. This number is vital for subsequent data cleaning steps, such as imputation or row deletion.

Investigating Rows Containing Null Values

While knowing the count of nulls is important, data analysis often requires viewing the actual records that contain the missing data to understand the context of the issue. Instead of performing the final `.count()` action, we can use the `.show()` action to display the filtered DataFrame subset. This is extremely useful for debugging and qualitative assessment of missingness.

By replacing the aggregation action with a display action, we maintain the filtering logic provided by `df.where(df.points.isNull())`. The output table clearly isolates the two records identified previously, allowing us to see which specific players (teams A and B in this case) are associated with the missing point data.

#display rows with null values in 'points' column
df.where(df.points.isNull()).show()

```
+----+-------+------+
|team|assists|points|
+----+-------+------+
| A| 10| null|
| B| 8| null|
+----+-------+------+
```

As demonstrated, the resulting DataFrame accurately shows only the two rows where the `points` column contains a null value. This technique is invaluable when performing targeted troubleshooting on large datasets.

Example 2: Comprehensive Null Counting Across All Columns

The second method addresses the need for a holistic view of data quality across the entire DataFrame. This involves constructing a dynamic SQL aggregation query that iterates through every column in the schema. This advanced approach is often preferred in production environments for generating automated data quality reports.

We leverage Python's list comprehension capabilities alongside powerful functions from

pyspark.sql.functions. The expression `count(when(col(c).isNull(), c)).alias(c)` is the operational heart of this method. For every column `c`, it performs a conditional check. If the column value is null, the conditional expression returns the column identifier `c`, which is then counted by the outer `count()` function. If it is not null, the count is effectively skipped. The `.select()` statement then gathers these calculated counts into a new, single-row DataFrame.

from pyspark.sql.functions import when, count, col

#count number of null values in each column of DataFrame
df.select().show()

+----+-------+------+
|team|assists|points|
+----+-------+------+
| 0| 1| 2|
+----+-------+------+

Interpreting the Comprehensive Null Count Results

The output of the conditional aggregation provides a clear, quantitative summary of data completeness across the entire dataset. This single summary row allows for immediate identification of data quality issues without needing to run separate queries for each column. We can break down the results from the aggregated summary table as follows:

There are **0** null values in the **team** column. This indicates that all records successfully recorded the team assignment.
There is **1** null value in the **assists** column. This suggests one player record is missing assist data.
There are **2** null values in the **points** column. This aligns perfectly with the result obtained from Method 1, verifying the consistency of both approaches.

This comprehensive reporting ability makes Method 2 the preferred choice when generating overall statistical metrics for data profiling and validation in a PySpark environment.

Summary of PySpark Null Handling Strategies

We have successfully demonstrated two powerful and distinct strategies for counting nulls in a PySpark DataFrame. Method 1 (using `.where()` and `.count()`) is excellent for targeted, quick assessment of a single column's completeness. It is intuitive and easy to read, making it suitable for ad-hoc analysis.

Method 2, employing conditional aggregation via `when()` and list comprehension, offers a robust,

scalable solution for generating a complete null report across all fields in a single query execution. This method adheres to the best practices of distributed computing by minimizing the number of passes over the data, which is crucial for performance optimization when dealing with petabytes of information.

Mastering these techniques ensures that data engineers can reliably profile their datasets and initiate appropriate data imputation or cleaning protocols, maintaining high data integrity throughout the ETL process.