Use “Is Not Null” in PySpark (With Examples)

Authored by

stats writer

November 16, 2025

The Importance of Null Handling in Data Processing

Working with large datasets often involves managing missing or undefined entries, commonly represented as Null values. In the context of big data frameworks like Apache Spark, accurately identifying and filtering these missing entries is a fundamental step in data cleansing and preparation. Unhandled nulls can lead to skewed aggregations, errors in mathematical operations, or unexpected behavior in machine learning models. Therefore, mastering techniques to explicitly exclude rows where a specific column--or any column--contains a null entry is essential for maintaining data integrity.

The PySpark DataFrame API provides robust and efficient mechanisms to address this challenge. Specifically, we focus on filtering operations designed to isolate and retain only those records that possess meaningful, non-null data points. The two primary methods employed for this purpose are the column-specific filter using **`isNotNull()`** and the DataFrame-wide cleaning operation using **`dropna()`**.

This guide will delve into these two powerful methods, offering detailed explanations and practical examples using PySpark. By the end of this tutorial, you will be proficient in selecting the appropriate null-filtering strategy based on your specific analytical requirements, ensuring your data pipelines are robust and reliable.

PySpark DataFrames: A Foundation for Data Manipulation

Before diving into the filtering syntax, it is crucial to understand the foundation: the PySpark DataFrame. A DataFrame is a distributed collection of data organized into named columns, conceptually equivalent to a table in a relational database or a data frame in R/Python (Pandas). This structure allows for highly optimized operations across a cluster, making it the primary data structure for large-scale data processing in PySpark.

When working with DataFrames, nulls often arise from various sources, such as missing entries in the original source file, failed joins, or explicit assignment during transformation steps. PySpark represents these missing values using the standard SQL concept of `NULL`, which is distinct from zero, an empty string, or any specific datatype value. The goal of filtering is to create a new DataFrame subset containing only the valid records.

Setting Up the PySpark Environment and Sample Data

To demonstrate the methods effectively, we first need to initialize a SparkSession and construct a sample DataFrame containing deliberate null entries. This setup allows us to precisely observe the outcome of both the **`isNotNull()`** and **`dropna()`** filtering techniques. Notice how the nulls are distributed across the `conference` and `points` columns in the initial dataset structure below.

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

# Define the raw data, including deliberate None entries
data = ,
,
,
,
]

# Define the column schema
columns =

# Create the dataframe using data and column names
df = spark.createDataFrame(data, columns)

# View the initial DataFrame structure, showing where nulls reside
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
| A| East| 11| 4|
| A| null| 8| 9|
| A| East| 10| 3|
| B| West| null| 12|
| B| West| null| 4|
| C| East| 5| 2|
+----+----------+------+-------+
```

The output clearly shows two types of nulls in our dataset: the second row has a null in the `conference` column, and rows four and five have nulls in the `points` column. Our subsequent filtering exercises will demonstrate how to handle these missing values based on whether we target a single column or the entire row.

Core Method 1: Filtering for Non-Null Values in a Specific Column (The `isNotNull()` Function)

Often, data analysis requires focusing on the quality of a single critical measurement while ignoring nulls elsewhere, or perhaps data imputation has already taken care of other columns. For such

scenarios, PySpark provides the column function **`isNotNull()`**. This method is part of the Column object API and is invoked directly on the column you wish to inspect within the DataFrame filter operation.

The **`isNotNull()`** function returns a boolean expression (True/False) for every row, indicating whether the value in that specific column is non-null. This expression is then passed to the DataFrame's **`filter()`** (or **`where()`**) method, allowing PySpark to efficiently select only the rows where the condition evaluates to True. This approach offers precise control over which column's data integrity dictates row retention.

The generalized syntax involves selecting the DataFrame (`df`), calling the filter method, and passing the column name followed by `.isNotNull()`. It is a highly idiomatic and performant way to perform targeted data quality checks within PySpark.

Implementation of `isNotNull()`: Focusing on the 'points' Column

Let's apply this method to our sample DataFrame. Suppose we are interested only in rows where the `points` measurement is recorded and valid, regardless of whether the `conference` field is null or not. We use the following syntax to filter the DataFrame to only show rows where the value in the **`points`** column is not null:

 # Filter for rows where value is not null in the 'points' column
df.filter(df.points.isNotNull()).show()

```
+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
| A| East| 11| 4|
| A| null| 8| 9|
| A| East| 10| 3|
| C| East| 5| 2|
+----+----------+------+-------+
```

The resulting DataFrame is noticeably smaller than the original, retaining four rows. Crucially, rows four and five (where `points` was null) have been removed. However, the second row, where the `conference` column was null but `points` was 8, remains. This confirms the targeted nature of the **`isNotNull()`** operation; it successfully isolates valid entries based only on the specified column's content.

This specific filtering technique is highly useful when different columns have varying levels of importance or data quality requirements. For instance, if `points` is the dependent variable in a

statistical analysis, dropping rows based on a missing `conference` might represent unnecessary data loss. The **`df.column_name.isNotNull()`** syntax is the cleanest way to execute this precise selection.

Core Method 2: Handling Nulls Across the Entire DataFrame (The `dropna()` Function)

In contrast to targeted column filtering, sometimes the requirement is strict row integrity: every column must contain a valid value for the row to be considered usable. This is particularly common before feeding data into systems that cannot tolerate any missing features. For this comprehensive cleanup, PySpark provides the **`dropna()`** function, which is an alias for the **`na.drop()`** method used on DataFrames.

The default behavior of **`dropna()`** is to drop any row that contains at least one null value across any of its columns (equivalent to setting `how='any'`). It acts as a strict filter, ensuring that every retained record is complete according to the defined schema. This method is exceptionally efficient for quick, holistic data cleaning, especially when dealing with wide datasets where any missing feature could compromise model integrity.

While simple, it is important to understand that **`dropna()`** can be customized using optional parameters, such as `how` ('any' or 'all') and `subset` (list of columns to check). Since our initial goal is to filter for rows where a value is not null in *any* column, the simple call to `df.dropna()` achieves this perfectly.

Implementation of `dropna()`: Excluding All Rows with Nulls

We can use the following concise syntax to filter the DataFrame to only show rows where there are absolutely no null values in any column. Based on our sample data, this operation will remove the second row (due to the null `conference`) and rows four and five (due to the null `points`).

 # Filter for rows where value is not null in any column (using default how='any')
df.dropna().show()

```
+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|  A|      East|    11|      4|
|  A|      East|    10|      3|
|  C|      East|     5|      2|
+----+----------+------+-------+
```

The resulting DataFrame contains only three complete rows. Comparing this output to the result of **`isNotNull()`** in the previous example, we see the critical difference: the row was excluded here because it contained a null in the `conference` column, even though `points` was valid. This demonstrates the strict, row-level completeness check enforced by the default **`dropna()`** operation.

For use cases demanding high data integrity across all features or when preparing a final dataset for export, **`dropna()`** is the simplest, most readable, and most direct method for eliminating incomplete records entirely.

Comparison and Best Practices for Null Filtering

Choosing between **`isNotNull()`** and **`dropna()`** depends entirely on the scope of your data quality requirement. Understanding the nuances of each function is key to efficient and accurate data cleansing in PySpark:

Use `df.column.isNotNull()` when: You need to ensure data validity for one or a specific set of critical columns, allowing other non-essential columns to potentially contain null values. This method retains maximum data, focusing the quality check narrowly. It is preferred when analyzing specific features where other features might be auxiliary or already handled.

Use `df.dropna()` (default) when: You require complete records, meaning every column must be non-null. This is ideal for preparing data for downstream models that strictly require dense feature vectors, or when running aggregations where missing data in any field could corrupt the result.

Advanced Filtering: Remember that **`isNotNull()`** is highly versatile and can be combined using logical operators (`&`, `|`, `~`) within the **`filter()`** method. For example: `df.filter(df.points.isNotNull() & df.conference.isNotNull())` achieves the same result as `df.dropna(subset=)`.

By consistently applying these robust null-filtering techniques, you ensure that your analytical results derived from PySpark DataFrame operations are based on valid, defined data points, leading to more reliable insights.