How to Fill Missing Values in PySpark Using Values From Another Column

Authored by

stats writer

February 10, 2026

RECOMMENDED CITATION

stats writer (2026). *How to Fill Missing Values in PySpark Using Values From Another Column*. PSYCHOLOGICAL SCALES. Retrieved from https://scales.arabpsychology.com/?p=129996

ARABPSYCHOLOGY.COM

Understanding the Necessity of Handling Missing Data in PySpark

In the expansive realm of **Big Data**, the presence of missing or incomplete information is an inevitable challenge that data engineers and scientists must confront. When working with **PySpark**, the **Python** **API** for **Apache Spark**, managing these gaps effectively is paramount for maintaining the integrity of **Data Analysis**. Missing values, often represented as **null** or NaN, can arise from various sources, including sensor failures, user omissions, or errors during the data ingestion process. If left unaddressed, these gaps can lead to skewed results, biased **Machine Learning** models, and inaccurate statistical summaries.

The **fillna()** function is a well-known utility within the **DataFrame** **API** designed to mitigate these issues by replacing missing entries with specific constant values. However, real-world scenarios often require a more dynamic approach where a missing value in one column should be supplemented by a value found in a secondary, related column. This technique is particularly vital when dealing with primary and fallback data sources. By leveraging the power of **distributed computing**, **PySpark** allows users to perform these replacements across massive datasets with high efficiency and minimal manual intervention, ensuring that the resulting dataset is both comprehensive and reliable.

This article provides an in-depth exploration of how to use the **coalesce()** function--often the programmatic successor to a simple **fillna()** when logic involves multiple columns--to fill missing values from one column with values from another. We will examine the underlying **SQL** logic that powers this transformation and provide a comprehensive guide on implementing this in your production pipelines. By the end of this guide, you will understand how to enhance the accuracy of your **PySpark DataFrames** by intelligently filling data gaps using existing contextual information from your dataset.

The Role of Data Integrity in Modern Analytics

Data integrity serves as the backbone of any successful **Data Science** initiative. When datasets contain significant gaps, the reliability of the entire analytical workflow is called into question. In **PySpark**, a **DataFrame** is conceptually a distributed collection of data organized into named columns, and ensuring that each column contains valid, actionable information is a primary task for **Data Engineering** teams. Filling missing values is not merely about aesthetic completeness; it is about ensuring that downstream processes, such as **aggregations** or **feature engineering**, do not fail or produce misleading outputs due to the presence of **null** values.

While a simple **fillna()** can replace **null** entries with a static value like zero or a mean average, this approach often lacks the nuance required for high-fidelity data. For instance, if a dataset contains a "current_price" column and an "estimated_price" column, it is far more accurate to fill missing

ARABPSYCHOLOGY.COM

current prices with the estimated values rather than a generic constant. This preservation of context is what separates basic data cleaning from sophisticated data curation. By utilizing the **coalesce()** function within **PySpark**, developers can define a hierarchy of columns to consult until a non-null value is found, thereby creating a robust fallback mechanism.

Moreover, the performance benefits of using native **PySpark** functions cannot be overstated. Because **Apache Spark** operates on a **lazy evaluation** model, transformations like column-to-column filling are optimized by the **Catalyst Optimizer**. This ensures that even when processing terabytes of information, the operation is executed in a highly performant manner across a cluster of machines. Understanding these mechanics allows practitioners to write code that is not only functional but also scalable and optimized for the unique constraints of **Big Data** environments.

Implementing Column-to-Column Filling: The Core Strategy

The primary method for replacing **null** values in one column with values from another in **PySpark** involves the **coalesce()** function, which is imported from the **pyspark.sql.functions** module. The **coalesce()** function is an implementation of the standard **SQL COALESCE** expression, which returns the first non-null value among its arguments. This makes it the ideal tool for our specific use case: we pass the primary column and the secondary "source" column as arguments, and **PySpark** evaluates them row by row, selecting the first available value.

To apply this logic to a **DataFrame**, we typically use the **withColumn()** method. This method allows us to either create a new column or overwrite an existing one by applying a transformation. In the context of filling missing values, we overwrite the target column with the result of the **coalesce()** operation. This approach is cleaner and more readable than complex **when-otherwise** statements, providing a declarative way to express data imputation logic. The syntax is straightforward and integrates seamlessly into existing **PySpark** workflows.

Below is the specific syntax used to achieve this replacement. By importing the **coalesce** function and applying it within the **withColumn** framework, we can effectively manage **null** values without losing the structural integrity of our data. This method is highly versatile and can even be extended to more than two columns if multiple fallback sources are available in the dataset.

from pyspark.sql.functions import coalesce

df.withColumn('points', coalesce('points', 'points_estimate')).show()

This particular example demonstrates how to replace **null** entries in the **points** column with the corresponding values found in the **points_estimate** column. By utilizing this syntax, the resulting **DataFrame** will have a completed **points** column where every missing entry has been "patched" by the estimate. This ensures that subsequent calculations, such as calculating the total points

ARABPSYCHOLOGY.COM

across a league, remain accurate and representative of all available data points.

Detailed Walkthrough: A Practical PySpark Example

To better understand how this functionality operates in a real-world scenario, let us examine a practical example involving basketball player statistics. In this scenario, we have a **DataFrame** that tracks the performance of various teams, including their actual points scored and an estimated point value. It is common for certain games to have missing actual scores due to reporting delays, while estimates remain available. Our goal is to consolidate this information into a single, reliable **points** column.

First, we must initialize a **SparkSession**, which serves as the entry point for all **PySpark** functionality. We then define our raw data and schema, creating a **DataFrame** that explicitly includes **null** values to simulate the real-world data issues we described earlier. Observe the structure of the data below, noting how the teams "Lakers", "Hawks", and "Wizards" currently have **null** values in their primary scoring column.

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|points_estimate|
+-------+------+---------------+
| Mavs| 18| 18|

ARABPSYCHOLOGY.COM

Nets	33	33
Lakers	null	25
Kings	15	15
Hawks	null	29
Wizards	null	14
Magic	28	28
+-------+------+---------------+

In the **DataFrame** displayed above, the **points** column contains **null** values that would potentially break numerical analysis. However, the **points_estimate** column contains viable data that we can use to fill these gaps. By identifying this relationship, we can proceed to apply our transformation logic to create a more complete and useful dataset for our **Data Analysis** tasks.

Executing the Transformation and Verifying Results

With our **DataFrame** established and the gaps identified, we can now execute the **coalesce()** logic. This step effectively instructs **PySpark** to look at the **points** column; if it finds a value, it keeps it; if it finds a **null**, it immediately looks at the **points_estimate** column for that same row. This operation is performed in parallel across the cluster, making it extremely fast even for millions of rows. The use of **withColumn** ensures that the original schema is maintained while the data content is updated.

The code below illustrates the final execution of this logic. By displaying the **DataFrame** after the transformation, we can verify that the **null** values have been successfully replaced by the estimates. This verification step is a crucial part of the **Data Engineering** lifecycle, ensuring that the logic applied matches the expected business requirements.

from pyspark.sql.functions import coalesce

#replace null values in 'points' column with values from 'points_estimate' column
df.withColumn('points', coalesce('points', 'points_estimate')).show()

+-------+------+---------------+
| team|points|points_estimate|
+-------+------+---------------+
Mavs	18	18
Nets	33	33
Lakers	25	25
Kings	15	15
Hawks	29	29

ARABPSYCHOLOGY.COM

|Wizards| 14| 14|
| Magic| 28| 28|
+-------+------+---------------+

As we can observe in the output, the teams that previously had **null** values in the **points** column--Lakers, Hawks, and Wizards--now display the values 25, 29, and 14, respectively. These values were pulled directly from the **points_estimate** column. This demonstration highlights the simplicity and power of using **coalesce()** for dynamic data imputation, providing a clean alternative to more verbose conditional logic.

Advanced Considerations: When to Use coalesce() vs. fillna()

While this guide focuses on using **coalesce()** for column-to-column replacement, it is important to distinguish when to use this method versus the standard **fillna()** function. The **fillna()** function is most appropriate when you want to fill **null** values with a fixed constant (like 0 or "Unknown") or when you want to apply a dictionary of different constants to different columns. It is a specialized tool for static replacement. However, **fillna()** cannot natively reference another column's values in its replacement logic, which is where **coalesce()** becomes indispensable.

In more complex scenarios, you might encounter situations where you need to check three or more columns for a valid value. The **coalesce()** function handles this effortlessly by accepting an arbitrary number of arguments. For example, **coalesce('col_a', 'col_b', 'col_c')** will check Column A, then Column B, and finally Column C, returning the first non-null entry it finds. This hierarchical fallback logic is a common pattern in **Master Data Management** where multiple data sources are merged.

Another alternative is the **when().otherwise()** construct, which provides the most flexibility. While **coalesce()** is perfect for simple **null** checks, **when()** allows for more complex conditions, such as filling a value only if a **null** exists AND another condition is met (e.g., only for certain teams). However, for the specific task of filling **nulls** from another column, **coalesce()** is generally preferred due to its conciseness and optimized performance within the **Spark** execution engine.

It is also worth noting that **coalesce()** expects all arguments to be of the same or compatible data types. If you attempt to coalesce a string column with an integer column, **PySpark** may throw an error or perform an implicit type cast that could lead to unexpected results. Always ensure your data types are aligned before performing these operations to maintain strict control over your **data schema**.

ARABPSYCHOLOGY.COM

Best Practices for Data Imputation in PySpark

To maximize the effectiveness of your data cleaning routines in **PySpark**, it is recommended to follow a set of best practices. First, always perform a count of **null** values before and after your transformations. This allows you to quantify the impact of your changes and ensure that no unexpected **nulls** remain. Using **DataFrame** statistics or simple filters can provide quick insights into the health of your data throughout the **ETL** (Extract, Transform, Load) process.

Secondly, consider the source of your fallback data. Using another column to fill missing values assumes that the fallback column is itself reliable. If both columns contain **nulls**, the **coalesce()** function will still return a **null**. In such cases, you might chain a **fillna()** at the end of the operation to provide a final default constant, ensuring that the target column is truly "null-free" before it reaches the analysis or **Machine Learning** stage.

Finally, document your data imputation logic clearly. When other **Data Scientists** or analysts use your processed **DataFrames**, they need to know whether a value was an original data point or a filled value. Sometimes, it is beneficial to create a boolean flag column (e.g., "is_imputed") that tracks which rows were modified. This transparency is vital for downstream **auditing** and ensures that the conclusions drawn from the data are interpreted with the correct context.

By mastering these techniques, you can turn **PySpark** into a robust engine for data quality. The ability to dynamically fill gaps using **coalesce()** is a fundamental skill that elevates your data processing from basic manipulation to professional-grade **Data Engineering**. For more details, you can consult the official **coalesce() documentation** to explore further nuances of this powerful function.

Additional Resources for PySpark Mastery

Expanding your knowledge of **PySpark** involves exploring a wide variety of functions and methods designed to handle complex data structures. Beyond filling missing values, there are numerous other transformations that can streamline your **Data Analysis** workflows. Understanding the full breadth of the **pyspark.sql.functions** module is key to writing efficient and readable code.

The following tutorials and documentation pages explain how to perform other common and advanced tasks in **PySpark**:

Official PySpark fillna() Documentation: Learn more about static null replacement.
Using when() and otherwise(): Master conditional logic for complex data transformations.
Managing Columns with withColumn(): A deep dive into modifying DataFrame schemas.
Handling Nulls with dropna(): When filling is not an option, learn how to safely remove incomplete rows.

ARABPSYCHOLOGY.COM

By integrating these tools into your skill set, you will be well-equipped to handle any data challenge that comes your way in the **Apache Spark** ecosystem. Continuous learning and adherence to best practices in **Big Data** processing will ensure that your analytical outputs are always of the highest quality.

ARABPSYCHOLOGY.COM