Learning PySpark: Filling Missing Values with Data from Another Column

Authored by

Mohammed looti

November 10, 2025

Mastering Data Integrity: Column-Based Null Handling in PySpark

In the realm of large-scale data processing, effectively managing missing data is perhaps the most critical prerequisite for ensuring data quality and model reliability. When dealing with massive, distributed datasets managed by frameworks like PySpark, simple methods for replacing **null values** often fall short. Data pipelines frequently encounter scenarios where a primary measurement column might fail to record a value, but a corresponding, reliable secondary or estimated value exists in an auxiliary column within the same row. This challenge requires a sophisticated, column-specific imputation strategy that moves beyond static scalar replacement.

The standard `df.fillna()` transformation in PySpark is highly effective for substituting nulls with fixed scalar values--such as zero, a designated string like 'Unknown', or aggregate statistics like the mean or median of the column. However, this method lacks the dynamic capability needed when the replacement value must be sourced directly from another column on a row-by-row basis. Attempting to use static imputation methods in a dynamic scenario can lead to significant data loss or the introduction of statistical bias, compromising the integrity of downstream analytical processes and machine learning models.

To tackle this dynamic imputation requirement efficiently in a distributed environment, we must leverage specialized functions designed for column evaluation. The solution lies in employing the robust `coalesce()` function, a powerful tool provided within pyspark.sql.functions. This function provides an elegant, highly optimized native solution for conditional data imputation across columns within a **PySpark** DataFrame, ensuring that we seamlessly adopt a fallback value whenever the primary value is absent.

The Limitations of fillna() and the Need for Dynamic Imputation

While the `fillna()` operation is ubiquitous in data cleaning, its inherent limitation lies in its inability to reference other columns dynamically. Imagine a scenario involving sensor data where the primary sensor reading (`temp_primary`) frequently fails, but a secondary, lower-fidelity sensor reading (`temp_backup`) is available. If we were to use `fillna(0)`, we would falsely introduce zeros into the dataset, which might be unrealistic or misleading, especially if the backup sensor holds a valid measurement.

The requirement, therefore, is not a static substitution but a conditional replacement: "If column A is **null**, use the value from column B; otherwise, keep the value from column A." Implementing this logic efficiently across millions or billions of rows in a distributed cluster demands highly performant functions that operate at the Spark SQL engine level, avoiding the overhead associated with less optimized methods like complex User Defined Functions (UDFs) or expensive data shuffling.

The core challenge is maintaining the integrity of the data lineage. We must preserve any non-null,

valid entry in the primary column while ensuring that every missing entry is filled using the next available source column in a defined order of preference. This hierarchical approach to data quality is foundational in complex data engineering pipelines, particularly those supporting real-time decision-making systems or high-stakes analytical platforms where data completeness is non-negotiable.

Leveraging the Power of coalesce() for Dynamic Replacement

To effectively execute column-based null replacement in a **PySpark** DataFrame, we combine the structural modification capability of the withColumn() transformation with the intrinsic null-checking logic of the coalesce() function. The `withColumn()` method allows us to overwrite an existing column or create a new one based on a specified column expression. By overwriting the target column itself, we achieve the desired in-place imputation.

The genius of the coalesce() function lies in its sequential evaluation. It accepts multiple column inputs (or expressions) and systematically checks them, returning the value of the very first argument encountered that is not **null**. If, and only if, all provided arguments are null, the function itself returns null. This behavior makes it the perfect tool for defining a fallback hierarchy: by listing the primary column first, followed by the secondary column, we explicitly instruct **PySpark** to prioritize the original, measured data and only fall back to the estimate when absolutely necessary.

This standard implementation pattern is not only concise and highly readable but also exceptionally performant. Because `coalesce()` is a native Spark SQL function, the operation is optimized internally for distributed processing across the cluster. This avoids the bottlenecks associated with moving data or executing complex Python logic row by row, ensuring that the imputation process scales effortlessly with growing data volumes. The conceptual simplicity mirrors a prioritized case statement, yet it executes with the efficiency required for massive datasets.

The following syntax demonstrates the standard and most efficient way to use coalesce() within withColumn() to replace **null values** in a target column with corresponding values from an auxiliary column in a **PySpark** DataFrame:

from pyspark.sql.functions import coalesce

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

This operation dictates that if the `points` column contains a value, that value is maintained. If `points` is null, the value from `points_estimate` is immediately substituted. This robust methodology maximizes data retention and ensures that the dataset remains as complete and accurate as possible before moving to subsequent analytical stages.

Practical Demonstration: Setting Up the PySpark Environment

To illustrate the practical application of this column-based imputation technique, we first need to establish a controlled environment within **PySpark**. This setup involves initializing a SparkSession, which acts as the unified entry point for all PySpark functionality, and creating a sample dataset that explicitly contains the data quality issue we aim to solve. Our sample DataFrame will represent basketball statistics, featuring a primary `points` column with intentional **null values** and a secondary `points_estimate` column that serves as the designated fallback source.

The construction of this sample data is crucial for verification. We define several rows where the primary measurement (`points`) is set to `None`, which PySpark correctly interprets as `null` in the context of Spark SQL. Our specific objective is to target the missing entries for teams like Lakers, Hawks, and Wizards, ensuring that their null measurement is accurately populated using the corresponding estimation data available in the adjacent column. This controlled scenario provides a clear, verifiable case study for the behavior and effectiveness of the coalesce() function.

The standard setup procedure involves structuring the data rows, defining the column schema (specifying names and types), and then utilizing the `spark.createDataFrame()` method. Carefully observe the output of the initial DataFrame below. The visual representation clearly pinpoints the exact locations of the **null values** that must be addressed before this data can be reliably used for any machine learning or complex analytical task. Achieving a complete dataset is a non-negotiable step toward building reliable data products.

Example: How to Use coalesce() with Another Column in PySpark

Consider the following **PySpark** DataFrame, which details points scored by various basketball teams. Notice the gaps in the primary `points` measurement 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, showing nulls
df.show()

```
+-------+------+---------------+
| team|points|points_estimate|
+-------+------+---------------+
| Mavs| 18| 18|
| Nets| 33| 33|
| Lakers| null| 25|
| Kings| 15| 15|
| Hawks| null| 29|
|Wizards| null| 14|
| Magic| 28| 28|
+-------+------+---------------+
```

As clearly illustrated, the `points` column contains three distinct rows where the value is missing (`null`). Our requirement is to use the values from the `points_estimate` column to fill these specific **null values**, thereby creating a complete and ready-to-use dataset for subsequent analysis. This need for reliable, row-wise imputation is ubiquitous across industries, whether handling gaps in sensor readings, incomplete financial records, or survey data where primary responses are sometimes unavailable but secondary estimates exist.

To perform this dynamic imputation efficiently, we must import the necessary function from pyspark.sql.functions and then apply the structural modification using withColumn(). This combination represents the most optimized method for performing row-level transformations within the distributed execution model of **PySpark**, drastically outperforming less scalable approaches like unoptimized UDFs or costly data collection operations.

We execute the column-based replacement operation using the following concise syntax, ensuring the target column is placed first within the `coalesce()` arguments to preserve non-null data:

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

Analyzing the Result and Advanced PySpark Techniques

A meticulous examination of the resulting DataFrame confirms the absolute success of the operation. Every single one of the original **null values** in the `points` column has been seamlessly replaced by the corresponding entry from the `points_estimate` column. Specifically, the Lakers row now displays 25 points, the Hawks row 29 points, and the Wizards row 14 points. Crucially, all rows that originally contained valid, non-null data (Mavs, Nets, Kings, Magic) were left entirely untouched, validating the prioritization logic inherent in the coalesce() function.

The power of `coalesce()` stems from its ability to provide a clean, single-function solution for a complex conditional replacement task. While one could achieve the same result using explicit `when().otherwise()` logic--checking if the column is null and then substituting the estimate--the `coalesce()` function offers superior readability and is generally preferred for simple null-filling hierarchies. By explicitly placing the measured value first, we establish a robust preference hierarchy: prioritize the primary data source, and only fall back to the secondary source when necessary, thereby maximizing data integrity.

This technique is vital for any data engineer working in PySpark. Whenever a dataset variable possesses multiple potential sources of data with varying degrees of reliability or priority, `coalesce()` offers the most efficient and intelligent structure for merging these sources. For advanced scenarios involving more complex imputation logic (e.g., if we need to check if the estimate itself is greater than a certain threshold), the `when()` and `otherwise()` functions provide the necessary flexibility to construct intricate, multi-layered imputation rules based on external or internal conditions.

Additional Resources for PySpark Data Manipulation

Building scalable data pipelines in **PySpark** requires proficiency with a diverse set of functions

designed specifically for distributed environments. While the coalesce() function elegantly solves the column-to-column null replacement problem, related tasks in data cleansing, feature engineering, and aggregation demand mastery of other tools within the pyspark.sql.functions module.

A core tenet of writing efficient **PySpark** code involves understanding the operational difference between transformation functions (which define the execution plan, such as withColumn() or `filter()`) and action functions (which trigger the computation and return a result to the driver, such as `show()` or `collect()`). For complex conditional logic that goes beyond simple null replacement--such as applying different imputation methods based on the data type or the source system--the combination of `when()`, `otherwise()`, and `select()` provides the structural backbone for defining these rules.

We highly recommend that data professionals seeking to deepen their expertise explore the official Apache Spark documentation thoroughly. Focused study on optimizing data joins, understanding partitioning strategies, and mastering various aggregation functions will lay the essential groundwork for building truly scalable and robust data processing solutions. The ability to choose the right function for the job--whether it is the simplicity of `coalesce()` or the complexity of nested `when()` statements--is the hallmark of an effective **PySpark** developer.

The following tutorials explain how to perform other common tasks in **PySpark** crucial for advanced data cleansing and aggregation:

Tutorial on using `groupBy()` and various aggregation functions (e.g., `sum`, `avg`).

Guide to optimizing data filtering using `where()` or `filter()` clauses for performance.

Deep dive into optimizing joins between large, disparate DataFrames using broadcast hints and proper key selection.