PySpark: Use fillna() with Another Column

Authored by

stats writer

November 16, 2025

1. The Challenge of Missing Data in PySpark

Handling null values is a fundamental requirement in data cleansing and preparation tasks across all data engineering pipelines. In the context of large-scale data processing using PySpark, missing data must be addressed efficiently to prevent downstream analytical errors or model bias. While the standard **`fillna()`** method in PySpark is highly effective for substituting nulls with fixed scalar values (like 0, the mean, or the median), it does not natively support substituting nulls based on the corresponding value of a secondary column. This limitation necessitates a different approach when the imputation strategy relies on contextual data already present within the DataFrame.

The common requirement is often this: if Column A contains a null value, we must check Column B (an estimate, backup sensor reading, or pre-calculated default) and use its value to populate Column A. This technique ensures that data integrity is maintained by leveraging the richest available information for each record. To achieve this sophisticated, row-wise conditional filling, we must utilize the powerful **`coalesce()`** function provided within `pyspark.sql.functions`. The implementation is concise yet requires precise understanding of how Spark handles column transformations, primarily through the **`withColumn()`** method.

2. Understanding the Coalesce Function for Imputation

The `coalesce` function is the core utility for implementing this dynamic null replacement strategy in PySpark. Unlike simple conditional logic (like `if/else` or `when/otherwise`), **`coalesce()`** is specifically optimized to return the first non-null expression among a sequence of columns or values passed to it. This functionality is perfectly suited for our goal: we list the primary column first, followed by the fallback column(s). Spark processes these columns sequentially for every row in the distributed dataset.

If the first column (our target column, say 'points') is non-null, **`coalesce()`** immediately returns that value and moves on to the next row. If, however, 'points' contains a null value, **`coalesce()`** proceeds to evaluate the next expression--which is our backup column ('points_estimate'). If 'points_estimate' is also null, it would proceed to the third argument, and so on. If all specified arguments are null, the function ultimately returns null for that row. This ensures a clean and highly efficient mechanism for imputation based on priority, effectively simulating the behavior of **`fillna()`** using a secondary column as the substitute value.

3. Implementing Column-Based Null Replacement Syntax

To execute the null replacement using a secondary column, we combine the **`coalesce()`** function with the **`withColumn()`** DataFrame transformation. The **`withColumn()`** method is crucial here because it allows us to either create a new column or overwrite an existing one based on a

specified expression. In this scenario, we overwrite the existing column that contains the nulls ('points') with the result of the **`coalesce()`** expression, thereby filling the missing data points only where necessary.

The generalized syntax requires importing **`coalesce`** from the functions module and then applying the transformation on the existing DataFrame (`df`). The expression passed to **`withColumn()`** specifies the target column name, followed by the function call: `coalesce(target_column, fallback_column)`. This powerful one-liner encapsulates the entire imputation logic, ensuring that the original data is preserved unless a null needs filling.

Here is the precise structure utilized to achieve this imputation goal, focusing on replacing nulls in the **`points`** column with values from the **`points_estimate`** column:

from pyspark.sql.functions import coalesce

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

This particular example demonstrates how to replace null values in the **points** column with corresponding values from the **points_estimate** column, leveraging the highly optimized **`coalesce()`** function for efficient, distributed processing.

4. Detailed Example: Setting Up the PySpark DataFrame

To demonstrate this functionality in a practical context, let us construct a sample DataFrame representing basketball statistics. This dataset includes three fields: `team`, `points` (the recorded score), and `points_estimate` (an estimated or backup score). The crucial element in this setup is the intentional inclusion of null values in the primary `points` column for several teams (Lakers, Hawks, Wizards).

This scenario perfectly mirrors real-world data issues, where primary metrics might occasionally be missing due to system failures or data entry errors, but a reliable secondary estimate exists. Our objective is to rigorously test the **`coalesce()`** function by using the `points_estimate` column to seamlessly fill these gaps in the `points` column. The initialization process involves setting up a `SparkSession`, defining the data structure, naming the columns, and finally creating and displaying the resulting distributed dataset.

The following code block outlines the creation and initial state of our demonstration DataFrame in PySpark:

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


5. Executing the Coalesce Transformation

Now that the DataFrame is established and the missing data points are clearly visible in the `points` column, we proceed with the transformation using **`withColumn()`** and `coalesce`. The objective remains focused: for any row where `points` is null, the value from `points_estimate` must be utilized instead. Crucially, rows that already contain non-null values (e.g., 'Mavs' or 'Nets') must remain unchanged, preserving the highest quality data available.

This method is highly scalable and efficient because PySpark executes the **`coalesce()`** function natively across the distributed partitions, making it suitable for petabyte-scale data operations. We

define the new (overwritten) `points` column by calling **`coalesce('points', 'points_estimate')`**. This instruction prioritizes the first column listed ('points'); only upon finding a null does it look at the second column ('points_estimate').

The implementation code and the resultant DataFrame, demonstrating the successful imputation, are presented below. Note the distinct change in the `points` column for the 'Lakers', 'Hawks', and 'Wizards' rows:

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|
+-------+------+---------------+
```

6. Analyzing the Imputation Results

Upon reviewing the output DataFrame, the effectiveness of the **`coalesce()`** operation is immediately apparent. Specifically, the rows corresponding to 'Lakers', 'Hawks', and 'Wizards' originally contained null values in the `points` column. Following the transformation, these nulls have been systematically replaced by the corresponding integers from the `points_estimate` column (25, 29, and 14, respectively).

It is equally important to confirm that the existing valid data was not corrupted or overwritten. For teams like 'Mavs', 'Nets', 'Kings', and 'Magic', the `points` column already held non-null values. Because **`coalesce()`** prioritizes the first non-null argument, the original values (18, 33, 15, 28) were retained. This behavior confirms that **`coalesce()`** provides a highly robust method for targeted imputation, guaranteeing that primary data is favored over fallback data whenever possible. This technique is often superior to simple imputation based on aggregated statistics, as it preserves the unique row-level context and utilizes richer input data.

7. Extending Coalesce for Multiple Fallback Columns

One of the distinct advantages of the `coalesce` function over basic conditional logic is its natural ability to handle multiple fallback columns with ease. If we have a scenario where we might have a primary value (Column A), a primary estimate (Column B), and a secondary, less reliable estimate (Column C), **`coalesce()`** allows us to define a strict order of preference. This is crucial for establishing data quality hierarchies in complex pipelines.

For example, if `points` is null, we check `points_estimate_1`. If that is also null, we check `points_estimate_2`. Only if all three are null will the resulting value remain null. The syntax for implementing this sequential priority is straightforward, requiring only the addition of subsequent column names as arguments to the function call. This hierarchical approach to data filling is invaluable in complex data cleaning pipelines where multiple data sources or imputation strategies are employed simultaneously.

The structure for multi-column fallback would look like this, demonstrating how easily the complexity can be managed within the PySpark framework, offering flexibility beyond standard **`fillna()`**:

from pyspark.sql.functions import coalesce

# Checks points -> points_estimate_1 -> points_estimate_2
df.withColumn('points', coalesce('points', 'points_estimate_1', 'points_estimate_2')).show()

This extensibility ensures that developers can prioritize data quality and completeness by establishing a robust chain of command for missing values, optimizing the usage of all available contextual information within the distributed DataFrame environment.

8. Alternative Approach: Using the When/Otherwise Structure

While `coalesce` is the most idiomatic and often the most performance-efficient way to handle sequential null replacement, it is also possible to achieve the same result using the conditional logic provided by **`when()`** and **`otherwise()`** functions in PySpark. This alternative is particularly useful when the replacement condition is more complex than simply checking for nullity, perhaps involving multiple criteria or specific thresholds that define "missing" or "invalid" data.

To simulate the null replacement behavior, we must check explicitly if the target column is null using **`pyspark.sql.functions.col(column_name).isNull()`**. If this condition evaluates to true, we use the value from the fallback column; otherwise, we retain the original value from the target column. While this provides greater control over complex conditions, it is generally less readable and slightly more verbose than the declarative simplicity of **`coalesce()`** when the sole goal is

prioritized null filling.

The code snippet demonstrating the `when/otherwise` approach for column-based filling showcases the increased structural complexity, though it offers flexibility for future modifications:

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

df_alternative = df.withColumn('points_new',
when(col('points').isNull(), col('points_estimate'))
.otherwise(col('points'))
)

df_alternative.show()
```

In summary, for basic null replacement using a prioritized list of columns, `coalesce()` should be the preferred tool due to its native optimization for this exact task. For highly customized, multi-condition logic that extends beyond simple null checks, the `when/otherwise` structure provides the necessary explicit control over the imputation process within your data cleaning workflow.

Note: You can find the complete documentation for the PySpark **coalesce()** function online.