How to Combine Multiple Columns into One Using PySpark’s Coalesce Function

Authored by

stats writer

January 19, 2026

Introduction to Data Coalescing in PySpark

Working with large datasets often involves handling missing or incomplete data, commonly represented as **null values**. In the context of data engineering and analysis using PySpark, a common requirement is to consolidate information scattered across several columns into a single, unified column. This process, known as **coalescing**, is crucial when you have multiple potential sources for a data point, but you only need the first valid entry.

PySpark provides a highly efficient and declarative way to achieve this through the built-in `coalesce` function. Unlike simple conditional statements that might require complex nesting, the `coalesce` function is specifically optimized for checking multiple columns sequentially and returning the first available piece of information. This method significantly streamlines data cleaning and feature engineering pipelines, ensuring that the resulting DataFrame is as complete as possible before subsequent transformations or modeling steps are applied.

The core principle revolves around resolving ambiguity created by sparse data. By defining an order of preference among the columns, we instruct Spark which column's value should be chosen if the preferred column contains a **null value**. This capability is foundational for data standardization tasks, especially when integrating data from disparate sources where column availability might vary row by row.

Understanding the PySpark `coalesce` Function

The `coalesce` function operates on a set of input columns and evaluates them row by row, from left to right. Its fundamental rule is straightforward: it returns the value of the very first column in the sequence that is not **null value** for that specific row. If, hypothetically, all specified columns for a given row contain **null values**, the function itself will return null for that row in the new consolidated column.

This sequential evaluation makes the order of columns within the function parameters highly significant. For instance, if you prioritize data accuracy from a primary source, that source column should always be listed first in the `coalesce` function. Secondary, less reliable, or derived data columns would follow. This design allows developers to implement specific data hierarchies directly within their transformation logic, providing robust control over data imputation without resorting to manual checks.

Crucially, the `coalesce` function is part of the `pyspark.sql.functions` module, meaning it must be explicitly imported before use. Once imported, it acts as a column expression that can be seamlessly integrated with standard PySpark methods like `withColumn`, which is typically used to add a new column or replace an existing one based on a derived expression. This integration simplifies the syntax required for complex data manipulations, adhering to the functional

programming paradigms favored by Spark.

Syntax and Implementation for Column Coalescing

To coalesce values from multiple columns into one column in a PySpark DataFrame, you must first import the required function and then apply it using the `withColumn` method. The syntax is concise and immediately communicates the intent of the operation.

The structure involves calling `df.withColumn()`, specifying the name of the new column, and then passing the list of existing columns, in their preferred order, into the `coalesce` function. Notice how the input columns are referenced directly as attributes of the DataFrame object (e.g., `df.points`).

The following code block demonstrates the essential syntax for importing and executing the `coalesce` operation, targeting statistical columns such as points, assists, and rebounds:

```
from pyspark.sql.functions import coalesce

#coalesce values from points, assists and rebounds columns
df = df.withColumn('coalesce', coalesce(df.points, df.assists, df.rebounds))
```

This specific implementation creates a new column--here temporarily named **`coalesce`** for demonstration purposes--that selectively retrieves the first available, **non-null value** from the **`points`** column. If **`points`** is null, it moves to **`assists`**, and finally to **`rebounds`**. This establishes a clear hierarchy: points > assists > rebounds.

Example: Coalescing Basketball Player Statistics in PySpark

To illustrate the power and simplicity of this function, consider a realistic scenario involving basketball player performance statistics. We often encounter data where certain metrics might not be recorded for every game or player, resulting in sparse columns. Our goal is to derive a single 'Primary Metric' column that prioritizes scoring (points), followed by passing (assists), and finally rebounding (rebounds).

First, we must initialize a SparkSession and define the sample data. This sample data is intentionally structured to contain numerous **null values** (represented by `None` in Python lists) to demonstrate how the `coalesce` logic handles missing data across different rows and columns.

The following setup code initializes the environment and creates the base DataFrame:

```
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()

+------+-------+--------+
|points|assists|rebounds|
+------+-------+--------+
| null| null| 3|
| null| 7| 4|
| 19| 7| null|
| null| 9| null|
| 14| null| 6|
+------+-------+--------+
```

Applying the Coalescing Logic to Sparse Data

With the initial **DataFrame** loaded, which clearly shows missing values in the `points` and `assists` columns for several records, we can now execute the transformation. The objective is to populate the new column, `coalesce`, by scanning the three statistical columns in order: `points` first, then `assists`, and finally `rebounds`. This sequence ensures that player scoring statistics take precedence over other metrics whenever they are available.

The use of `df.withColumn` is essential here. It allows us to derive the new column directly from the existing ones without modifying the original data structure, which is a key practice in immutable data processing environments like PySpark. If a value exists in `points`, that value is selected immediately, and the remaining columns (`assists`, `rebounds`) are ignored for that specific row calculation.

We repeat the import command for clarity, followed by the transformation and the resulting

DataFrame display:

from pyspark.sql.functions import coalesce

#coalesce values from points, assists and rebounds columns
df = df.withColumn('coalesce', coalesce(df.points, df.assists, df.rebounds))

#view updated DataFrame
df.show()

```
+------+-------+--------+--------+
|points|assists|rebounds|coalesce|
+------+-------+--------+--------+
| null| null| 3| 3|
| null| 7| 4| 7|
| 19| 7| null| 19|
| null| 9| null| 9|
| 14| null| 6| 14|
+------+-------+--------+--------+
```

Detailed Analysis of Coalesce Results

Observing the output DataFrame, we can confirm that the `coalesce` column successfully aggregates the statistics according to our defined preference hierarchy. Each row clearly demonstrates the sequential logic applied by the function, returning the **non-null value** encountered first when reading from left to right across the source columns (points, assists, rebounds).

This result validation is critical for ensuring that data integrity is maintained throughout the transformation process. The simplicity of the `coalesce` function belies its importance in preparing data where fallback mechanisms are required to handle expected data sparsity. We can trace the derivation of each value in the new column:

First row: The first **non-null value** was **3**.
Second row: The first non-null value was **7**.
Third row: The first non-null value was **19**.
Fourth row: The first non-null value was **9**.
Fifth row: The first non-null value was **14**.

Common Use Cases for PySpark Coalescing

The application of the `coalesce` function extends far beyond statistical record aggregation. It is a highly versatile tool fundamental to various aspects of big data processing and cleanup. One common use case is handling customer contact information, where data might come from CRM systems, web forms, or third-party vendors. You might prioritize a verified email address (`email_primary`) but fall back to a web-scraped email (`email_secondary`) if the primary one is missing.

Another crucial application is data default assignment. If you have a column representing a user setting (e.g., `user_config_value`) but it contains nulls, you can use `coalesce` to assign a predefined system default value. By placing the system default--often a literal constant created using `lit()`--as the last argument in the `coalesce` function, you ensure that every row receives a **non-null value**, thereby eliminating nulls completely from that column. This technique is often mandatory when preparing data for machine learning models that cannot tolerate null input features.

Furthermore, `coalesce` is invaluable during ETL (Extract, Transform, Load) processes when merging schema changes or dealing with versioned data. If a new version of data includes a preferred column (e.g., `new_price`) but older records only have the legacy column (`old_price`), `coalesce(df.new_price, df.old_price)` cleanly creates a single price column that utilizes the newest data where available. This ensures forward compatibility and simplifies downstream processing logic considerably.

Conclusion and Further Resources

The `coalesce` function in PySpark is an indispensable tool for data preparation, enabling users to efficiently merge data from multiple sources based on priority and availability. By selecting the first **non-null value** sequentially, it allows data engineers to define clear fallback mechanisms, drastically reducing data sparsity and improving the quality of input features for downstream processes. Mastering this function is key to writing clean, high-performance Spark code for large-scale data manipulation.

For developers seeking deep dives into PySpark's extensive capabilities, continuous reference to the official documentation is highly recommended. Understanding the nuances of functions like `coalesce` ensures that your data pipelines are robust and scalable.

You can find the complete documentation for the PySpark **coalesce** function to explore additional parameters and related functions.

The following tutorials explain how to perform other common tasks in PySpark: