How to Union PySpark DataFrames with Different Columns: A Step-by-Step Guide

Authored by

stats writer

February 4, 2026

How to Use PySpark to Union DataFrames with Different Columns

Introduction to PySpark and Data Integration Challenges

PySpark serves as the Python API for Apache Spark, providing an immensely powerful framework for large-scale data processing and analysis. When handling vast datasets, data often originates from multiple, disparate sources. Integrating this data frequently requires combining different structures, which can be challenging if standard merging techniques are employed. Standard relational database operations or simple concatenation methods often fail when the source tables, known as DataFrames in PySpark, possess non-identical schema, meaning they have different column names, different column orders, or simply different sets of columns altogether.

The ability to perform a robust data union between such DataFrames is critical for effective data integration, especially in scenarios like ETL (Extract, Transform, Load) pipelines or combining results from different experimental runs. Historically, achieving this in distributed computing environments required significant preprocessing--like renaming columns, adding placeholder columns, or manually ensuring schema alignment--before the union could proceed. PySpark offers a streamlined solution to this complex problem, allowing developers to combine these dissimilar structures efficiently and gracefully handle missing data elements.

Understanding Standard PySpark Union Operations

In PySpark, the most basic method for combining rows from two DataFrames vertically is the standard `union()` transformation. However, this base function relies purely on the positional order of the columns. If `df1` has columns (A, B, C) and `df2` has columns (X, Y, Z), a standard `df1.union(df2)` operation will align X with A, Y with B, and Z with C, regardless of their semantic meaning or actual column names. This approach is prone to errors and data corruption when schemas diverge, making it suitable only when the column names and types are perfectly synchronized and in the exact same order.

When dealing with large-scale data lakes or federated data systems, strict positional alignment is often impossible or impractical to enforce. Data governance standards might vary across different sources, leading to minor variations in column naming (e.g., "Customer_ID" vs. "CustomerID"). For these reasons, relying on positional unions is not feasible when column sets differ, necessitating a more intelligent, name-based approach to schema reconciliation. This is where PySpark's enhanced functionality, specifically tailored for schema flexibility, becomes indispensable for professional data engineers.

The Power of unionByName: Handling Schema Mismatch

To overcome the limitations of the standard positional union, PySpark introduced the `unionByName` method. As the name suggests, this method performs the union operation by matching columns based on their respective names, rather than their index position within the DataFrame. This capability is fundamental when merging DataFrames that have columns in different orders or possess columns that are unique to one dataset.

While `unionByName` is powerful, its default behavior still expects both DataFrames to share the same set of column names. If a column exists in `df1` but not in `df2`, the default behavior will raise an error. To enable true integration where schemas are allowed to differ--the core requirement for combining diverse data sources--we must leverage a critical function parameter: `allowMissingColumns`. By setting this parameter to `True`, we instruct PySpark to proceed with the union, inserting placeholder values for any column that is missing from one of the constituent DataFrames. This flexibility is what transforms complex data merging tasks into a single, clean PySpark operation.

Implementing the Solution: Syntax and Key Parameters

To effectively perform a union on two PySpark DataFrames that contain different columns, the syntax utilizes the `unionByName` function paired with the schema flexibility argument. This specific combination ensures that the column names are used for alignment, and any discrepancies in the column sets are handled automatically without manual schema manipulation.

The required syntax structure is straightforward and highly declarative, clearly stating the intention to merge schemas intelligently:

The following is the precise syntax used to execute the union operation:

df_union = df1.unionByName(df2, allowMissingColumns=True)

This command executes a union between the PySpark DataFrames named **df1** and **df2**. By including the argument **`allowMissingColumns=True`**, we explicitly override the default schema checking mechanism. This specification is crucial, as it tells the PySpark engine that the set of column names between the two DataFrames are intentionally allowed to differ. When a column is present in one DataFrame but absent in the other, PySpark automatically adds the missing column and populates its corresponding rows in the resulting DataFrame with null values. This mechanism guarantees a complete and consistent combined schema.

Detailed Practical Example: Setting Up Source DataFrames

To demonstrate this functionality in a practical context, consider two distinct DataFrames representing sports statistics. The first DataFrame, **df1**, contains standard team performance metrics, while the second DataFrame, **df2**, focuses on auxiliary player statistics. Notice that the schemas are clearly different, but they share a common identifier column: `team`.

We begin by defining and creating **df1**, which includes columns for `team`, `conference`, and `points`. This DataFrame represents one structure of our input data, focused on general scoring and location metrics.

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

#define data
data1 = ,
,
,
,
,
]

#define column names
columns1 =

#create DataFrame
df1 = spark.createDataFrame(data1, columns1)

#view DataFrame
df1.show()

+----+----------+------+
|team|conference|points|
+----+----------+------+
A	East	11
B	East	8
C	East	31
D	West	16
E	West	6
F	East	5
+----+----------+------+

Next, we define **df2**. This DataFrame introduces a different metric, `assists`, and crucially omits the `conference` and `points` columns present in **df1**. The only common column is `team`, which allows the data to be treated as independent rows relating to the same general entity (the team). This difference in schema highlights why the standard `union()` method would fail or produce meaningless results, necessitating the use of the advanced `unionByName` method.

#define data
data2 = ,
,
,
,
,
]

#define column names
columns2 =

#create DataFrame
df2 = spark.createDataFrame(data2, columns2)

#view DataFrame
df2.show()

+----+-------+
|team|assists|
+----+-------+
G	4
H	8
I	11
J	5
K	2
L	4
+----+-------+

Executing the Union Operation with Differing Columns

With both DataFrames defined and loaded into the Spark context, we can now execute the powerful union operation. We apply `unionByName` to **df1**, passing **df2** as the target DataFrame, and most importantly, setting `allowMissingColumns=True`. This single line of code handles the entire process of schema discovery, reconciliation, and row merging across a distributed

architecture.

The resulting DataFrame, `df_union`, will automatically contain the superset of all unique columns found in both **df1** and **df2**. The union process ensures that data from **df1** is correctly mapped to the columns `team`, `conference`, and `points`, while data from **df2** is mapped to `team` and `assists`. The cross-mapping is handled seamlessly by PySpark, resulting in a single, cohesive dataset ready for further analytical processing.

We use the following commands to perform the union and display the final, integrated DataFrame:

```
#perform union with df1 and df2
df_union = df1.unionByName(df2, allowMissingColumns=True)

#view final DataFrame
df_union.show()
```

```
+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|  A|      East|    11|   null|
|  B|      East|     8|   null|
|  C|      East|    31|   null|
|  D|      West|    16|   null|
|  E|      West|     6|   null|
|  F|      East|     5|   null|
|  G|      null|  null|      4|
|  H|      null|  null|      8|
|  I|      null|  null|     11|
|  J|      null|  null|      5|
|  K|      null|  null|      2|
|  L|      null|  null|      4|
+----+----------+------+-------+
```

Analyzing the Resulting DataFrame and Null Value Handling

A careful examination of the final DataFrame, `df_union`, reveals a unified schema containing four columns: `team`, `conference`, `points`, and `assists`. The total number of rows is the sum of the rows from the two input DataFrames, twelve rows in this case. Crucially, the resulting DataFrame contains all of the rows from both source DataFrames, successfully integrating the data vertically.

When observing the rows originating from **df1** (teams A through F), the `assists` column contains

`null` values, because this column did not exist in the original **df1** schema. Conversely, the rows originating from **df2** (teams G through L) contain `null` values in the `conference` and `points` columns. This mechanism of inserting `null` values for missing fields is the intended behavior of `unionByName(..., allowMissingColumns=True)`, ensuring that all data points are preserved while maintaining a rectangular, uniform structure across the combined dataset.

This automatic handling of missing columns prevents data loss and simplifies subsequent analysis. Data scientists can now work with a single, consolidated DataFrame, applying appropriate filtering or imputation techniques to manage the introduced `null` values, rather than spending time on manual schema alignment prior to the union.

Best Practices and Performance Considerations

While `unionByName` is exceptionally useful, especially with `allowMissingColumns=True`, practitioners should adhere to best practices to ensure optimal performance and data quality. It is important to remember that PySpark operates on distributed principles. Although the union operation itself is generally highly optimized, the resulting DataFrame must accommodate the widest possible schema. If the difference in schemas is vast--for instance, if one DataFrame has hundreds of unique columns not present in the other--this can lead to a very wide, sparsely populated DataFrame, potentially impacting memory usage and query efficiency in later stages.

Furthermore, users must be mindful of data types. PySpark attempts to infer compatible data types during the union process. If two DataFrames have the same column name but assign conflicting types (e.g., one uses an integer type while the other uses a string type), PySpark might throw an error or implicitly cast one type to another, potentially leading to data truncation or unexpected behavior. Before performing an advanced union, it is advisable to inspect the schemas using `df.printSchema()` and explicitly cast any conflicting column types to a common, robust type, such as ensuring all numerical identifiers are consistently handled as long integers or strings.

Further Resources for Advanced PySpark Operations

The **`unionByName`** function is a cornerstone tool for complex data ingestion and merging tasks within the PySpark ecosystem. Understanding its nuances, particularly the use of **`allowMissingColumns`**, is essential for any professional working with heterogenous big data sources. For those seeking deeper technical information regarding this method, the complete and authoritative documentation is readily available.

Additional tutorials and official resources explain how to perform other common tasks and transformations essential for leveraging the full power of distributed computing in PySpark, including advanced join strategies, window functions, and user-defined functions (UDFs). Continuing to explore these advanced features will enhance efficiency in managing large-scale

data workflows.

The following resources provide excellent next steps for mastering PySpark DataFrames:

Understanding various join types (inner, outer, left anti).

Techniques for managing null values and data quality checks after schema integration.

Optimizing data partitioning and shuffling strategies for large union operations.