PySpark Tutorial: Combining DataFrames with Differing Columns

Authored by

Mohammed looti

November 11, 2025

The Limitations of Standard Positional PySpark Union

In the domain of large-scale data engineering, utilizing PySpark is standard practice for distributed processing. A frequent requirement in data preparation involves consolidating two or more datasets vertically, a procedure typically achieved using the standard `union()` operation. While highly optimized for performance, this method operates under a strict set of constraints: it concatenates rows based purely on the sequential position of columns. This means that for a standard union to execute successfully and produce meaningful results, both source DataFrames must possess an identical schema, including the exact same number of columns, matching data types, and critically, the columns must be ordered in the same sequence.

However, real-world data integration scenarios rarely conform to this ideal. Data often originates from disparate systems, evolves over time, or is collected through separate ETL pipelines, inevitably leading to schema discrepancies. These discrepancies, often termed **schema drift**, manifest as differences in column names, data types, or the presence of entirely unique columns in one dataset but not the other. Attempting a conventional positional union in such situations is guaranteed to fail or, worse, result in silently corrupted data where values are aligned to the wrong fields. For instance, if one DataFrame has (Name, Age) and the second has (Age, Name), a positional union would combine Name with Age, leading to severe data integrity issues.

To build truly robust and fault-tolerant data pipelines, engineers require a method that abstracts away the physical order of columns and focuses instead on logical alignment. This necessity drives the use of advanced PySpark features designed specifically to manage and reconcile these structural variations, ensuring that data merging remains seamless and accurate even when the input schemas are mismatched. This approach is fundamental for maintaining data quality when dealing with dynamic data sources.

Introducing unionByName for Schema-Flexible Merging

When faced with the challenge of combining DataFrames that have differing column sets, the appropriate tool within the PySpark API is the unionByName method. As its name suggests, this function performs the union operation by matching and combining rows based on the explicit **names** of the columns, rather than relying on their arbitrary positional indices. This fundamental shift from positional matching to named matching provides the flexibility necessary to handle real-world data heterogeneity.

The standard behavior of `unionByName` is already a significant improvement over `union()`, as it automatically handles cases where columns are merely out of order. However, by default, it still imposes a restriction: all columns present in the first DataFrame must also exist in the second DataFrame to ensure a coherent merge. If a column is missing from the first but present in the second, that column is simply added to the resulting schema. This default setting is useful but does

not solve the primary problem where columns are missing from the second DataFrame relative to the first.

To unlock the full potential of merging DataFrames with truly dissimilar schemas--where columns are missing from either or both inputs--we must utilize a specialized argument. This capability is essential for operations such as consolidating monthly reports that track different metrics, or merging log streams where various events include unique fields. By prioritizing the column name, unionByName ensures that only semantically similar data is stacked together, regardless of where it appears in the source file structure.

The Critical Role of allowMissingColumns=True

The key to successfully performing a union operation between two DataFrames with non-overlapping or partially overlapping schemas lies in the optional argument: **allowMissingColumns=True**. This parameter acts as a waiver, relaxing the strict schema requirements imposed by default. When set to **True**, this argument instructs PySpark to allow the set of column names in the input DataFrames to differ completely, accommodating scenarios where columns exist only in the first DataFrame, only in the second DataFrame, or in neither.

The operational mechanics of using **allowMissingColumns=True** are straightforward yet powerful. The resulting schema of the combined DataFrame (the union result) becomes the **superset** of all unique columns found across all source DataFrames. If a row originates from a source DataFrame that did not contain a specific column present in the final merged schema, that column slot is automatically populated with a null value for those records. This process is often referred to as **schema reconciliation** or **schema evolution management**.

This mechanism is not merely a convenience; it is a critical feature for building resilient data integration layers. It eliminates the necessity of pre-processing source DataFrames--such as adding placeholder columns with default values--before the union can occur. Instead, the framework handles the schema alignment automatically, significantly reducing boilerplate code and the complexity of managing schema changes upstream. The concise syntax below demonstrates the implementation required for this advanced union operation:

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

Practical Demonstration: Setting Up Dissimilar Schemas in PySpark

To effectively illustrate the utility of unionByName, we must first establish a consistent execution environment and define the source data. We begin by initializing a SparkSession, which is the entry point for all functionality in Apache Spark. Following this, we create two distinct DataFrames,

df1 and **df2**, intentionally designed to possess non-matching schemas to simulate a typical data merging challenge.

Our first DataFrame, **df1**, focuses on primary statistics. It contains key columns such as **team** (serving as the primary identifier), **conference**, and **points**. This represents a foundational dataset with a specific structural layout. Conversely, **df2** is designed to hold supplementary or specialized metrics. It shares the common identifier **team** but introduces a new metric, **assists**, while conspicuously omitting the **conference** and **points** columns. This deliberate variance in schemas sets the stage for demonstrating the reconciliation capabilities of the name-based union.

The following code block outlines the setup of the Spark environment and the creation and display of **df1**, confirming its initial structure:

```
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 proceed to define **df2**, which possesses the contrasting schema. This step is critical, as it provides the necessary structural difference to test the resilience and intelligence of the name-based union operation. The existence of the shared column **team** provides the common semantic link, but the divergence in metrics (points/conference vs. assists) mandates the use of flexible merging logic.

#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 Flexible Union and Analyzing the Unified Schema

With the source DataFrames prepared, the final action involves invoking the unionByName function, explicitly enabling schema flexibility via **allowMissingColumns=True**. This single line of

code instructs PySpark to combine the disparate records from **df1** and **df2**, resulting in a single, coherent DataFrame where column alignment is guaranteed by name.

The resulting combined DataFrame, conventionally named **df_union**, will contain all records from both sources. Crucially, its schema will represent the union of all unique columns encountered: **team**, **conference**, **points**, and **assists**. This demonstrates the seamless schema evolution managed by the function, accommodating all known fields across the input datasets.

Executing the union operation and displaying the resultant DataFrame provides concrete evidence of this integration:

```
#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|
|  A|  East|     8|   null|
|  A|  East|    31|   null|
|  B|  West|    16|   null|
|  B|  West|     6|   null|
|  C|  East|     5|   null|
|  A|  null|  null|      4|
|  A|  null|  null|      8|
|  A|  null|  null|     11|
|  B|  null|  null|      5|
|  B|  null|  null|      2|
|  C|  null|  null|      4|
+----+----------+------+-------+
```

The output confirms that **df_union** now holds all 12 records (6 from **df1** and 6 from **df2**). Notice the distinct pattern in the final table: the first six rows, derived from **df1**, contain valid data for **conference** and **points** but have nulls in **assists**. Conversely, the last six rows, sourced from **df2**, contain valid **assists** data but have nulls in **conference** and **points**. This structured null imputation is the hallmark of a successful name-based union with missing column allowance.

Deep Dive into Null Value Behavior and Data Coherence

Understanding the mechanism of null value imputation is vital when leveraging **allowMissingColumns=True**. When the union operation expands the schema to include all unique columns (the superset schema), any row sourced from a DataFrame lacking a specific column must have a placeholder value inserted to maintain structural consistency. This placeholder is the SQL null marker.

For the rows originating from **df1**, which only defined the **team**, **conference**, and **points** fields, the newly introduced **assists** column is automatically populated with a null value. Similarly, rows from **df2**, which only contained **team** and **assists**, receive null markers in the **conference** and **points** columns. This process ensures that every record in the merged dataset strictly adheres to the unified schema, preventing schema misalignment or type coercion errors.

This behavior is highly desirable in data warehousing and analytic contexts, as the presence of null explicitly signifies that the corresponding metric was simply not collected or available in the original source dataset, rather than representing a zero or a corrupted value. Data engineers must therefore incorporate subsequent processing steps to correctly handle these generated nulls, whether through filtering, imputation based on domain knowledge, or using conditional logic tailored to the business requirements. The inherent flexibility provided by **unionByName** allows for rapid integration of varied datasets without manual schema synchronization.

Conclusion and Further Learning Resources

The ability to combine DataFrames with dissimilar schemas using unionByName and the **allowMissingColumns=True** parameter is a cornerstone of modern, flexible data pipeline design in PySpark. This technique moves beyond the restrictive requirements of positional unions, empowering engineers to integrate evolving and heterogeneous data sources seamlessly.

Mastery of PySpark requires proficiency in various data manipulation techniques beyond simple unions. To further enhance your skills in managing and transforming distributed datasets, consider exploring advanced operations such as complex joins, window functions, and advanced schema definition strategies.

Official PySpark Documentation: Provides detailed technical specifications and examples for the entire API, including the `unionByName` function.

Advanced DataFrame Joins: Tutorials focusing on horizontal merging techniques (inner, outer, left, right joins) which contrast with the vertical stacking nature of unions.

PySpark Schema Definition and Evolution: Guides on how to explicitly define schemas using

`StructType` and manage schema changes over time for performance and integrity.