How to Combine DataFrames Vertically in PySpark Using `union`

Authored by

stats writer

January 20, 2026

Introduction to Vertical Concatenation in PySpark

Vertically concatenating DataFrames in PySpark is a fundamental operation used extensively in large-scale data processing pipelines. This process, often referred to as appending or stacking, involves combining two or more distinct DataFrames by placing the rows of one DataFrame directly underneath the rows of another, resulting in a single, unified dataset. The primary mechanism for achieving this combination is through the use of the `union` family of functions, which efficiently manage the distributed nature of the data structures within the PySpark environment.

This technique is vital when dealing with partitioned datasets--for instance, when data is segregated by time period (e.g., monthly sales logs) or geographical region, but needs to be analyzed holistically. By vertically combining these separate, identically structured datasets, analysts can prepare the data for comprehensive processing, aggregation, and machine learning model training without the overhead of complex joins. It is crucial for successful vertical concatenation that all input DataFrames share the same number of columns and compatible column types, although the order of columns might be handled differently depending on the specific union function employed.

In contrast to horizontal concatenation (joining), which adds columns side-by-side based on a shared key, vertical concatenation focuses purely on expanding the number of records. Understanding the optimized functions provided by PySpark for this purpose is essential for maintaining performance and scalability, especially when handling petabyte-scale data where traditional row-by-row appending would be prohibitively slow. The following sections detail the specific methods and robust syntax recommended for performing this operation efficiently across multiple DataFrames.

The Role of `union` and `unionAll` in PySpark

The backbone of vertical concatenation in PySpark relies on the `union` methods. Historically, `unionAll` was the function used for this purpose. The `unionAll` method simply appends rows without attempting to reconcile schema differences or remove duplicates, assuming that the schema (number and order of columns) is identical across all input DataFrames. While fast, this method lacks safety, potentially leading to misalignment if columns were subtly reordered between datasets.

Modern DataFrame operations typically utilize the standard `union` function. In PySpark (from version 2.0 onwards), the `union` function behaves exactly like `unionAll` (it does not perform distinct aggregation like SQL `UNION`), meaning it appends all rows, including duplicates, and requires that the schemas are identical in terms of column count and order. For scenarios where DataFrames have the same columns but potentially in different orders, the safer and more flexible

function is `unionByName`, which aligns the columns based on their names before combining the rows, automatically inserting null values for columns present in one DataFrame but missing in another.

When working with multiple DataFrames--more than just two--we need an iterative approach to apply the binary `union` operation across the entire list of DataFrames. Directly chaining multiple `.union()` calls can become cumbersome and difficult to read, especially with a large number of inputs (e.g., `df1.union(df2).union(df3).union(df4)`). Therefore, standard practice involves using Python's functional tools, specifically the `reduce` function from the `functools` module, to efficiently handle the sequential accumulation of results. This approach ensures cleaner code and robust execution when stacking numerous distributed data structures.

Applying `reduce` for Concatenating Multiple DataFrames

When the task involves combining three or more DataFrames, the Python standard library function `reduce` offers an elegant and efficient solution. The reduce function applies a specified function cumulatively to the items of an iterable, reducing the iterable to a single resultant value. In the context of DataFrame concatenation, we use `reduce` to iteratively apply the `union` (or `unionAll`, depending on the environment) method across a list of DataFrames.

This powerful pattern allows for the concise handling of any arbitrary number of DataFrames, avoiding the need for explicit loops or verbose chaining. The function passed to `reduce` must accept two DataFrames as arguments and return a single combined DataFrame. We leverage the `DataFrame.unionAll` method (or `DataFrame.union`) directly as the reduction function, as it perfectly fits this required signature, operating on the list sequentially from left to right.

The following canonical syntax illustrates how to vertically concatenate multiple DataFrames named `df1`, `df2`, and `df3` into one comprehensive DataFrame called `df_all`:

To vertically concatenate multiple DataFrames in PySpark, use the following functional programming pattern:

```
from functools import reduce
from pyspark.sql import DataFrame

# specify DataFrames to concatenate
df_list =

# vertically concatenate all DataFrames in list using the reduce function
df_all = reduce(DataFrame.unionAll, df_list)
```

In this example, the reduce function iteratively takes the first two DataFrames (`df1` and `df2`), applies `unionAll`, then takes the resultant DataFrame and applies `unionAll` with `df3`, continuing until the list is exhausted and a single final DataFrame, `df_all`, is produced. This is highly scalable and adheres to Python's best practices for functional iteration.

Understanding the Context: Use Cases for Vertical Concatenation

Vertical concatenation is not merely a technical exercise; it solves several critical data preparation challenges in big data environments. One primary application is consolidating time-series data. Imagine a scenario where daily sensor readings or transaction records are stored in separate files or tables optimized for daily ingestion. Before performing monthly or quarterly trend analysis, these daily snapshots must be combined into a single, comprehensive DataFrame. Using the `unionAll` or `union` approach ensures that the historical continuity of the data is preserved while leveraging Spark's distributed architecture for efficient stacking.

Another significant use case involves managing A/B test results or survey data collected from multiple systems or geographical locations. If three different servers collected user interaction data over the same period, each resulting in a DataFrame with identical schema (e.g., user ID, action type, timestamp), vertical concatenation is the fastest way to unify these logs for a global analysis of user behavior. This method bypasses complex joining logic, which is unnecessary since the rows are inherently independent and additive.

Furthermore, in machine learning preprocessing, it is common to concatenate training, validation, and testing datasets that were initially split but need to be recombined temporarily for specific transformations (like feature engineering) before being split again. The key requirement for all these practical applications is maintaining strict schema compatibility. If the schemas differ, the resulting concatenated DataFrame may contain corrupted or misaligned data, necessitating the use of schema-aware methods like `unionByName`, which we will discuss later.

Practical Example: Setting Up the PySpark Environment and DataFrames

To demonstrate the vertical concatenation process effectively, we will establish a controlled PySpark environment and generate three sample DataFrames. These DataFrames will simulate aggregated scoring data for basketball teams, where each DataFrame represents data collected from a different source or time period. We must first initialize a SparkSession, which serves as the entry point to programming Spark with the Dataset and DataFrame API.

In this example, all three DataFrames are designed to have an identical schema: two columns named 'team' (string) and 'points' (integer). This ensures that the subsequent `unionAll` operation executes smoothly without schema conflicts. Defining the data structure explicitly using Python lists and then using `spark.createDataFrame` is the standard procedure for creating local DataFrames

for demonstration or testing purposes within a distributed environment.

Observe the initialization of the SparkSession and the creation of three separate datasets, `df1`, `df2`, and `df3`. Viewing the output of each DataFrame using the `.show()` method confirms their structure and individual contents before combination.

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

# define data for DataFrame 1
data1 = ,
,
]

# define data for DataFrame 2
data2 = ,
,
,
]

# define data for DataFrame 3
data3 = ,
,
,
]

define consistent column names
columns =

# create distributed dataframes using data and column names
df1 = spark.createDataFrame(data1, columns)
df2 = spark.createDataFrame(data2, columns)
df3 = spark.createDataFrame(data3, columns)

# view individual dataframes contents
df1.show()
df2.show()
df3.show()

+------+------+
| team|points|
+------+------+

| Mavs| 18|
| Nets| 33|
|Lakers| 12|
+------+------+

+-------+------+
| team|points|
+-------+------+
| Kings| 15|
| Hawks| 19|
|Wizards| 24|
| Magic| 28|
+-------+------+

+-------+------+
| team|points|
+-------+------+
|Celtics| 25|
| Spurs| 29|
|Rockets| 14|
| Heat| 30|
+-------+------+

Executing the Vertical Concatenation and Reviewing Results

Once the individual DataFrames are prepared and verified, the next step is to apply the functional aggregation mechanism using reduce. This process combines the data structures stored across the distributed cluster nodes into a single logical DataFrame, ready for subsequent processing steps. This step effectively takes the 3 rows from `df1`, the 4 rows from `df2`, and the 4 rows from `df3`, and stacks them together to form a resulting DataFrame of 11 rows.

The use of `DataFrame.unionAll` (or `DataFrame.union`) within the reduce function is key here. It tells Spark to execute the union operation iteratively, minimizing the complexity of the code while maximizing the efficiency of the underlying distributed computation engine. The result, `df_all`, will be a new distributed DataFrame referencing the combined physical data blocks.

After the execution, it is standard practice to inspect the resulting DataFrame using `df_all.show()` to confirm that the concatenation was successful and that all original rows are present and correctly ordered. Note that because `union` methods do not guarantee any specific sorting, the row order might reflect the order in which the DataFrames were listed in `df_list`, but

the internal ordering of the rows within each original DataFrame is typically preserved.

```
from functools import reduce
from pyspark.sql import DataFrame

# specify DataFrames to concatenate
df_list =

# vertically concatenate all DataFrames in list
df_all = reduce(DataFrame.unionAll, df_list)

# view resulting DataFrame
df_all.show()

+-------+------+
| team|points|
+-------+------+
| Mavs| 18|
| Nets| 33|
| Lakers| 12|
| Kings| 15|
| Hawks| 19|
|Wizards| 24|
| Magic| 28|
|Celtics| 25|
| Spurs| 29|
|Rockets| 14|
| Heat| 30|
+-------+------+
```

The resulting DataFrame, `df_all`, successfully contains all 11 records from the three source DataFrames, stacked vertically according to the order defined in `df_list`. This confirms the successful vertical concatenation using the robust reduce pattern, providing a clean and maintainable method for data integration in PySpark.

Advanced Considerations: Schema Mismatches and `unionByName`

While the `unionAll` (or `union`) function is efficient, it imposes a strict requirement: the schema of the input DataFrames must match exactly in terms of column count and column order. If, for example, `df1` had columns `(team, points)` and `df2` had columns `(points, team)`, using the standard `union` would result in misaligned data, treating the 'points' column in `df2` as 'team' data,

and vice versa. This scenario introduces data corruption and invalid analysis results.

To address schema differences where columns exist but are misordered, or where one DataFrame possesses additional columns not found in others, PySpark provides `unionByName`. This function aligns DataFrames based on column names rather than positional order. If a column exists in one DataFrame but not the other, `unionByName` intelligently fills the missing values with `null` in the resulting combined DataFrame, ensuring data integrity across the unified schema.

For complex production pipelines where data sources are not strictly controlled, using `unionByName` is highly recommended for its safety features. If you were to apply the `reduce` pattern to DataFrames with varying schemas, you would simply replace `DataFrame.unionAll` with `DataFrame.unionByName`, thereby utilizing the robustness of name-based alignment across all input DataFrames efficiently. This slight modification ensures maximum resilience against unexpected changes in upstream data schema structure.

Further Learning and Related PySpark Operations

The vertical concatenation illustrated here is one of many essential data manipulation tasks in the distributed computing landscape. Mastering the use of SparkSession, distributed DataFrames, and functional tools like reduce is paramount for effective big data engineering. For those seeking to deepen their expertise, exploring related operations such as horizontal joining (using `join`), aggregating data (using `groupBy`), and restructuring data (using `pivot` or `melt` equivalents) is a logical next step.

A solid grasp of SparkSession management and efficient data loading techniques complements the ability to concatenate DataFrames effectively. High-performance data processing requires minimizing shuffling and optimizing resource allocation, and choosing the correct union method is a key component of this optimization strategy. For example, understanding when to use `union` versus `unionByName` can drastically reduce debugging time related to schema alignment errors.

We have successfully demonstrated the creation and vertical combination of multiple PySpark DataFrames. The new DataFrame named **df_all** contains the data from all three DataFrames concatenated vertically, ready for large-scale analysis. The final note below points towards comprehensive documentation for advanced reference on these core functionalities.

Note: You can find the complete documentation for the PySpark **DataFrame.union** function on the official Apache Spark API documentation page.

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

How to perform joins efficiently in PySpark.

Techniques for handling missing data using PySpark functions.

Using Window functions for complex aggregations.