Learning How to Vertically Concatenate PySpark DataFrames Using `unionAll` and `reduce`

Authored by

Mohammed looti

November 11, 2025

Managing and manipulating large datasets efficiently is the cornerstone of modern data engineering. In the **PySpark** environment, one of the most common requirements is the ability to combine separate data structures--specifically, vertically appending multiple **DataFrames** into a single, cohesive unit. This process, often referred to as **vertical concatenation**, is essential when dealing with datasets that have the same schema but are partitioned across different files or processing stages.

The most concise and scalable approach to vertically concatenate an arbitrary number of **DataFrames** in **PySpark** involves using the powerful combination of Python's built-in **reduce function** from the `functools` module and the PySpark `DataFrame.unionAll` method. Below is the standard syntax utilized for this operation:

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)

This particular example uses the **reduce function** along with the **unionAll** function to iteratively combine the DataFrames named **df1**, **df2**, and **df3** into one final DataFrame called **df_all**. This methodology is highly efficient for combining numerous datasets in a distributed computing environment.

Understanding Vertical Concatenation in Big Data Environments

In the context of distributed computing frameworks like Apache Spark, combining datasets requires careful consideration to maintain efficiency and integrity across the cluster. **Vertical concatenation**, often analogous to SQL's `UNION ALL` or pandas' `pd.concat(..., axis=0)`, means stacking one DataFrame on top of another. This operation assumes that all participating **DataFrames** share an identical schema, meaning they must have the same number of columns, the same column names, and corresponding data types in the same order.

While PySpark offers two primary methods for vertical merging--`union` and **unionAll**--the choice between them is critical for performance. The `unionAll` method is typically faster because it does not incur the cost of distinct row calculation. Unlike `union`, which automatically removes duplicate rows, `unionAll` simply appends the rows without performing resource-intensive deduplication checks, making it the preferred method for scenarios where the input data sources are known to be disjoint or where duplicate rows are intentionally preserved.

When dealing with two or three DataFrames, explicitly chaining `df1.unionAll(df2).unionAll(df3)` is feasible. However, when the number of DataFrames grows, especially if they are generated dynamically, a programmatic solution is necessary to avoid tedious manual chaining. This is precisely where the Python **reduce function** proves invaluable for scalable data aggregation tasks within **PySpark**.

The Core Mechanism: Leveraging reduce and unionAll

The successful execution of the concatenation script hinges on understanding how the **reduce function** works in conjunction with the **unionAll** method. The `reduce` function, found in Python's `functools` module, is designed to apply a specified function cumulatively to the items of an iterable (like a list), reducing the iterable to a single resultant value. In our case, the iterable is `df_list`, which contains all the DataFrames to be combined.

The operation proceeds iteratively: `reduce` takes the first two elements (`df1` and `df2`) and applies the specified function (`DataFrame.unionAll`) to them. The result of this first operation is then passed as the first argument, along with the next element (`df3`), to the function again. This process continues until every DataFrame in the list has been processed. The final output is a single, consolidated DataFrame, `df_all`, containing all the rows from the input list.

Using `reduce(DataFrame.unionAll, df_list)` is a highly idiomatic and efficient pattern in **PySpark** for large-scale merging. It abstracts away the complexity of managing a variable number of merging steps, providing a clean, functional programming approach to what would otherwise require an explicit loop or a long chain of method calls. This technique is mandatory when you need to combine more than two DataFrames dynamically, ensuring the code remains robust regardless of the size of the input list.

Practical Demonstration: Creating Sample PySpark DataFrames

To illustrate the process of **vertical concatenation**, let us work with a realistic scenario involving basketball statistics. Suppose we have collected scoring data from three different sources, each stored as a separate **DataFrame**. These DataFrames, `df1`, `df2`, and `df3`, all share a common schema: `team` (string) and `points` (integer). Before we can combine them, we must initialize a Spark Session and define our sample data.

The following code snippet sets up our environment, defines the sample row data, specifies the column names, and uses `spark.createDataFrame` to instantiate the three independent DataFrames. Notice that each DataFrame is printed using `.show()` to confirm its structure and content prior to the merge operation. This setup confirms that we are dealing with standard PySpark data structures that are ready for aggregation.

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

#define data
data1 = ,
,
]

data2 = ,
,
,
]

data3 = ,
,
,
]

#define column names
columns =

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

#view dataframes
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 Operation

With our DataFrames initialized, the next step is to pool these resources into a single list and apply the aggregation logic. We import the necessary components--`reduce` from `functools` and the `DataFrame` type from `pyspark.sql`--to execute the concatenation. The list `df_list` is defined to hold `df1`, `df2`, and `df3`, preparing them for sequential merging.

As described previously, the **reduce function** iterates through `df_list`, applying `DataFrame.unionAll` sequentially. This results in `df_all`, which contains all 11 rows combined from the original three data sources. This single, clean operation replaces multiple nested `unionAll` calls, offering superior readability and maintainability, especially in larger scripting environments and complex **PySpark** workflows.

The resulting DataFrame confirms that all rows have been successfully appended without altering the column structure. This is the desired outcome of **vertical concatenation**, producing a master data set ready for subsequent analytical transformations or persistence.

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

Summary and Further Considerations

The new DataFrame named **df_all** successfully contains the data from all three DataFrames concatenated vertically. It is critical to reiterate that this technique is highly optimized for scenarios where data volume is large and the schema is consistent across all input sources. Using `reduce` with **unionAll** ensures that the operation is executed efficiently across the distributed cluster managed by Apache Spark, minimizing unnecessary resource usage by skipping the duplicate checking phase.

When selecting the appropriate merging technique in **PySpark**, always consider the schema requirements. While `unionAll` requires identical column order, the newer `unionByName` function offers flexibility by matching columns based on name rather than position, although using it might introduce slightly more overhead. For maximum performance in straightforward vertical stacking where schema order is guaranteed, the `reduce(DataFrame.unionAll, list)` pattern remains the community standard due to its speed and elegance.

Mastering these fundamental aggregation techniques is essential for any data professional working with large-scale distributed data processing. For those interested in exploring further data manipulation capabilities within PySpark, specifically concerning column-wise combinations (horizontal concatenation), or more advanced data joining based on key values, the official documentation provides extensive resources.

Note: You can find the complete documentation for the PySpark **unionAll** function, which is the underlying method powering the vertical merge operation, by clicking the embedded link.

Additional Resources for PySpark Data Wrangling

To deepen your expertise in PySpark data manipulation, consider exploring tutorials on related common tasks:

Understanding and implementing horizontal concatenation (joining DataFrames).

Handling schema mismatches during union operations.

Using Window functions for complex analytical tasks in PySpark.

Optimizing Spark performance through caching and partitioning strategies.