How can new rows be added to a PySpark DataFrame?

Authored by

stats writer

February 9, 2026

Mastering Row Addition in PySpark DataFrames: A Comprehensive Guide

Introduction to Data Merging in PySpark

The ability to dynamically update and modify datasets is fundamental in modern data processing workflows. When utilizing PySpark for large-scale data manipulation, appending new records to an existing DataFrame is a common requirement, particularly during Extract, Transform, Load (ETL) processes or incremental data loading. The primary and most efficient method for achieving this in PySpark is by employing the powerful `union()` transformation.

This function is specifically designed to combine two DataFrames, effectively appending the rows of the source DataFrame (containing the new data) to the existing target DataFrame. It is essential to understand that `union()` operates under a strict principle of schema compatibility, requiring both DataFrames to have an identical structure, including the sequence of columns, their names, and corresponding data types.

The efficiency of the `union()` function stems from its nature as a high-performance, distributed operation inherent to the Apache Spark framework. Unlike traditional database operations that might involve complex joins or inserts, `union()` simply concatenates the underlying RDDs (Resilient Distributed Datasets) of the two DataFrames. The standard syntax for invoking this operation is straightforward: `df1.union(df2)`, where `df1` is the initial DataFrame and `df2` contains the records to be appended.

Prerequisites for Successful DataFrame Union

Before attempting to add new rows using `union()`, data engineers must ensure that the new data is formatted correctly as a DataFrame. PySpark operations demand that the source and target structures align perfectly. This means that if the original DataFrame has three columns named `id` (Integer), `name` (String), and `salary` (Double), the DataFrame containing the new rows must also have three columns with the exact same names and types, presented in the same order.

Failure to meet these strict schema requirements often leads to silent errors or, more commonly, exceptions where Spark indicates a mismatch in column count or type. In situations where the new data source has non-matching column names, developers must utilize PySpark functions like `withColumnRenamed()` or `select()` on the new data source (`df2`) prior to executing the `union()` operation, ensuring the necessary structural harmony.

Furthermore, it is crucial to understand the distinction between `union()` and `unionByName()`. While standard `union()` relies on column order for alignment, `unionByName()` allows column

alignment based on matching names, regardless of their position. However, if the goal is strictly to append rows where the schema is known to be identical, `union()` is the standard, high-performance choice.

Setting Up the Initial PySpark Environment and DataFrame

To illustrate the application of the `union()` method, we first need a working SparkSession and an initial DataFrame. The SparkSession acts as the entry point for utilizing Spark functionality, and our base DataFrame will represent existing data that we intend to augment.

We will define a sample dataset containing basketball player statistics, including their team, position, and points scored. This foundation allows us to execute and visualize the row addition processes effectively, confirming that the new data is successfully merged into the existing structure.

You can use the following methods to add new rows to a PySpark DataFrame:

Method 1: Adding a Single New Row to the DataFrame

```
#define new row to add with values 'C', 'Guard' and 14
new_row = spark.createDataFrame(, columns)
#add new row to DataFrame
df_new = df.union(new_row)
```

Method 2: Adding Multiple New Rows to the DataFrame

```
#define multiple new rows to add
new_rows = spark.createDataFrame(, columns)

#add new rows to DataFrame
df_new = df.union(new_rows)
```

Initializing the Example DataFrame

The following code snippet demonstrates the creation of the base DataFrame, `df`, which will be used for all subsequent demonstrations. This foundational step ensures a consistent environment for testing the row addition methods. We utilize `SparkSession.builder.getOrCreate()` to instantiate our Spark context.

```
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()

+----+--------+------+
|team|position|points|
+----+--------+------+
| A| Guard| 11|
| A| Guard| 8|
| A| Forward| 22|
| A| Forward| 22|
| B| Guard| 14|
| B| Guard| 14|
| B| Forward| 13|
| B| Forward| 7|
+----+--------+------+
```


As observed in the output, the initial DataFrame `df` contains eight records spread across two teams (A and B). The columns are clearly defined as `team`, `position`, and `points`. Any new rows we add must conform precisely to this three-column structure and their respective data types (String, String, Integer).

Example 1: Adding a Single Row to the DataFrame

One common scenario involves appending just a single record, perhaps representing a late entry

or a calculation result, to an existing dataset. Although it may seem inefficient for distributed systems, PySpark handles this gracefully by first converting the single tuple of data into a small, temporary DataFrame, which is then combined using `union()`.

In this example, we define `new_row` corresponding to a player on 'C' team, playing 'Guard', and scoring '14' points. It is crucial that when creating this temporary DataFrame using `spark.createDataFrame()`, we pass the existing `columns` list to ensure the schema of the new row exactly matches that of the base DataFrame, `df`.

We use the `union()` function to merge the two structures, storing the result in `df_new`. This process demonstrates the fundamental mechanism of row addition: transforming raw data into a compatible DataFrame format and then concatenating it.

#define new row to add
new_row = spark.createDataFrame(, columns)

#add new row to DataFrame
df_new = df.union(new_row)

#view updated DataFrame
df_new.show()

+----+--------+------+
|team|position|points|
+----+--------+------+
A	Guard	11
A	Guard	8
A	Forward	22
A	Forward	22
B	Guard	14
B	Guard	14
B	Forward	13
B	Forward	7
C	Guard	14
+----+--------+------+

Upon reviewing the output, we notice that exactly one new row has been successfully added to the end of the combined DataFrame, `df_new`, featuring the values `C`, `Guard`, and `14`, just as we specified. The total row count increases from 8 to 9.

Example 2: Adding Multiple Rows Efficiently

The true power of `union()` is realized when appending large batches of data, which is typical in production PySpark environments. Instead of iterating and appending records one by one (a highly inefficient practice), we define all new rows simultaneously within a single list of tuples and convert this entire collection into a single source DataFrame.

In this second example, we introduce three new records, including two players from team 'C' and one from a new team 'D'. This process showcases how scalable the `union()` transformation is, allowing for the rapid incorporation of substantial data volumes without taxing the underlying cluster resources excessively.

The `new_rows` DataFrame contains three records, all structured according to the predefined `columns` schema. We then apply `df.union(new_rows)` to generate the final, consolidated dataset.

#define multiple new rows to add
new_rows = spark.createDataFrame(, columns)

#add new rows to DataFrame
df_new = df.union(new_rows)

#view updated DataFrame
df_new.show()

+----+--------+------+
|team|position|points|
+----+--------+------+
A	Guard	11
A	Guard	8
A	Forward	22
A	Forward	22
B	Guard	14
B	Guard	14
B	Forward	13
B	Forward	7
C	Guard	14
C	Forward	32
D	Forward	21
+----+--------+------+

The result clearly demonstrates that three new records have been successfully appended to the

end of the base DataFrame. The combined DataFrame now holds a total of eleven rows, seamlessly integrating the new data points from teams 'C' and 'D' alongside the original records from teams 'A' and 'B'.

Important Considerations and Alternatives

While the `union()` function is the standard method for appending rows, developers must be mindful of its inherent characteristics. The primary characteristic is that `union()` does not inherently handle deduplication; if the new rows (`df2`) contain exact duplicates of rows already present in `df1`, both copies will persist in the resulting DataFrame.

If the requirement is to append new rows while simultaneously guaranteeing uniqueness across the entire dataset, a subsequent call to `df_new.distinct()` would be necessary. However, applying `distinct()` incurs a significant performance overhead because it triggers a shuffle operation across the cluster to compare all records globally. Therefore, `union()` is typically preferred when the new data source is known to contain unique or necessary duplicates.

Another method, `unionByName()`, should be used if the column order of the input DataFrames is not guaranteed to be identical, but their schemas are otherwise compatible. `unionByName()` aligns columns using their names, inserting `null` values for any columns present in one DataFrame but absent in the other. This flexibility comes at a slightly higher computational cost compared to the strict, order-based `union()`.

Conclusion: Efficient Data Augmentation in PySpark

In summary, the most effective and idiomatic way to add new rows to a PySpark DataFrame is through the use of the `union()` function. This transformation allows for high-throughput data merging by combining two DataFrames into a new, consolidated structure. Whether you are adding a single late record or integrating millions of new rows from an incremental load, the underlying principle remains the same: define the new data as a schema-compatible DataFrame and concatenate it using the `union()` operator.

Always prioritize strict schema adherence--matching column names, data types, and order--to ensure smooth and error-free execution of the union operation. By mastering the application of `spark.createDataFrame()` to construct new records and utilizing `df.union()`, data professionals can efficiently manage dynamic datasets within the Apache Spark ecosystem.

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