Add New Rows to PySpark DataFrame (With Examples)

Authored by

Mohammed looti

November 11, 2025

Introduction: Appending Data in a Distributed Environment

Adding new records to a data structure is a fundamental requirement in data manipulation. However, when working within the Apache Spark ecosystem, specifically using Python via PySpark DataFrame objects, this process differs significantly from standard Pandas or SQL operations. Since Spark is designed for distributed computing, operations that modify data often involve generating a completely new data structure rather than modifying the existing one in place. This guide will thoroughly explore the efficient and standard methodologies for seamlessly integrating one or multiple new rows into an existing PySpark DataFrame, focusing primarily on the robust `union` transformation. We will break down the process step-by-step, ensuring clarity for developers and data engineers operating in big data environments.

The primary challenge when appending data in Spark stems from its core philosophy of immutability. A PySpark DataFrame, once created, cannot be altered directly. Any operation that appears to modify the DataFrame, such as adding a column, filtering rows, or in this case, adding new rows, actually results in a new DataFrame being created, reflecting the original data plus the new transformation logic. Therefore, to add new rows, we must first construct the new data as a mini-DataFrame and then combine it with the existing structure using a suitable transformation. The techniques detailed below leverage this immutable architecture to guarantee data integrity and maintain the parallel processing capabilities that make Spark so powerful for handling vast datasets.

Prerequisites: Setting Up the Spark Environment and Initial Data

Before we demonstrate the methods for adding new rows, we must ensure our environment is correctly configured and establish a sample DataFrame for practical demonstration. All PySpark operations require an active SparkSession, which acts as the entry point to communicate with the Spark cluster. By defining a clear, simple dataset, we can accurately observe the results of our row addition operations and verify that the `union` transformation is working as intended in a distributed context. This initial setup is crucial for reproducible examples.

The following code block defines and initializes the necessary components: the SparkSession, the raw data, the column schema, and finally, the initial PySpark DataFrame named `df`. Notice how the structure involves lists of lists, where each inner list represents one row of data, matching the defined column names: `team`, `position`, and `points`. Once this DataFrame is created, we use `df.show()` to display the contents, confirming the baseline data structure before any modifications occur.

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

Method 1: Appending a Single Row Using the `union` Transformation

The most idiomatic and reliable way to append rows in Apache Spark is by utilizing the `union` transformation. This function concatenates two or more DataFrames vertically, effectively stacking the rows of the second DataFrame onto the first. For this operation to succeed, it is absolutely critical that the schemas of the two DataFrames--the existing one (`df`) and the new row(s) container--must align perfectly. This means they must have the same number of columns, the same column names, and crucially, the same data types in the corresponding positions. To add a

single row, we first define the data for that row and then immediately convert it into a valid PySpark DataFrame using the `spark.createDataFrame` method, ensuring we pass the original `columns` definition.

The process starts by defining the singular row we wish to introduce. This row, represented as a list or tuple of values, is wrapped in a list (to denote the collection of rows) and passed to `spark.createDataFrame`. This yields the temporary DataFrame, `new_row`. Subsequently, the `union` function is called on the original DataFrame (`df`), accepting `new_row` as its argument. The result is a new, immutable DataFrame, `df_new`, which contains all the records from `df` followed by the single new record. This method is highly transparent and ensures that Spark's optimized execution engine handles the merging of the underlying RDD partitions efficiently across the cluster.

Example 1: Adding One New Row to DataFrame

We will now implement the steps described above to add a single new record representing a player from 'Team C' with 14 points. Note how the creation of the `new_row` DataFrame explicitly uses the predefined `columns` list to guarantee schema compatibility with the existing `df`.

#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)

Executing the `union` operation and viewing the result confirms that the new data has been successfully appended to the end of the data structure. The new row `('C', 'Guard', 14)` now exists within the resulting DataFrame `df_new`, exactly as specified.

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

We can clearly observe that one new row has been added to the end of the DataFrame, carrying the specified values: **C**, **Guard**, and **14**. This method is straightforward and highly effective for small additions.

Method 2: Appending Multiple Rows Efficiently

The efficiency of union truly shines when handling large batches of new data. Instead of performing the `union` operation iteratively for dozens or hundreds of single rows--an approach that would lead to significant performance overhead due to repeated execution plan generation--it is far more efficient to package all the new data into a single, comprehensive batch DataFrame. This approach minimizes the number of transformations Spark must execute and optimizes the underlying data movement across the cluster nodes. We define all the records in one large list of tuples or lists before creating the temporary PySpark DataFrame.

To append multiple rows, the fundamental steps remain identical to Method 1, emphasizing the need for schema alignment. We define a list containing all the new row data points. For instance, if we need to add three new records, the list passed to `spark.createDataFrame` will contain three inner tuples. This results in a single `new_rows` DataFrame that already contains the desired batch of data. Subsequently, calling `df.union(new_rows)` executes a single, atomic transformation operation, which is far superior in terms of performance and resource utilization compared to looping through single row additions. This is the recommended practice when integrating external data batches into an existing PySpark DataFrame.

Example 2: Adding Multiple New Rows to DataFrame

In this practical demonstration, we define three new records simultaneously: two for 'Team C' and one for 'Team D'. These are packaged together into a single structure and then combined with the original DataFrame `df` using the `union` operation.

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

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

Upon viewing the resulting DataFrame, we can confirm the successful addition of all three new records. They are appended sequentially to the end of the original data. This demonstrates the efficiency and scalability of using `union` for batch data integration, making it a cornerstone technique in Apache Spark data pipelines.

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

We can clearly see that three new rows have been successfully integrated into the resulting DataFrame `df_new`. This confirms the utility of packaging multiple records into a single temporary DataFrame before executing the `union` transformation, which is crucial for maintaining performance in a distributed environment.

Performance Considerations: `union` vs. `unionAll`

It is essential to distinguish between `union` and its older counterpart, `unionAll`, especially when discussing performance in earlier versions of Spark. In modern versions of Apache Spark (Spark 2.0 and later), the method `union` (specifically `pyspark.sql.DataFrame.union`) is the preferred method for concatenating two DataFrames that share the exact same schema. While the older `unionAll` is technically still available, `union` is generally optimized and ensures compatibility. Importantly, neither `union` nor `unionAll` performs distinct row elimination; they simply append all rows from the second DataFrame to the first, including any duplicates that might exist across both datasets. If duplicate elimination is required after combining the data, an explicit `.distinct()` transformation must be applied to the resulting DataFrame `df_new`.

For situations where the schemas of the two DataFrames are similar but not perfectly identical--for example, if column order is different, but the column names are the same--developers should use `unionByName`. The `unionByName` transformation aligns the columns based on their names rather than their positional index, offering greater flexibility and robustness when merging data from disparate sources. However, for the simple task of appending newly created rows where the schema is explicitly controlled during the creation step (as demonstrated in our examples using `spark.createDataFrame`), the standard `union` method remains the cleanest and most performant choice, provided the schema alignment is strictly maintained. Always prioritize batching new data to minimize transformation overhead.

Summary and Best Practices for Data Addition

The overarching principle governing row addition in PySpark DataFrames is the concept of immutability. Since DataFrames cannot be modified in place, all append operations must generate a brand-new DataFrame. We achieve this by creating a temporary DataFrame containing the new records and merging it using the `union` transformation. This transformation ensures that the resulting dataset is distributed and optimized for subsequent processing stages within the cluster.

Key takeaways for successful and efficient row addition include:

Schema Alignment: Always ensure the temporary DataFrame containing the new rows has an identical schema (column names, order, and data types) to the target DataFrame.
Batching: For optimal performance, always define multiple new rows within a single list and create one batch DataFrame using `spark.createDataFrame`, rather than performing multiple sequential `union` operations for single rows.
Choosing the Right Union: Use `union` when schemas are identical and order matters. Consider `unionByName` if schemas are identical but the column order might vary between the two DataFrames being merged.

These practices ensure that data manipulation tasks in Apache Spark remain highly efficient, scalable, and compliant with the framework's distributed architecture principles.

Additional Resources

For those looking to deepen their understanding of PySpark DataFrame manipulation and other common data operations in a distributed environment, the following topics are highly recommended:

Filtering and Selecting Data in PySpark
Joining DataFrames (Inner, Outer, Left, Right Joins)
Understanding Lazy Evaluation and Execution Plans in Spark
Optimizing Data Partitioning for Performance