Add New Rows to PySpark DataFrame (With Examples)

Authored by

stats writer

November 16, 2025

Introduction: Understanding Data Modification in PySpark

When working with large-scale data processing using PySpark, a common requirement is the ability to dynamically add new records or rows to an existing DataFrame. Unlike traditional relational database systems or Pandas, PySpark DataFrames are immutable. This means that you cannot directly modify them in place; instead, operations generate a new DataFrame reflecting the desired changes. The most efficient and standard way to append data in PySpark is by utilizing the powerful transformation functions available, specifically the **`union` function**.

The concept of immutability in distributed computing environments like Apache Spark ensures consistency and fault tolerance. When you "add rows," you are essentially concatenating two separate DataFrames: the original dataset and a newly created, small DataFrame containing the records you wish to append. This approach maintains the integrity of the original data while allowing for complex data manipulation workflows. We will explore two primary methods for this operation: adding a single row and adding multiple rows in a batch process, both relying on the creation of temporary DataFrames.

To successfully append new rows, it is critical that the structure (schema) of the new records precisely matches the schema of the existing DataFrame, including column names and data types. Failure to match the schema will result in runtime errors when the **`union` transformation** is attempted. Below, we outline the foundational techniques used to define and incorporate new data points into your existing dataset efficiently.

Core Technique: Utilizing the `union` Transformation for Row Addition

The primary mechanism for combining data vertically in PySpark is the union function. This function merges the rows of two DataFrames, assuming they have an identical number of columns and compatible data types based on column position. This is a fundamental concept in distributed SQL operations.

The challenge when appending new rows is that they typically originate from a small list of values, not an existing large file or table. Therefore, the first step is always to convert these new values into a valid DataFrame using **`spark.createDataFrame`**. This temporary DataFrame, whether it holds one row or hundreds, is then suitable for the **`union` operation**.

Here is a conceptual overview of the two methods we will implement, demonstrating how new data is formatted into a temporary DataFrame before being combined with the existing one.

Method 1: Add One New Row to DataFrame

 # Define a new row as a list of tuples, ensuring schema match (e.g., 'C', 'Guard', 14)

new_row = spark.createDataFrame(, columns)

# Use the union function to append the new row and create a resulting DataFrame
df_new = df.union(new_row)

Method 2: Add Multiple New Rows to DataFrame (Batch Append)

 # Define multiple new rows within a list of tuples
new_rows = spark.createDataFrame(, columns)

# Combine the new rows DataFrame with the original using union
df_new = df.union(new_rows)

Prerequisites: Setting up the PySpark Environment and Initial DataFrame

Before executing the row addition techniques, we must establish a working PySpark session and define the base DataFrame (`df`) that we intend to modify. The initialization process involves importing necessary modules and creating an instance of **`SparkSession`**, which is the entry point for all Spark functionality in the application.

The initial data we use represents basketball statistics, defined by three key columns: `team`, `position`, and `points`. When manually defining data within a Python script, it is standard practice to structure the data as a list of lists or a list of tuples, where each inner element represents a single record. Defining the column names explicitly ensures that the created DataFrame (`df`) has the correct schema required for subsequent operations.

The following setup code initializes the environment and displays the resulting base DataFrame. This `df` will be the recipient of our new rows in the subsequent examples. Note the use of **`SparkSession.builder.getOrCreate()`** to manage the Spark context efficiently.

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

# Define the data records for the initial DataFrame
data = ,
,
,
,
,
,
,

```
]

# Define column names (Crucial for schema matching)
columns =

# Create the base DataFrame using spark.createDataFrame
df = spark.createDataFrame(data, columns)

# Display the initial DataFrame structure and content
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: Adding a Single Row to the PySpark DataFrame (Detailed Explanation)

The simplest form of data appending involves adding just one new record. While this might seem trivial, it perfectly illustrates the process required for any row addition in PySpark. Since Spark operates fundamentally on distributed data collections, a single row must still be encapsulated within a valid DataFrame structure before it can be merged with the original data.

To achieve this, we use **`spark.createDataFrame`**, providing a list containing a single tuple. This tuple holds the specific values for the new row, and we reuse the `columns` list defined earlier to enforce the correct schema. This step transforms the static Python data element into a Spark distributed dataset, which is a prerequisite for the **`union` transformation**. It is essential that the data types within the tuple (e.g., string for team, integer for points) align with the inferred or defined schema of the original `df`.

Once the single-row DataFrame (`new_row`) is successfully created, the **`union` function** is called on

the original DataFrame (`df`), accepting the `new_row` DataFrame as an argument. The result is a completely new DataFrame (`df_new`) that contains all the records from `df` followed by the single record from `new_row`. This approach is highly reliable and ensures that the immutable nature of the original DataFrame is preserved.

Implementation Example 1: Executing the Single Row Addition

In this practical example, we demonstrate the exact syntax used to define and append a new record representing a player on 'Team C' with a 'Guard' position and '14' points. The process confirms that the new row is successfully integrated into the existing structure, adding to the total count of records.

We begin by defining the structure for the new record. We then execute the **`df.union(new_row)`** command. Since `union` operates based on the positional order of columns, maintaining the correct sequence ('team', 'position', 'points') in the new data tuple is paramount to avoid data misalignment.

The output clearly shows the appended row at the bottom of the resulting DataFrame, confirming the success of the operation. This method is ideal when integrating sporadic, real-time data points or when testing data scenarios.

Example 1: Add One New Row to DataFrame

We use the following syntax to create a new temporary DataFrame and append one new row to the end of the existing dataset using the **`union` function**:

 # Define the new row to append, ensuring it matches the 'columns' schema
new_row = spark.createDataFrame(, columns)

# Append the new row to the original DataFrame
df_new = df.union(new_row)

# View the updated DataFrame, which now includes the new record
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|
+----+--------+------+
```

Notice that one new row has been successfully added to the end of the DataFrame with the values **C**, **Guard**, and **14**, precisely as defined in our `new_row` construction. The original eight records are now supplemented by this ninth record.

Method 2: Appending Multiple Rows Efficiently (Detailed Explanation)

While adding a single row is useful for specific cases, real-world data engineering often requires appending large batches of new records simultaneously. Fortunately, the **`union` approach** scales seamlessly for multiple rows. Instead of creating a DataFrame from a list of one tuple, we simply provide **`spark.createDataFrame`** with a list containing multiple tuples, where each tuple represents a record.

This batch appending method is far more efficient than iterating through records and performing individual union operations, which would incur significant overhead due to the creation of many small, intermediate DataFrames. By defining all new records within a single list, Spark can process the data creation and subsequent union transformation in a highly optimized manner across the cluster.

For robust data pipelines, ensuring the structural consistency of the incoming batch data is non-negotiable. If any tuple in the list defining `new_rows` is missing a value or has an incorrect data type compared to the original DataFrame schema, the **`spark.createDataFrame`** or the subsequent **`union` operation** will fail. This emphasizes the need for strong Data Model governance during the data preparation phase.

Implementation Example 2: Batch Appending Multiple Records

In this example, we append three distinct new records simultaneously. The records introduce a new team ('D') and additional data for 'Team C', showcasing how the union operation seamlessly handles heterogeneity in the new data, provided the schema remains consistent.

The definition of `new_rows` now includes three tuples: `('C', 'Guard', 14)`, `('C', 'Forward', 32)`, and `('D', 'Forward', 21)`. This single variable is then passed to the **`union` function**,

generating a final DataFrame that is five records longer than the original.

Observing the output confirms that all three specified records are appended successfully to the end of the existing data set. This technique is highly practical for scenarios involving daily log ingestion or integrating temporary lookup data into a primary dataset.

Example 2: Add Multiple New Rows to DataFrame

We use the following syntax to define a list of multiple new records, create a temporary DataFrame from that list, and append all three new rows using the **`union` function**:

 # Define multiple new rows using a list of tuples
new_rows = spark.createDataFrame(, columns)

# Append the multiple new rows to the base DataFrame
df_new = df.union(new_rows)

# View the resulting 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|
+----+--------+------+
```

Notice that three new rows have been added to the end of the resulting DataFrame. The initial 8 rows plus the 3 new rows yield a total of 11 records in `df_new`. This validates the efficiency of the batch append approach for handling multiple records.

Deep Dive into `union()` vs. `unionByName()`

While we utilized the standard **`union()` function** throughout these examples, it is crucial to understand its counterpart, **`unionByName()`**. The standard `union()` concatenates two DataFrames based strictly on the positional order of their columns. This means that if the first column of `df1` is 'name' and the first column of `df2` is 'age', the resulting column will contain a mix of names and ages, leading to corrupted data, even if the data types match.

In contrast, **`unionByName()`** is designed to combine DataFrames by matching column names, regardless of their position. This provides far greater flexibility and robustness, especially when dealing with data sources where the column order might vary or evolve over time. If a column exists in one DataFrame but not the other, **`unionByName()`** will typically fill the missing values with nulls for the records originating from the DataFrame lacking that column.

For the specific task of adding rows created manually via **`spark.createDataFrame`**, using standard **`union()`** is often acceptable because we explicitly control the column order of the small temporary DataFrame using the `columns` list. However, if the source of the new records were another external DataFrame whose schema order could not be guaranteed, **`unionByName()`** would be the safer and recommended practice.

Performance Considerations and Best Practices

While appending rows using the union function is straightforward, understanding the underlying performance implications is key to writing scalable Spark code. Every time a **`union` transformation** is executed, Spark builds a new Directed Acyclic Graph (DAG) and lineage for the resulting DataFrame. Continuous, iterative appends of single rows can lead to a very deep lineage, potentially degrading performance due to increased overhead in tracking the execution plan.

The core principle is to minimize transformations and maximize batch operations. If you have many individual rows to add, it is a much better practice to collect all of them into a single Python list and execute **`spark.createDataFrame`** once, followed by a single **`union` call** (Method 2), rather than performing multiple single-row union operations (Method 1). Batching reduces the number of stages Spark must execute and optimizes the shuffle necessary for the union.

Remember that the union operation, like most Spark transformations, is lazy. The actual data merging does not occur until an action (like **`show()`** or **`write()`**) is called. Furthermore, the union function does not implicitly handle duplicate records. If duplicate rows are present in both the original and the appended DataFrames, they will both appear in the final result. If deduplication is required, an explicit call to **`df_new.distinct()`** must be made immediately following the union.

The complete documentation for the PySpark **`union` function** provides extensive details on its

usage, parameter specifications, and behavior concerning schema evolution.