How to Combine and Deduplicate DataFrames Using PySpark Union

Authored by

stats writer

February 4, 2026

RECOMMENDED CITATION

stats writer (2026). *How to Combine and Deduplicate DataFrames Using PySpark Union*. PSYCHOLOGICAL SCALES. Retrieved from https://scales.arabpsychology.com/?p=129369

ARABPSYCHOLOGY.COM

Optimizing Data Integration: Performing Union and Returning Distinct Rows in PySpark

PySpark stands as a foundational tool for handling immense datasets within a distributed computing environment. A fundamental requirement in complex data analysis pipelines is the ability to merge or combine multiple datasets efficiently. This operation is typically handled by the union transformation. While merging datasets is straightforward, ensuring data quality often requires the removal of redundancy. To guarantee that only unique records persist after the merge, the subsequent use of the distinct function is essential. By meticulously combining the power of the **union** and **distinct** functions, data engineers can perform robust data integration, yielding clean, optimized DataFrames ready for further analytical processing. This guide explores the mechanisms and practical application of this combined operation, crucial for maintaining data integrity in large-scale analysis.

Understanding the PySpark Union Operation

The **union** operation in PySpark is designed to append the rows of one DataFrame to another. It is crucial to note that for the union to execute successfully, both source DataFrames must possess the same number of columns and compatible data types for those columns. Unlike SQL's UNION ALL, the PySpark `.union()` method is generally preferred for its compatibility and ease of use when dealing with schema matching, though it inherently behaves similarly to UNION ALL in SQL by retaining all rows, including duplicates.

When utilizing the standard `.union()` function, if the input DataFrames contain identical rows--meaning all column values match across both DataFrames--these rows will appear multiple times in the resulting merged DataFrame. For workflows where redundancy is acceptable or expected, the simple **union** operation suffices. However, in most real-world scenarios involving master data consolidation, merging incremental logs, or combining results from parallel processing, duplicate records must be eliminated to ensure the integrity and accuracy of subsequent computations. The simple union method alone does not guarantee a clean dataset.

The Role of the Distinct Transformation

The distinct transformation is the primary mechanism used within PySpark to identify and remove duplicate rows across all columns in a DataFrame. When chained immediately after a **union** operation, the distinct function forces PySpark to analyze the combined result set and return only the truly unique records. This process effectively converts the simple concatenation into a true set-based union, mirroring the behavior of the SQL UNION operator.

ARABPSYCHOLOGY.COM

It is important to understand the performance implications of this combination. Both the union and **distinct** operations are categorized as wide transformations, meaning they require data shuffling across the cluster partitions. While **union** generally involves less computational overhead (primarily data movement), the **distinct** operation requires a full comparison, grouping, and hash calculation across all rows and columns to precisely identify and discard duplicates, making it significantly resource-intensive. Therefore, this operation should be used judiciously, prioritizing performance by ensuring deduplication is only executed when strictly necessary for the analytical outcome.

Core Syntax for Union and Distinct

To successfully execute the combined operation of merging two DataFrames and retaining only the unique rows, the methods must be chained together sequentially. The union function is applied first to combine the source DataFrames, and the result of this operation--the intermediate, potentially duplicated DataFrame--is immediately passed to the distinct function for deduplication.

You can use the following concise syntax to perform a union on two PySpark DataFrames and return only distinct rows, storing the resultant, clean dataset into a new DataFrame variable:

df_union_distinct = df1.union(df2).distinct()

This particular example performs a merge using the **union** method between the PySpark DataFrames named **df1** and **df2**. By immediately appending the `.distinct()` call, the resulting collection of rows is processed to eliminate any identical records that may have originated from either **df1**, **df2**, or existed in both. This streamlined approach ensures the final output DataFrame contains only unique entries, fulfilling the requirements for many downstream analytical applications.

Practical Example: Setting Up Our DataFrames

The following comprehensive example illustrates how to implement this combined operation in a real-world context. We will first establish two separate PySpark DataFrames, **df1** and **df2**, some of whose rows will overlap, allowing us to clearly observe the effects of both the simple union and the union with the distinct constraint. This setup mirrors a common scenario where data arrives from two different sources and requires merging.

Suppose we define our first DataFrame, named **df1**, which represents sales data for various teams:

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

ARABPSYCHOLOGY.COM

#define data
data1 = ,
,
,
,
]

#define column names
columns1 =

#create DataFrame
df1 = spark.createDataFrame(data1, columns1)

#view DataFrame
df1.show()

+----+----------+------+
|team|conference|points|
+----+----------+------+
A	East	11
B	East	8
C	East	31
D	West	16
E	West	6
+----+----------+------+

Next, we define a second DataFrame, **df2**. Critically, **df2** contains two rows (Team A and Team B) that are exact duplicates of rows found in **df1**, alongside two unique rows (Team G and Team H). This overlap is intentional and will be the focus of our deduplication effort later in the process.

#define data
data2 = ,
,
,
]

#define column names
columns2 =

#create DataFrame
df2 = spark.createDataFrame(data2, columns2)

ARABPSYCHOLOGY.COM

```
#view DataFrame
df2.show()

+----+----------+------+
|team|conference|points|
+----+----------+------+
| A| East| 11|
| B| East| 8|
| G| East| 31|
| H| West| 16|
+----+----------+------+
```

Demonstrating Union Without Distinct (Identifying Duplicates)

Before applying the distinct filter, let us first examine the behavior of a simple **union** operation. This step clearly demonstrates the need for deduplication when attempting to achieve true set union semantics, as the base `.union()` function is designed solely for concatenation, not cleaning.

We use the following code snippet to merge **df1** and **df2** without any modification to remove duplicate rows:

```
#perform union with df1 and df2
df_union = df1.union(df2)

#view final DataFrame
df_union.show()

+----+----------+------+
|team|conference|points|
+----+----------+------+
| A| East| 11|
| B| East| 8|
| C| East| 31|
| D| West| 16|
| E| West| 6|
| A| East| 11|
| B| East| 8|
| G| East| 31|
| H| West| 16|
+----+----------+------+
```

ARABPSYCHOLOGY.COM

Upon reviewing the resulting `df_union` DataFrame, it is immediately clear that the rows corresponding to 'Team A' and 'Team B' are present twice, once from **df1** and once from **df2**. This confirms that the base `.union()` function in PySpark simply concatenates the two datasets, retaining all duplicates. This intermediate result, though merged, is not ready for analysis if uniqueness is a requirement.

Achieving the Final Result: Union and Distinct

To transform the concatenated dataset into a mathematically correct set union--a collection where every row is unique--we must append the distinct method to our union operation. This powerful chained execution handles the merging and subsequent filtering in a single, efficient command, producing the clean, final result required by data analysts.

We utilize the combined syntax below to perform the union and mandate that only unique rows be returned:

#perform union with df1 and df2 and return only distinct rows
df_union_distinct = df1.union(df2).distinct()

#view final DataFrame
df_union_distinct.show()

```
+----+----------+------+
|team|conference|points|
+----+----------+------+
| A| East| 11|
| B| East| 8|
| C| East| 31|
| D| West| 16|
| E| West| 6|
| G| East| 31|
| H| West| 16|
+----+----------+------+
```

The resulting `df_union_distinct` DataFrame successfully merges the contents of **df1** and **df2** while eliminating the duplicate entries for Teams A and B. We are left with a final dataset containing only seven unique records. This demonstrates the seamless and efficient use of the combined union and **distinct** transformation for clean data integration within PySpark workflows, providing a reliable method for generating unique merged datasets.

ARABPSYCHOLOGY.COM

Summary of Best Practices

When performing data integration tasks in large-scale environments, maximizing efficiency and minimizing data redundancy are critical. The combination of `.union()` followed by `.distinct()` is the canonical method in PySpark for achieving a SQL-style UNION result, ensuring the data is merged and simultaneously cleaned.

Key considerations for optimizing this process include:

Schema Alignment: Always verify that the input DataFrames share an identical schema (column names, types, and order) before performing the union. Using `unionByName()` can mitigate issues if column order differs but schema types are compatible.
Performance Trade-offs: Remember that `.distinct()` is a resource-intensive operation due to the required data shuffling and hash comparisons across all partitions. If you only need to deduplicate based on a subset of columns (e.g., matching on ID but keeping the latest timestamp), consider using `.dropDuplicates(subset=)` instead, as it may offer superior performance for partial deduplication scenarios.
Alternative Approaches: For very large datasets where the cost of `.distinct()` is prohibitive, consider filtering the DataFrames before the union if you have a reliable key to identify already-processed rows, reducing the volume of data that must be shuffled during the final distinct phase.

Note: You can find the complete documentation for the PySpark **union** function, along with detailed API specifications for all DataFrame transformations, on the official Apache Spark documentation website.

ARABPSYCHOLOGY.COM