How to Combine and Deduplicate DataFrames in PySpark Using Union and Distinct

Authored by

stats writer

January 2, 2026

Understanding PySpark DataFrames and Set Operations

In the realm of Apache Spark and big data processing, the ability to efficiently combine large datasets is fundamental. PySpark provides powerful, distributed mechanisms to handle these tasks, primarily through its DataFrame API. A DataFrame is essentially a distributed collection of data organized into named columns, analogous to a table in a relational database or a data frame in R/Python (Pandas), but built for massive scalability.

When merging two or more DataFrames, we are performing a set operation. The most common operation is the **Union**, which appends the rows of one DataFrame to the rows of another. However, a crucial consideration in data integration is handling redundancy. If both source DataFrames contain identical records, a simple union operation will retain these duplicates, potentially skewing subsequent analysis or increasing storage overhead. Therefore, knowing how to perform a union while simultaneously ensuring that only **distinct** rows are returned is essential for producing clean, reliable datasets.

To achieve this seamless merger and deduplication, we utilize a combination of two core PySpark DataFrame methods: the union() method for concatenation, and the distinct() method for row-level uniqueness filtering. By chaining these operations together, we execute a single, optimized transformation that guarantees a comprehensive and deduplicated result.

The Difference Between Union and Deduplication Requirements

It is important to understand the behavior of the standard `union()` method in PySpark compared to its SQL counterpart. In traditional SQL environments, the `UNION` operator inherently removes duplicate rows, while `UNION ALL` retains them. PySpark's DataFrame API simplifies this by having a single `union()` method that behaves like SQL's `UNION ALL`--it combines all rows from both DataFrames, preserving all records, including duplicates, provided the schemas are compatible.

Compatibility requires that both DataFrames have the exact same number of columns, column names, and corresponding data types. If the schemas differ, PySpark will raise an error. For scenarios where schemas are identical but we need the output to be truly unique (like SQL's standard `UNION`), we must manually apply the deduplication step. This necessity is common in ETL pipelines where data streams might contain overlapping entries due to re-processing or source system synchronization issues.

The distinct() method is the specified tool for this purpose. When applied to a DataFrame, it evaluates every row and ensures that the final output DataFrame contains only unique rows across all columns. By calling `distinct()` immediately following the `union()`, we instruct the underlying Spark engine to first merge the datasets and then process the resulting intermediate data to

eliminate redundancy before finalizing the output.

Syntax for Combining and Deduplicating DataFrames

The solution for performing a union and returning only distinct rows is achieved through method chaining, a highly readable and idiomatic practice in PySpark. This approach allows developers to sequentially apply transformations to a DataFrame object without generating numerous intermediate variables.

The standard syntax for performing a combined union and deduplication operation in PySpark is concise and relies on method chaining:

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

This powerful chain first merges all records from **df1** and **df2** using the union() method, resulting in a temporary DataFrame that includes duplicates. Subsequently, the distinct() method is applied, which scans the combined dataset and eliminates any rows that are exact matches across all columns, yielding the final deduplicated result: **df_union_distinct**.

It is worth noting that while other methods like `dropDuplicates()` exist, the simple `distinct()` call is specifically designed for row-level uniqueness based on all columns, making it the most direct and idiomatic choice for following a general union operation. The efficiency of this operation relies heavily on Spark's ability to distribute the comparison and shuffling processes across the cluster, ensuring that performance remains strong even with massive datasets.

Setting up the PySpark Environment and Sample Data (df1)

To illustrate this process clearly, we will set up a minimal PySpark environment and define two sample DataFrames. The first step involves initializing a **SparkSession**, which serves as the entry point to programming Spark with the DataFrame API. This session manages the connection to the cluster and allows us to create, transform, and query DataFrames.

We define the first dataset, **df1**, representing initial sales or team data. This DataFrame is crucial as it forms the baseline of our combined dataset, containing unique records that we expect to be retained.

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

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

As shown in the output, **df1** contains five distinct records, each identified by a unique team identifier and associated conference and points data. This structure ensures that we have a clean starting point for demonstrating both duplication and deduplication in the subsequent steps.

Creating the Second PySpark DataFrame (df2)

Next, we create a second DataFrame, **df2**, which represents new or updated data. This dataset is intentionally designed to include two types of records relative to **df1**: records that are exact duplicates (to test the deduplication logic) and records that are unique (to ensure they are correctly merged).

Specifically, the first two rows of **df2** (Teams A and B) are identical to the first two rows of **df1**. The remaining rows (Teams G and H) are new and unique. This setup provides the perfect scenario to observe the behavior of the standard union operation versus the union and distinct operation.

#define data
data2 = ,

,
,
]

#define column names
columns2 =

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

#view DataFrame
df2.show()

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

Notice that **df2** holds four records. When we combine **df1** (5 records) and **df2** (4 records) using a standard union, we expect a total of 9 rows, as the two duplicate rows (A and B) will be retained.

Demonstrating the Standard Union Operation (With Duplicates)

Before applying the deduplication logic, let us first execute the basic union() method. This demonstrates why the subsequent `distinct()` call is necessary. We simply call `df1.union(df2)` to merge the two datasets.

#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
+----+----------+------+

Upon reviewing the output DataFrame, **df_union**, we clearly see that it contains 9 rows (5 from **df1** + 4 from **df2**). Crucially, the rows corresponding to Team A ('A', 'East', 11) and Team B ('B', 'East', 8) appear twice. This confirms that PySpark's default `union()` operation retains all records from both source DataFrames, even when they are exact duplicates.

While this behavior is correct for many data aggregation tasks (like financial transaction logging where every entry, even identical ones, must be preserved), it poses a problem for set-based integration where we require a single, consolidated view of all unique entities. This redundancy must be eliminated to ensure accurate cardinality counts and streamlined data processing.

Performing Union and Returning Distinct Rows

Now we implement the full solution by chaining the `union()` and distinct() method. This single line of code instructs Spark to perform the union and then immediately apply a global deduplication across the resultant dataset. This is the canonical way to achieve the effect of SQL's `UNION` operator in PySpark.

#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 DataFrame, **df_union_distinct**, contains 7 rows. If we compare this to the original 9 rows in **df_union**, we see that the two duplicate entries (Teams A and B) have been successfully removed. The final dataset accurately represents the consolidated set of unique records from both source DataFrames.

This demonstrates the effectiveness and simplicity of method chaining for complex data manipulation tasks in PySpark. The entire operation is handled internally by the Spark engine, which optimizes the distribution and comparison of data partitions to ensure efficient processing across the cluster nodes, even for incredibly large data volumes.

Performance Considerations and Alternatives

While `union().distinct()` is the most straightforward method for general deduplication following a merger, practitioners should be aware of potential performance implications. The `distinct()` operation is computationally expensive because it requires a global shuffle across the entire cluster. During a shuffle, data rows are physically moved between execution nodes to gather identical keys (or rows, in this case) together for comparison and elimination. This network overhead can be significant for extremely large DataFrames.

If the DataFrames are extremely large and you only need uniqueness based on a subset of columns (e.g., uniqueness based on an ID column), an alternative approach is to use the `dropDuplicates()` method, specifying the subset of columns to consider. However, for ensuring full row integrity uniqueness after a union, `distinct()` remains the standard and most robust choice. Always profile your operations in a real-world cluster environment to ensure the chosen method scales efficiently with your data growth.

Alternatively, if you are certain that one DataFrame is a strict subset of the other (meaning **df2** only contains new records or updates to existing records in **df1**), you might consider more sophisticated join or window functions to achieve merging and deduplication, but for simple concatenation and row-wise uniqueness, the chained `union().distinct()` call provides the cleanest code and necessary functionality.

Summary and Further Resources

Combining data efficiently is a cornerstone of data engineering using PySpark. By leveraging the

DataFrame methods `union()` and `distinct()`, we can achieve a SQL-style unique union operation that merges two datasets while guaranteeing that only unique rows are retained. This method is concise, idiomatic, and leverages the distributed processing power of Apache Spark.

To summarize the steps:

Initialize the **SparkSession** and create the source DataFrames (df1 and df2).
Ensure both DataFrames share compatible schemas (number, names, and types of columns).
Chain the operations: `df1.union(df2).distinct()` to perform the merger and subsequent deduplication.

For detailed information on the functionalities discussed, refer to the official PySpark documentation. The documentation provides exhaustive details on parameters, return types, and advanced usage scenarios for both the union and distinct DataFrame methods.

Note: You can find the complete documentation for the PySpark **union** function here and the `distinct` function in the pyspark.sql.functions module documentation.