Learning PySpark: Combining DataFrames Using Union for Distinct Rows

Authored by

Mohammed looti

November 11, 2025

The Imperative of Data Merging: PySpark and Set Theory

In modern data engineering and big data processing environments, the ability to efficiently consolidate disparate datasets is not merely a feature but a foundational requirement. Apache Spark, through its powerful Python API, the PySpark DataFrame, offers highly optimized tools for data manipulation, heavily leveraging concepts rooted in set operations. Data, frequently sourced from various systems, historical archives, or real-time streams, often requires vertical stacking--appending rows from one source onto another--to create a unified view. However, this merging process invariably introduces the risk of redundancy. When source datasets contain overlapping or identical records, a simple concatenation results in unwanted duplicates, compromising the integrity of subsequent analysis and modeling efforts.

The default behavior of the standard union transformation in PySpark is designed for maximum speed and performance. It executes a straightforward concatenation, effectively stapling the rows of the second DataFrame onto the first without performing any intrinsic check for data duplication. This intentional design choice acknowledges that checking for duplicates across petabyte-scale datasets is a profoundly computationally expensive task, often necessitating extensive data shuffling across the cluster. Consequently, if the objective is to produce a mathematically accurate set union--a consolidated dataset where every single row is guaranteed to be unique--the user must explicitly mandate a deduplication procedure immediately following the initial concatenation.

This article serves as a comprehensive guide to mastering the precise methodology required to execute a robust union operation between two PySpark DataFrames while simultaneously enforcing the return of only **distinct rows** in the final output. This technique is absolutely indispensable for maintaining data quality, ensuring the accuracy of standardization pipelines, and guaranteeing data integrity, especially before deploying data for critical analytical calculations or feeding cleansed datasets into resource-intensive machine learning models. We will dissect the exact syntax, explore the underlying execution logic, and provide a detailed, practical, step-by-step example demonstrating how to transform two potentially overlapping raw datasets into a single, comprehensive, and fully deduplicated result set suitable for production use.

Mastering the Union and Distinct Chaining Technique

The path to performing a successful, deduplicated union in PySpark involves chaining two specific, yet fundamental, DataFrame transformations. The process begins with the standard **union** method, responsible for the initial vertical stacking of rows, and is immediately followed by the **distinct** transformation, which enforces global uniqueness. The primary function of the `union()` method is to combine the rows, but it operates under a strict set of prerequisites: the schemas of the two source DataFrames must align exactly. This means they must share the same number of columns, and corresponding columns must possess compatible data types to prevent execution

errors. Once the rows are combined--often resulting in an intermediate DataFrame riddled with duplicates if the source data overlapped--the secondary transformation takes effect.

The subsequent invocation of the distinct transformation is the crucial step that compels the Spark execution engine to scan the entirety of the newly combined dataset. During this scan, Spark identifies and flags any rows that are identical across all columns. It then efficiently retains only a single instance of each unique row, thereby performing the necessary filtering and ensuring the output aligns with true set theory principles. It is paramount to grasp that the **distinct()** operation is a global function; it considers two rows duplicates only if the values across every single column match perfectly. This comprehensive check is what distinguishes a reliable, clean union from a simple, potentially redundant concatenation.

The syntax for implementing this powerful, two-part operation is remarkably concise, reflecting PySpark's expressive API design. Assuming we have two existing DataFrames, conventionally named **df1** and **df2**, the chained operation is executed as follows:

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

This single line of code cleanly encapsulates the entire operational workflow. First, **df2** is appended to **df1** via the raw concatenation provided by union, and immediately thereafter, the full set of redundant records is systematically eliminated by the distinct call. The final resultant DataFrame, **df_union_distinct**, is guaranteed to contain the complete combined data structure, but with the necessary assurance of uniqueness across all constituent records, thereby fulfilling the stringent requirement for a clean and analytically sound union operation.

Setting the Stage: Constructing Overlapping PySpark DataFrames

To provide a concrete, reproducible illustration of this essential process, we must first establish two sample PySpark DataFrames that intentionally contain overlapping records. This setup is critical because it visually demonstrates the shortcomings of a standard union operation when deduplication is mandatory. The foundation of any PySpark operation begins with the initialization of the SparkSession, which acts as the primary gateway for utilizing all Spark cluster functionality. Following this initialization, we define our first dataset, **df1**, which represents an initial collection of hypothetical team performance metrics.

The following comprehensive code snippet outlines the creation of the first DataFrame, **df1**. This includes the explicit definition of the raw data, the clear naming of the column schema, and the necessary command to display the resulting structure for verification. It is important to pay close attention to the specific rows and their values defined here, as they will be central to observing the duplication issues when the second DataFrame is introduced and merged.

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

Subsequently, we introduce the second DataFrame, **df2**. This dataset is intentionally designed to be structurally identical to **df1**, a non-negotiable prerequisite for any successful union operation within PySpark. Crucially, **df2** is constructed to include two specific records (Team A and Team B) that are exact matches to existing entries in **df1**, alongside two completely new and unique entries (Team G and Team H). This calculated overlap ensures that when we execute the standard, non-deduplicated union, we can clearly and unequivocally observe the problem of retained duplicate rows, setting the stage for the final corrective action.

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

Analyzing the Default Behavior: The Standard Union Operation

Prior to implementing the desired deduplicated union, it is highly instructive and necessary to first execute a standard union operation between the two prepared DataFrames, **df1** and **df2**. This preliminary step vividly illustrates the default, high-speed behavior of PySpark, which is designed exclusively to append the contents of the second DataFrame to the first, completely disregarding the content's uniqueness. This basic concatenation operation is inherently faster than any process involving duplicate checking because it avoids the computationally expensive steps of a full data shuffle and record comparison phase across the distributed cluster.

By executing the command `df_union = df1.union(df2)`, we generate a new DataFrame, **df_union**, which is the simple aggregation of all rows present in both source DataFrames. Given that **df1** contained five distinct rows and **df2** contained four rows, the resulting DataFrame must necessarily contain a total of nine rows. The critical observation here is that the records corresponding to Team A (East, 11) and Team B (East, 8) are now present twice--once originating from **df1** and then again from **df2**. This outcome conclusively confirms that the basic union operation is strictly a concatenation utility, not a true mathematical set union which inherently removes redundancy.

The output below provides the undeniable visual evidence of these retained duplicates. The initial five rows correspond precisely to the contents of **df1**, and the subsequent four rows are the

appended contents of **df2**. Note the clear repetition of the 'A' and 'B' entries appearing in sequence within this raw combined result set.

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

This intermediate step is fundamentally important as it provides a tangible confirmation of the presence of redundant data, a condition that is often highly undesirable and detrimental in rigorous analytical pipelines. If this combined DataFrame were to be used directly for aggregation--for example, calculating the total points for all teams--the resulting metric would be significantly skewed due to the unintended double counting of performance metrics for teams A and B. Correcting this critical duplication issue requires the immediate and careful introduction of the highly efficient distinct operation.

Executing the True Set Union: Deduplication with .distinct()

To definitively resolve the issue of retained duplicates observed in the preceding step and successfully achieve a mathematically precise set union, we integrate the powerful **distinct()** method directly into the operation chain, following the initial `union()` call. This deliberate chaining ensures that once the two source DataFrames have been appended, Spark immediately initiates a comprehensive, global deduplication process. This involves scanning the entirety of the intermediate combined dataset to identify and subsequently discard any rows that constitute exact matches, thereby cleansing the data. This robust operation inherently requires a data shuffle

across the distributed cluster, a necessary step because Spark must gather all identical records onto the same partition before it can accurately filter them down to a single instance. This requirement explains why this process is not the default behavior of the simple union function.

The following refined syntax applies this essential correction, creating the final, pristine, and fully deduplicated DataFrame, named **df_union_distinct**:

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

By meticulously observing the output of `df_union_distinct.show()`, we can confidently confirm the complete and successful deduplication. The resultant DataFrame now displays exactly seven unique records, a significant reduction from the nine rows present in the raw, concatenated union. The two duplicate records (Team A and Team B) have been successfully merged into their unique representations, ensuring that our final dataset precisely and accurately represents the consolidated, non-redundant set of team entries drawn from both the initial source DataFrames. This technique, combining the speed of union with the rigor of distinct, is absolutely fundamental for robust and trustworthy data merging operations in any large-scale PySpark workflow.

Advanced Considerations: Performance, Shuffling, and Schema Integrity

While the `union().distinct()` pattern stands as the definitive and highly effective methodology for achieving a fully deduplicated result set, experienced data engineers must maintain a keen awareness of its operational costs, particularly when deployed against truly massive, production-scale datasets. The **distinct()** operation, unlike the simple and inexpensive row concatenation,

mandates a full comparison of every row against every other row across the distributed environment, which necessitates resource-intensive operations within the PySpark DataFrame execution engine.

The primary performance implications arise from the substantial requirement for data shuffling. When **distinct()** is invoked, Spark must meticulously reorganize and redistribute the data across all available cluster nodes to ensure that every identical record is routed to the exact same partition. This grouping is an absolute necessity for the subsequent filtering process to occur accurately and reliably. For DataFrames that are extremely large, this required shuffling stage can introduce significant latency and consume substantial network and disk I/O resources. If performance is the overriding critical factor, and duplication is expected only across a small, defined subset of columns (and not the entire row), alternative, though more complex, strategies such as utilizing the `groupBy().agg()` pattern might offer a more performant alternative. However, for the unconditional guarantee of a full-row distinct result across all columns, the chained `union().distinct()` remains the standard, most reliable, and conceptually simplest approach.

Finally, rigorous attention must always be paid to ensuring strict schema compatibility before attempting any union operation. PySpark imposes non-negotiable requirements for the DataFrames being merged:

The DataFrames must possess the exact same number of columns.
The columns must be ordered identically.
Corresponding columns in both DataFrames must exhibit compatible data types.

Failure to meet these structural requirements will inevitably cause PySpark to raise a runtime error, halting the execution. If two source DataFrames contain the same columns but they are presented in differing orders, it is imperative to use DataFrame methods such as `select()` or `withColumnRenamed()` to explicitly align the column sequence before attempting to execute the union command. Adhering strictly to these compatibility rules ensures that the initial union operation executes smoothly, allowing the subsequent distinct transformation to successfully cleanse the merged data of any redundancies.