# Select Distinct Rows in PySpark (With Examples)

Authored by

**Mohammed looti**

November 11, 2025

Welcome to this expert guide on performing data deduplication using **PySpark**. Working with large datasets often necessitates identifying and removing duplicate records to ensure data integrity and accuracy in subsequent analytical processes. The **PySpark DataFrame** API provides robust and efficient methods for achieving this goal, whether you need to check for distinct rows across the entire dataset or isolate unique values within specific columns.

This article will break down the three primary techniques available in PySpark for handling distinct operations. We will demonstrate how to implement these methods using clear, runnable code examples, ensuring you can apply these powerful features to your own big data pipelines immediately. Understanding these operations is fundamental for anyone working in a **distributed computing** environment where data redundancy can significantly skew results and waste computational resources.

The following methods are the cornerstone of selecting distinct data within a PySpark environment:

**Method 1: Selecting Distinct Rows in the DataFrame:** Using the `.distinct()` transformation to identify and return only the unique rows based on all columns.

**Method 2: Selecting Distinct Values from Specific Columns:** Combining `.select()` and `.distinct()` to find the unique set of values within one or more designated columns.

**Method 3: Counting Distinct Rows:** Utilizing `.distinct()` followed by `.count()` to quickly determine the total number of unique records present in the dataset.

## Setting Up the PySpark Environment and Sample Data

Before diving into the deduplication methods, we must establish a working PySpark environment and define a sample **DataFrame** that contains known duplicate values. This setup allows us to rigorously test and observe the results of each distinct operation. The starting point for any PySpark application is initiating the **SparkSession**, which acts as the entry point to communicate with the Spark cluster.

Our sample data represents statistics for various players, containing columns for `team`, `position`, and `points`. Notice that this raw data intentionally includes several duplicate rows (e.g., the combination of 'A', 'Forward', 22 appears twice, and 'B', 'Guard', 14 appears twice). This structure is ideal for demonstrating effective **data deduplication**.

The following script initializes the necessary objects and displays our initial DataFrame, illustrating the redundancy we aim to resolve:

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

As shown in the output, the initial DataFrame contains 8 rows, with several rows being identical across all three columns. Our subsequent examples will demonstrate how to filter this DataFrame down to its truly unique records.

## Method 1: Selecting Distinct Rows Across the Entire DataFrame

The simplest and most common method for deduplication is applying the `.distinct()` transformation directly to the DataFrame. When `.distinct()` is called without any preceding

column selection, PySpark assumes that a row is only distinct if the combination of values across **all** columns is unique. This is the definition of a truly unique record within the context of the entire dataset schema.

This operation is critical when you need to ensure that every dimension and measure represented in your final dataset is unique. It forces Spark to compare the values in the `team`, `position`, and `points` columns simultaneously for every record. It is important to note that `.distinct()` is a wide transformation, meaning it requires shuffling data across the cluster nodes, which can be computationally intensive on extremely large datasets.

We utilize the following concise syntax to select the distinct rows in our sample DataFrame. The resulting output clearly shows that the duplicate rows (like the second instance of ('A', 'Forward', 22) and ('B', 'Guard', 14)) have been successfully eliminated, reducing the total row count from 8 to 6.

**#display distinct rows only**
**df.distinct().show()**

+----+--------+------+
|team|position|points|
+----+--------+------+
| A| Guard| 11|
| A| Guard| 8|
| A| Forward| 22|
| B| Guard| 14|
| B| Forward| 13|
| B| Forward| 7|
+----+--------+------+

Observe that the resulting DataFrame retains only those rows where the combined values of 'team', 'position', and 'points' are unique. This is the most stringent form of deduplication available in the PySpark API.

## Method 2: Isolating Unique Values in Specific Columns

In many analytical scenarios, the goal is not to find unique rows, but rather to identify all the unique categories or values present within one or two specific columns. For instance, we might want to know all the unique teams or unique positions represented in the dataset, regardless of the other column values.

To achieve this, we combine the `.select()` transformation with the `.distinct()` transformation.

By using `df.select('column_name')`, we project the DataFrame to only include the columns of interest. Then, applying `.distinct()` ensures that only the unique combinations within that smaller set of columns are returned. This technique is highly efficient when dealing with dimensional analysis or generating lookup tables.

Below, we demonstrate how to select the distinct values solely from the `team` column. This operation discards all information related to position and points, focusing purely on the categorical values of the team identifier.

**#display distinct values from 'team' column only**
**df.select('team').distinct().show()**

+----+
|team|
+----+
| A|
| B|
+----+

The resulting output confirms that the sample data contains only two distinct teams: **A** and **B**. This method is invaluable for tasks such as data profiling or generating lists of unique primary keys or categorical identifiers within a larger dataset. If you were to select multiple columns (e.g., `df.select('team', 'position').distinct()`), the resulting DataFrame would contain all unique combinations of those two fields.

## Method 3: Calculating the Total Count of Unique Records

While viewing the distinct rows is useful for verification, often the core requirement in production environments is simply obtaining the number of unique records, rather than materializing the full distinct dataset. This count provides a crucial metric for data quality checks, data volume reporting, and assessing the level of redundancy.

To calculate this metric, we chain the `.distinct()` transformation with the `.count()` action. Remember that `.distinct()` is a transformation that defines the operation, but it is not executed until an action like `.count()` or `.show()` is called. The `.count()` action forces the execution of the deduplication logic across the cluster and returns a single numerical result representing the total number of unique rows identified.

Using this method is significantly faster than calculating the distinct rows and then manually counting them, as it optimizes the execution plan specifically for returning the final scalar value. The following code demonstrates this efficient approach:

```
#count number of distinct rows
df.distinct().count()
```

```
6
```

The output, **6**, accurately reflects the number of unique records we observed in Method 1. This confirms that of the original 8 records, only 6 are truly distinct when all columns are considered. This powerful combination of methods is frequently used in large-scale data auditing.

## Performance Considerations for PySpark Distinct Operations

While the `.distinct()` method is syntactically simple, it is important for expert PySpark developers to understand its performance implications. Since Spark is a **DataFrame** processing engine built for parallelism, any operation that requires comparing all records across all partitions necessitates a data shuffle.

The `.distinct()` operation is inherently expensive because it must move all data records that share the same key to the same physical node to ensure accurate comparison and deduplication. This full network transfer can become a bottleneck when dealing with petabyte-scale datasets. Therefore, optimization strategies must be considered, especially if distinct counts are performed frequently.

If performance is critical, consider using the `dropDuplicates()` function, which offers a slight variation. While `distinct()` operates on all columns, `dropDuplicates()` allows you to specify a subset of columns to check for uniqueness, keeping the first occurrence of the unique combination found in the specified columns. Furthermore, for situations where an exact count is not necessary, Spark offers the `approx_count_distinct()` function within `pyspark.sql.functions`. This function uses algorithms like HyperLogLog to provide a highly accurate estimate of the distinct count with significantly reduced shuffle requirements, offering a massive performance boost for large-scale approximations.

## Additional Resources

Mastering deduplication is just one step in optimizing your PySpark workflow. To continue expanding your expertise in data manipulation and transformation within the Apache Spark ecosystem, we recommend exploring tutorials on related topics.

These concepts are fundamental to clean and effective data engineering:

Understanding how to filter DataFrames using complex conditional logic.

Implementing window functions for advanced analytical processing.

Optimizing data partitioning to minimize shuffles and improve query execution times.