How to Count Duplicate Rows in PySpark Easily

Authored by

stats writer

February 5, 2026

Ensuring high data quality is paramount in large-scale data processing, and identifying redundant or duplicate records is a critical step. When working with massive datasets in PySpark, efficiently counting duplicate rows requires leveraging Spark's distributed processing capabilities. The most robust and widely accepted method involves using a combination of aggregation functions to group the data based on all columns and subsequently filter for those groups that appear more than once.

The fundamental strategy relies on transforming the DataFrame into a structure where each unique combination of values across all columns is represented as a single entry, accompanied by a frequency count. If this frequency count is greater than one, it signifies the presence of duplicates. This technique mirrors common practices used in traditional SQL environments, adapted for the distributed nature of the Spark ecosystem.

Alternatively, while less efficient for simple counting across very large datasets, one can determine the number of duplicate rows by calculating the difference between the total number of rows in the original DataFrame and the total number of unique rows remaining after applying the `.dropDuplicates()` transformation. However, the aggregation method using `.groupBy()` and `.count()` generally provides greater insight by allowing users to inspect which specific rows are duplicated and how many times they occur, which is the focus of the following detailed steps.

Standard Aggregation Method for Counting Duplicates

The standard methodology for calculating the total count of duplicate records relies on chaining several critical PySpark transformations and actions. This approach is highly efficient because it leverages Spark's ability to hash and group data across partitions, minimizing unnecessary data shuffling and maximizing performance on large clusters.

To accurately count rows that appear more than once, we must first aggregate based on all columns present in the DataFrame. This ensures that only true, record-for-record duplicates are identified. The following syntax provides the blueprint for achieving this count efficiently, yielding a single result representing the total number of non-unique entries.

The core logic involves four chained operations: `.groupBy()`, `.count()`, `.where()`, and `.select()`. By grouping on `df.columns`, we enforce a grouping key that encompasses every field, guaranteeing that aggregation only occurs when all values match exactly. The subsequent `.count()` transformation calculates the frequency of occurrence for each unique group, storing this frequency in a temporary column often named 'count'.

The filtering step, `.where(F.col('count') > 1)`, isolates only those groups that are duplicates (i.e., they appeared more than once). Finally, the `.select(F.sum('count'))` operation sums the resulting counts. Crucially, this sum provides the total number of duplicate **occurrences**, which is

the total count of rows that belong to a redundant set.

import pyspark.sql.functions as F

df.groupBy(df.columns)
.count()
.where(F.col('count') > 1)
.select(F.sum('count'))
.show()

Preparing the Sample Data for Duplicate Row Detection

To effectively demonstrate this counting mechanism, let us define a sample DataFrame containing athlete statistics. This dataset is intentionally structured to include several full-row duplicates, allowing us to verify the accuracy of the PySpark aggregation method. We start by initializing the **SparkSession**, which is the entry point to all Spark functionality required for distributed computing.

The dataset, which includes columns for `team`, `position`, and `points`, clearly shows two distinct sets of duplicated records. These duplicates represent instances where the entire row content--all three columns--is identical to another row within the DataFrame. Specifically, we have two instances of 'A, Forward, 22' and two instances of 'B, Guard, 14'.

The following code snippet shows the creation of the dataset and the resulting DataFrame structure. The output confirms the presence of 8 total records before processing, which serves as our baseline for identifying data redundancy.

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

Executing the Aggregate Query and Interpreting the Total Count

Once the DataFrame is defined, we can apply the four-step aggregation process defined earlier. This process efficiently sweeps through the dataset, identifies the groups with redundancy, and sums the excess row counts. This action provides us with the single, definitive metric for the total number of duplicate occurrences in the data.

The result of the `.show()` action will be a small DataFrame containing one row and one column (named `sum(count)`), indicating the scalar total. It is crucial to remember that this number represents the count of rows that are identical copies of other rows **plus** the original rows that belong to a duplicate set, essentially counting every row involved in redundancy.

In our basketball player example, we anticipate finding a count of 4. This is because the aggregation isolates two groups, each occurring twice. Summing their counts (2 + 2) yields the total number of records that were not uniquely present in the original dataset.

```
import pyspark.sql.functions as F

#count number of duplicate rows in DataFrame
df.groupBy(df.columns)
.count()
.where(F.col('count') > 1)
```

```
.select(F.sum('count'))
.show()

+----------+
|sum(count)|
+----------+
| 4|
+----------+
```

The result clearly shows that **four** rows in the DataFrame are involved in duplicate groups. This aligns perfectly with our manual inspection, confirming the reliability and accuracy of this structured aggregation in PySpark.

Viewing the Specific Duplicate Rows for Data Inspection

While obtaining the total count is useful for summary reporting, data governance often requires analysts to inspect the specific records identified as duplicates. This inspection helps determine the source of the data redundancy and define a strategy for remediation (e.g., dropping all but one occurrence, or investigating the data pipeline). We can easily achieve this by modifying the end of the previous code chain.

By removing the final `.select(F.sum('count'))` operation, the DataFrame output retains the grouped records along with their respective frequency counts. The `.where(F.col('count') > 1)` filter ensures that only the groups identified as having more than one occurrence are displayed.

This detailed view provides immediate feedback on which specific row values are duplicated and confirms the frequency (the number in the `count` column). This level of granularity is essential for debugging data integrity issues and formulating data cleaning rules.

import pyspark.sql.functions as F

```
#view duplicate rows in DataFrame
df.groupBy(df.columns)
.count()
.where(F.col('count') > 1)
.show()

+----+--------+------+-----+
|team|position|points|count|
+----+--------+------+-----+
| A| Forward| 22| 2|
```

| B| Guard| 14| 2|
+----+--------+------+-----+

The resulting table clearly isolates the two unique combinations that are duplicated. We observe that the row `A, Forward, 22` occurred **2** times, and the row `B, Guard, 14` also occurred **2** times. Summing these counts (2 + 2) confirms the previously calculated total of **4** duplicate occurrences within the original DataFrame.

Returning the Duplicate Count as a Single Scalar Value

In environments where the final output needs to be integrated into another system, or when the calculation is part of a larger workflow, it is often necessary to retrieve the count as a simple, non-DataFrame, scalar value. The `.show()` action is designed primarily for display purposes, but the `.collect()` action can be used to pull the result back to the driver program as a native Python object.

When using `.collect()` on a DataFrame that has been reduced to a single row and column (as is the case after the aggregation and summation steps), we must access the specific elements of the resulting list of **Row** objects. Since the aggregated DataFrame contains only one row at index , and only one column (the sum of counts) at index , accessing `.collect()` yields the raw integer count.

This technique is highly valuable when the duplicate count needs to be used immediately in conditional logic, logging, or passing the metric to reporting functions outside the Spark execution environment, eliminating the need to parse the structured output of `.show()`.

import pyspark.sql.functions as F

#count number of duplicate rows in DataFrame
df.groupBy(df.columns)
.count()
.where(F.col('count') > 1)
.select(F.sum('count'))
.collect()

4

Using this particular syntax bypasses the display formatting of `.show()` and returns only the integer value, which is **4** in this example, streamlining the output for further programmatic use within Python scripts.

Alternative Approach: Calculating Duplicates via the `dropDuplicates()` Method

While the aggregation method is generally preferred for its detailed view and standard application, an alternative approach involves comparing the size of the original DataFrame against the size of a deduplicated version. This method utilizes the `dropDuplicates()` transformation, which removes all rows that are exact copies of previous rows, leaving only unique entries.

The mathematical relationship is straightforward: the total number of rows in the original DataFrame (A) minus the count of truly unique rows after deduplication (B) gives the count of rows that were redundant copies (C). This calculation provides the number of *redundant copies* rather than the total number of occurrences in the duplicate set.

To implement this, you would first calculate `df.count()` to get the total row count. Then, you would calculate `df.dropDuplicates().count()` to get the unique row count. The difference between these two results provides the count of rows removed due to duplication (2 in our example: 8 - 6 = 2). This method is often useful when the primary goal is simply to quantify data reduction rather than analyze the frequency distribution of specific duplicate sets.

Summary of Techniques for Data Quality Assurance in PySpark

Counting duplicate rows is a foundational task in data cleansing and validation workflows within big data environments. PySpark offers highly optimized functions to perform this analysis efficiently across distributed clusters.

We explored the primary method utilizing `groupBy(df.columns)` followed by `.count()` and filtering. This technique provides the most detailed view, allowing not only the calculation of the total number of duplicate occurrences but also the inspection of the specific records involved. Furthermore, integrating `.collect()` facilitates seamless integration of the result into non-Spark Python processes, enhancing automation capabilities.

By mastering these techniques, data engineers and analysts can ensure the integrity and reliability of their datasets, leading to more accurate analytical outcomes and robust production systems. Choosing between the aggregation method and the `dropDuplicates()` method depends on whether the goal is to count all occurrences in duplicate sets (aggregation) or simply to count the redundant copies that can be removed (difference method).

The following resources explain how to perform other common data quality and manipulation tasks in PySpark: