Counting Duplicate Rows in PySpark DataFrames: A Step-by-Step Guide

Authored by

Mohammed looti

November 11, 2025

Handling data quality issues, such as identifying and quantifying duplicate rows, is a fundamental and often challenging task in modern **data engineering**. When processing datasets that span terabytes or petabytes, relying on powerful distributed computing frameworks becomes absolutely essential. This comprehensive guide focuses on demonstrating how to efficiently calculate the exact total number of redundant records within a DataFrame using PySpark. Crucially, this technique goes beyond merely counting the number of distinct groups that are duplicated; it computes the precise overall count of rows that are redundant.

The entire process is built upon a robust and highly optimized combination of aggregation and filtering functions made available within the **pyspark.sql.functions** module. This methodology ensures that every single column across the entire dataset is rigorously considered during the duplication check, providing a comprehensive and accurate count necessary for rigorous data cleaning and validation pipelines. Achieving this level of precision and scale requires leveraging Apache Spark's parallel processing capabilities.

The general syntax provided below represents the most concise and effective mechanism to execute this aggregation successfully across a distributed environment. This pattern is not only highly optimized for performance but also maintains excellent readability within a typical PySpark workflow, making it the industry standard for full-row duplication assessment:

import pyspark.sql.functions as F

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

In the following sections, we will walk through a practical, step-by-step example. We will define sample data, execute the calculation, and rigorously validate the final count. Furthermore, we will dissect each component of the chained functions--from the initial grouping to the final summation--to fully understand the underlying mechanics necessary for identifying these critical redundant records.

Setting Up the PySpark Environment and Sample Data

Before any advanced distributed data analysis can commence, the establishment of an Apache Spark session is a mandatory prerequisite. This session provides the essential execution context needed for interacting with distributed data structures efficiently. For the purpose of this demonstration, we will define a simple, yet highly illustrative, dataset representing simulated

basketball player statistics. This sample data has been intentionally structured to include several clear instances of redundancy, allowing us to easily and definitively verify the accuracy of our duplicate counting methodology later on.

Our sample data incorporates columns for the **team**, **position**, and **points** scored. A row is formally defined as a duplicate only if the values across all three of these columns match another row exactly. This holistic approach is vital; it ensures we are identifying true, full-row duplicates rather than merely partial attribute matches, which is often the goal in data quality checks. Using a small, controlled dataset is perfect for illustrating the core concept before scaling this technique to production-grade data volumes, where manual verification would be impractical or impossible.

The code snippet below handles three key actions: first, the initialization of the Spark session; second, the definition of the raw Python data structure; and third, the subsequent conversion of this local structure into a distributed DataFrame. Note the utilization of **SparkSession.builder.getOrCreate()**, which is the standard, idempotent method for ensuring the core execution environment is properly initialized. We conclude by displaying the resulting DataFrame using the **show()** action, confirming its structure and content before moving forward with the analysis.

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

Upon visual inspection of the resulting output, two distinct sets of duplicate rows are immediately observable: the row corresponding to Team A, Forward, 22 points, and the row for Team B, Guard, 14 points. Since each of these unique duplicate records appears exactly twice, we have a total of four individual rows contributing to the overall duplicate count (two pairs of duplicates). Our primary analytical objective is now to confirm this precise count programmatically using PySpark's optimized functions.

Step-by-Step Implementation: Calculating the Total Duplicate Count

The core methodology for counting duplicates involves a sophisticated yet elegant chaining of operations that effectively leverages the inherent efficiency of PySpark's distributed engine. This technique is markedly superior to simplistic approaches, such as using **dropDuplicates()** followed by a count, because **dropDuplicates()** only reports the number of unique rows remaining, failing to provide the essential metric of the total number of redundant entries that were removed.

The first critical step in the code block below utilizes the groupBy transformation, specifically passing **df.columns** as the argument. Grouping by all columns forces Apache Spark to enforce the strict condition that two rows can only belong to the same group if every single field matches exactly. This is the precise definition required for identifying a true, full-row duplicate. Following this grouping, the **count()** function is applied. This transforms the grouped DataFrame into a result set where each row now represents a unique combination, and a new column, conveniently named 'count', indicates the frequency of that specific combination within the original dataset.

Subsequently, the **where(F.col('count') > 1)** clause acts as a filter on this aggregated result set. It retains only those groups--those combinations--where the frequency is strictly greater than one. These retained groups are, by definition, the groups that contain duplicates. Finally, to derive the total number of redundant rows (rather than just the count of duplicate groups), we must sum the frequency counts of these filtered groups. The **select(F.sum('count'))** action executes this final,

crucial aggregation, producing a single value representing the total count of rows involved in any duplication.

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

As definitively confirmed by the output, the analytical process successfully identified a total of **4** redundant rows within the DataFrame. This outcome validates our initial visual inspection and provides a quantifiable, essential metric for immediate data quality assessment. This single, aggregated value is frequently the key performance indicator reported in formal data validation summaries, clearly indicating the scale of the necessary cleaning or remediation effort required for the dataset.

Detailed Analysis: Identifying the Duplicate Rows

While knowing the raw total number of duplicate rows is absolutely critical for reporting, data engineers often require the ability to visualize or isolate the specific records involved in the duplication. This is necessary for forensic investigation, root cause analysis, and debugging data quality issues. This visualization step helps determine the source of the redundancy--for example, whether the duplicates are a result of ingestion errors, artifacts from complex joins, or systemic errors embedded in upstream processing pipelines.

To achieve this required visualization, we simply make a minor modification to the previous code chain: we remove the final aggregation step, **.select(F.sum('count'))**. By halting the chained operations immediately after the **where()** filter, the resulting DataFrame explicitly shows the unique duplicate groups along with their corresponding frequency, which is listed in the 'count' column. This intermediate result is highly informative, as it details precisely which data combinations are responsible for the observed redundancy in the dataset.

The initial steps remain identical: we utilize groupBy across all columns, calculate the frequency using the **count()** function, and then filter these frequencies to only include those greater than 1. Executing the **show()** action at this precise stage yields the following clear and explicit output, which lists the duplicated records by group:

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

This detailed output clearly confirms the existence of two distinct groups containing redundancy. The first row indicates that the exact combination (Team A, Forward, 22 points) occurs **2** times in the dataset. Likewise, the second row specifies that the combination (Team B, Guard, 14 points) also occurs **2** times. Summing the values in the 'count' column (2 + 2) flawlessly confirms the previously calculated total of **4** duplicate rows. This detailed, group-level view is absolutely essential for comprehensive data auditing and debugging, offering profound clarity regarding the nature and specific location of the data errors.

Alternative Method: Returning a Single Scalar Value

In most production environments, particularly within automated scripts or robust data pipelines, the primary goal is not to display a formatted table of results but simply to retrieve a single, raw numeric value representing the total duplicate count. This **scalar value** is crucial, as it can be immediately utilized in conditional logic, integrated into monitoring dashboards, or passed as input to subsequent functions. While the **show()** action is superb for interactive use and debugging within PySpark, it prints the result to standard output and returns a value of 'None', making it unsuitable for programmatic capture. For direct programmatic access to the numerical result, the **collect()** action is mandatory.

The **collect()** function serves the critical role of retrieving the resulting distributed data from the

Apache Spark cluster and bringing it back to the driver program as a local Python object, typically structured as a list of **Row** objects. Since our desired output--the result of **sum(count)**--is a single cell, the output of **collect()** will be a list containing one Row object, which in turn contains a single element (the integer sum).

To extract the raw integer value (4) from this structured Python object, precise indexing into the result is required: **.collect()**. The first index successfully accesses the first (and only) Row object within the list, and the second index then accesses the value of the first (and only) column within that specific Row object. This refinement is paramount for seamless integration into larger Python scripts that demand a direct numerical result for subsequent processing or reporting concerning duplicate rows in the DataFrame.

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
```

Notice that this refined syntax returns only the number **4** directly to the console or the designated calling environment variable, making it the ideal method for automated pipeline checks and validation routines. This is the preferred operational method whenever the script needs to execute subsequent actions based directly on the calculated count (e.g., if the count exceeds a predefined threshold, automatically raising a data quality alert).

Summary and Further Reading

Effectively managing and maintaining **data quality** is an ongoing commitment in modern data infrastructure. The ability to quickly and accurately quantify redundant records using the specialized groupBy aggregation technique in PySpark represents an indispensable skill for any data professional. By skillfully chaining **groupBy(df.columns)**, **count()**, and conditional filtering with **where(F.col('count') > 1)**, we can precisely isolate and measure the exact level of redundancy present in even the largest datasets. Crucially, this technique scales highly efficiently because the most computationally intensive operations--the grouping and counting--are executed in parallel across the distributed cluster nodes.

The choice between using the **show()** action and the **collect()** action depends entirely on the

specific operational context: **show()** is optimized purely for display, visualization, and interactive inspection during development, whereas **collect()** is absolutely essential for programmatically retrieving the numerical result and integrating it into robust, automated data workflows. A thorough mastery of these fundamental Apache Spark operations ensures the maintenance of data integrity and prepares the data effectively for complex downstream analytical tasks.

We strongly recommend further exploration into the extensive capabilities of the **pyspark.sql.functions** module. This module offers a vast array of powerful methods for complex data manipulation, sophisticated aggregation, and advanced quality checks that extend far beyond simple duplication counting. Developing a deep understanding of these functions is the key prerequisite to achieving true proficiency in big data processing using distributed systems.

Additional Resources

The following resources and tutorials explain how to perform other common and advanced tasks in PySpark:

How to utilize **window functions** for complex time-series analysis and advanced ranking operations in PySpark.

Essential techniques for efficiently joining extremely large DataFrames and strategies for effectively mitigating common issues like **data skew**.

Implementing robust schema validation and precise type checking protocols for streaming or bulk ingested data streams.

Strategies for optimizing cluster memory usage and effectively utilizing caching mechanisms for performance acceleration in iterative algorithms.