How to Remove Duplicate Rows from a PySpark DataFrame

Authored by

stats writer

February 7, 2026

Managing data quality is a critical task in large-scale data processing, and handling redundant or duplicate records is a common requirement. In the context of big data engineering using PySpark, efficiently identifying and isolating these duplicate entries within a DataFrame is essential for accurate analysis and model training. While the most straightforward approach involves removing duplicates using native functions, this article focuses on techniques used specifically for finding and inspecting the duplicate records themselves, which is often necessary for debugging or reporting purposes.

The fundamental function for duplicate management in PySpark is `dropDuplicates()`. As its name suggests, this function is designed to eliminate rows where all column values are identical, leaving only unique rows behind. However, to retrieve the actual list of duplicate records--those rows that are slated for removal--we must employ more advanced set operations available within the PySpark SQL API. This usually involves combining `dropDuplicates()` with set difference operations like `exceptAll()` to isolate the redundant entries.

Furthermore, flexibility is key. Data duplication rarely occurs across all columns; often, it is based on a specific subset of identifying columns, such as 'user_id' and 'timestamp'. PySpark accommodates this requirement by allowing users to pass a list of column names to `dropDuplicates()`. Beyond simple identification, techniques involving aggregation functions like `groupBy()` coupled with `count()` provide powerful mechanisms not only to detect duplicates but also to quantify their frequency, offering deeper insight into the data anomalies present within the massive datasets handled by the PySpark environment.

Techniques for Identifying Duplicate Rows in a PySpark DataFrame

The PySpark Approach to Duplicate Management

PySpark, built atop the powerful Apache Spark engine, handles data parallelism and distribution, which makes standard SQL techniques for duplicate checking impractical or inefficient when dealing with petabytes of data. Instead, PySpark leverages its optimized catalyst engine and specialized transformation functions to manage data integrity. When faced with the task of identifying duplicates, one must distinguish between two primary goals: first, merely counting how many duplicates exist, and second, extracting the actual list of rows that constitute the duplicate set.

The native method for identifying the duplicate set involves comparing the original DataFrame against a version of itself that has been cleaned of duplicates. This comparative approach is highly performant in a distributed environment because it relies on well-optimized set difference

operations. Specifically, we utilize the `exceptAll()` transformation, which returns the rows present in the left DataFrame that are not present in the right DataFrame, considering all rows, including duplicates, for comparison. By applying this logic, the original DataFrame minus its unique counterpart reveals the redundant entries.

There are two primary patterns that address nearly all duplicate identification scenarios. The first pattern targets rows where every single column value matches another row exactly, representing a true, full-row redundancy. The second, more common pattern involves identifying redundancy only when specific key columns (like identifiers or compound keys) match, ignoring potentially differing values in non-key columns (like timestamps or status flags). Mastery of these two fundamental techniques allows data engineers to precisely target and manage data quality issues.

Core Techniques for Identifying Duplicates

As mentioned, the core strategy relies on the interplay between the `dropDuplicates()` function and the set difference operation `exceptAll()`. The `dropDuplicates()` function is responsible for creating a pristine, unique version of the dataset. When called without arguments, it considers all columns for uniqueness check. When provided a list of column names, it only enforces uniqueness across those columns, retaining the first occurrence of the row found for that unique combination of keys.

The `exceptAll()` operation then becomes the key differentiator. Unlike the simpler `except()` function, `exceptAll()` accurately accounts for the number of occurrences of each row. If the original DataFrame (A) has three identical rows (R1) and the unique DataFrame (B) has one instance of that row (R1), then A.exceptAll(B) will correctly return the two extra duplicate instances of R1. This precise handling of multiplicity is crucial for accurate duplicate reporting in large-scale data cleansing pipelines.

The following common ways summarize the two most effective programmatic methods for displaying duplicate rows in a PySpark DataFrame:

Method 1: Find Duplicate Rows Across All Columns

This method identifies rows that are perfect copies of another row in the dataset. It requires the comparison of the entire row vector.

#display rows that have duplicate values across all columns
df.exceptAll(df.dropDuplicates()).show()

Method 2: Find Duplicate Rows Across Specific Columns

This method focuses on identifying rows that share the same values for a defined subset of identifying columns, even if other columns differ.

#display rows that have duplicate values across 'team' and 'position' columns
df.exceptAll(df.dropDuplicates()()).show()

Setting Up the Demonstration DataFrame

To illustrate these methods clearly, we will first define and create a sample DataFrame using PySpark. This DataFrame represents typical sports data, containing columns for `team`, `position`, and `points`. Crucially, the data is intentionally structured to include both full-row duplicates and partial duplicates (where only key columns match).

Initialization begins with importing the necessary `SparkSession` object, which is the entry point for using Spark functionality. We then define a list of data rows and the corresponding schema names. This ensures the DataFrame is created with the expected structure and data types, preparing it for subsequent transformation operations. Creating a session and defining the structure is a foundational step in any PySpark workflow.

The resulting DataFrame serves as our testing ground. Careful observation reveals specific redundant entries: rows 3 and 4 (A, Forward, 22) are identical; and rows 5 and 6 (B, Guard, 14) are also identical. Additionally, rows 1 (A, Guard, 11) and 2 (A, Guard, 8) are duplicates based only on the 'team' and 'position' columns, but not on 'points'. This variety allows us to test both primary duplicate detection methods effectively.

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

Method 1: Detecting Duplicates Across All Columns

When the objective is to find rows that are perfect, exact duplicates of one another, the check must be performed across every single column in the DataFrame. This scenario is fundamental for cleaning data where an entire record was accidentally ingested multiple times, perhaps due to logging errors or faulty ETL processes. The operation is highly sensitive, meaning even a difference in a single byte or character will cause the rows to be treated as unique.

The implementation leverages the power of set difference using `exceptAll()`. We first generate the unique subset of the data by calling `df.dropDuplicates()` without any arguments. This unique subset contains only one instance of each fully identical row. We then subtract this unique subset from the original DataFrame (df). The remainder is the set of rows that were redundant--the duplicates we were looking for.

This technique is statistically robust for identifying true data redundancy. It is critical to ensure that auxiliary columns, such as auto-generated primary keys or timestamps of insertion, are either excluded from the DataFrame before this operation or verified not to mask true duplicates, although in this standard implementation, all columns participate in the uniqueness check.

Implementation Example: Duplicates Across All Columns

We execute the required PySpark syntax to identify and display rows that possess identical values across all three columns: `team`, `position`, and `points`. This demonstrates the basic, powerful application of using the set difference logic to isolate fully duplicated records.

#display rows that have duplicate values across all columns
df.exceptAll()(df.dropDuplicates()).show()

```
+----+--------+------+
|team|position|points|
+----+--------+------+
| A| Forward| 22|
| B| Guard| 14|
+----+--------+------+
```

We can see that there are two rows that are exact duplicates of other rows in the DataFrame, confirming the identification of the second instance of the fully redundant records.

Method 2: Pinpointing Duplicates Based on Specific Columns

In many real-world data quality scenarios, defining a duplicate record involves only a subset of columns, often referred to as a candidate key. For instance, if we are tracking player statistics, a row should be considered redundant if the combination of `team` and `position` is repeated, regardless of the `points` scored in that specific instance. This selective approach is crucial for mastering data normalization and integrity checks.

To implement this selective duplicate identification in PySpark, we pass a list of the key columns--in our case, --to the `dropDuplicates()` function. When given these specific columns, the function will generate a unique DataFrame where no two rows share the same combination of values in the defined list. It effectively aggregates the data based on these keys and discards subsequent occurrences.

We then reuse the robust `exceptAll()` logic. The original DataFrame (df) contains all records, including those that share a key combination. The unique key DataFrame (df.dropDuplicates()) contains only one instance for each key combination. By subtracting the latter from the former, the resulting set precisely isolates all rows that represent the secondary, tertiary, or subsequent occurrences for that specific combination of `team` and `position`, fulfilling the requirement for targeted duplicate identification.

Implementation Example: Duplicates Based on Key Columns

The following syntax demonstrates how to specifically target rows that duplicate the combination of

values found in the `team` and `position` columns. Notice how the logic captures duplicates that were missed by the previous, full-row check because their 'points' values differed.

#display rows that have duplicate values across 'team' and 'position' columns
df.exceptAll(df.dropDuplicates()).show()

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

The resulting DataFrame contains only the rows identified as duplicates based on the shared values across both the **team** and **position** columns, fulfilling the objective of targeted duplicate identification.

Alternative Strategy: Counting Duplicate Occurrences

While the `exceptAll()` method provides the actual duplicate rows, sometimes the requirement is simply to measure the extent of the duplication--that is, how many times a particular key combination appears. This is achieved through aggregation functions, offering a powerful alternative or supplement to row extraction.

This technique relies on the `groupBy()` transformation, followed by the `count()` aggregation. We select the columns we wish to check for uniqueness (the key columns), group the DataFrame based on these columns, and then count the number of rows within each group. Any group having a count greater than one indicates a duplicate occurrence based on the selected key.

To isolate only the duplicate combinations, we filter the results of the grouping operation, retaining only those records where the resulting count column is greater than 1. This method does not return the actual duplicate rows themselves, but rather provides a highly summarized report of the problematic key combinations and their total frequencies in the dataset. This is highly efficient for data quality monitoring and reporting dashboards.

Example syntax for counting duplicates based on 'team' and 'position':

 # Count the frequency of each (team, position) combination
from pyspark.sql.functions import col

```
df.groupBy('team', 'position').count().filter(col('count') > 1).show()

+----+--------+-----+
|team|position|count|
+----+--------+-----+
| B| Guard| 2|
| A| Forward| 2|
| A| Guard| 2|
+----+--------+-----+
```

This output clearly shows that three key combinations have duplicates, quantifying the redundancy (two occurrences each), thereby providing actionable metrics regarding data quality issues without needing to extract every single duplicate row.

Conclusion and Further Reading

PySpark provides robust, scalable tools for managing data integrity. Whether you require identification of exact row matches using the generalized `dropDuplicates()` approach or need precise counting via `groupBy()`, the framework is optimized for handling these challenges efficiently in a distributed computing environment. The key to success lies in understanding how set operations like `exceptAll()` interact with the uniqueness definitions provided by `dropDuplicates()`.

These techniques empower data professionals to ensure the highest level of data quality, making derived insights and machine learning models built on the cleansed data far more reliable and trustworthy.

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