Learning PySpark: Identifying Duplicate Rows in DataFrames

Authored by

Mohammed looti

November 11, 2025

The Importance of Identifying Duplicate Records

The process of data cleaning is a foundational step in any robust data pipeline, especially when working with Big Data environments utilizing tools like PySpark DataFrames. Duplicate records pose significant threats to data integrity, often leading to skewed statistical results, inaccurate model training, and wasted computational resources. In the context of large-scale distributed processing, efficiently identifying and managing duplicates is paramount for maintaining reliable analysis.

A duplicate row can be defined in several ways: it might be an exact replica across all columns, or it might represent an unnecessary replication based on a subset of key identifier columns. PySpark offers powerful, distributed methods to handle both scenarios, leveraging optimized operations that work seamlessly across clusters. This guide details the two most common and effective techniques for pinpointing these problematic rows within a large dataset, ensuring the underlying data remains pristine and trustworthy for subsequent operations.

We will explore a highly efficient strategy that involves comparing the original DataFrame against a deduplicated version of itself. This strategy relies on two core PySpark DataFrame methods: `dropDuplicates()` and `exceptAll()`. Understanding how these functions interact is crucial for mastering duplicate identification in a distributed context.

Understanding the PySpark Approach to Duplication

In PySpark DataFrames, finding duplicates is not as simple as iterating through rows, which would be prohibitively slow in a distributed setting. Instead, we utilize set operations. The fundamental principle is to identify the difference between the complete dataset and the dataset where only unique rows are retained. The resulting difference set must, by definition, contain only the rows that had duplicates in the original dataset.

The first key function is `dropDuplicates()`. When applied to a DataFrame, this function returns a new DataFrame containing only the unique rows, retaining one instance of any row that was previously duplicated. The second critical function is `exceptAll()`. This function performs a set difference, returning the rows in the first DataFrame that are not present in the second DataFrame. Unlike the older `except()` method, `exceptAll()` respects the number of times a duplicate row appears, which is essential for accurately isolating the non-unique instances.

By applying `exceptAll()` to the original DataFrame using the deduplicated DataFrame as the argument, we effectively isolate all rows that were removed during the deduplication process--meaning the rows that were truly duplicates. There are two primary use cases for this approach, differentiated by whether the definition of a duplicate spans all columns or is restricted to a selected subset of columns.

Method 1: Identifying Duplicates Across All Columns

The most stringent definition of a duplicate requires that two rows are identical across every single column. This is often necessary when ensuring transactional integrity or preventing data entry errors where an entire record was accidentally entered twice. To implement this check, we use the `dropDuplicates()` method without passing any column arguments. When no arguments are provided, PySpark assumes that the uniqueness check must apply to all available columns.

The resulting expression, `df.exceptAll(df.dropDuplicates())`, is a highly optimized way to filter down to the exact duplicate instances. The first DataFrame (the original `df`) contains all rows, while the second (`df.dropDuplicates()`) contains only the single, unique representations. The output of `exceptAll()` will therefore return every row that occurred two or more times, effectively showing the duplicate count minus one unique instance for each set of identical rows. This provides a clear, actionable list of records requiring attention.

Here is the syntax for finding rows that are exact duplicates across all fields in the DataFrame:

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

Method 2: Targeting Duplicates Based on Specific Columns

Often, a record is considered a duplicate if a combination of certain key attributes--such as 'user ID' and 'timestamp'--is repeated, even if other non-key columns (like 'session details' or 'points earned') differ. In such cases, enforcing uniqueness only across a specified subset of columns is necessary. This approach allows us to define a logical key for the DataFrame, ensuring that records sharing the same values for these keys are flagged as duplicates, regardless of variation in non-key fields.

To achieve this, we modify the `dropDuplicates()` function by passing a list of the columns that

define uniqueness. For instance, if we consider records duplicates only if they share the same 'team' and 'position', we supply to the function. This action retains only one row for every unique combination of 'team' and 'position', keeping the other columns arbitrary for the retained row.

When this modified deduplicated DataFrame is used in the `exceptAll()` operation, the resulting output will highlight all rows that belong to a group defined by the specified column subset that had more than one occurrence. This powerful technique is crucial for tasks like identifying multiple entries for the same user ID or determining redundant entries in master data tables.

Below is the syntax to find rows that share identical values across the specified 'team' and 'position' columns:

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

Setting Up the Example PySpark DataFrame

To demonstrate the practical application of these two methods, we will first create a sample PySpark DataFrame. This dataset simulates player statistics, containing potential duplicates based on both all columns (exact matches) and specific columns (team and position matches). The examples below illustrate how Spark handles the identification of these duplicate rows in a structured manner.

The following code block initializes a SparkSession, defines the sample data (which includes several intentional duplicates), sets the schema, and constructs the DataFrame. Viewing the resulting DataFrame allows us to visually anticipate which rows should be flagged as duplicates in the subsequent examples. Note that rows 3 and 4 are exact duplicates ('A', 'Forward', 22), and rows 5 and 6 are also exact duplicates ('B', 'Guard', 14).

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

Example 1: Finding Rows with Duplicate Values Across All Columns

Using the first method, we instruct PySpark to treat two rows as duplicates only if every value across the 'team', 'position', and 'points' columns is identical. The operation `df.dropDuplicates()` will reduce the DataFrame size by retaining only one instance of the fully identical rows.

When `exceptAll()` compares the original 8-row DataFrame against the deduplicated version (which contains 6 unique rows), it isolates the two rows that were removed--the excess copies. These excess copies represent the true duplicates in the dataset.

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

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

As demonstrated by the output, only two rows are returned. These are the second instances of the ('A', 'Forward', 22) record and the second instance of the ('B', 'Guard', 14) record. This confirms that these were the only rows that were exact duplicates of other rows across all columns in the original DataFrame.

Example 2: Finding Rows with Duplicate Values Across Specific Columns

In this scenario, we adjust our definition of a duplicate. We are now interested in any row where the combination of the **team** and **position** fields is repeated, regardless of the 'points' value. This is useful for analyzing group distribution or ensuring that aggregated stats are not double-counted based on a key combination.

By using `df.dropDuplicates()`, we reduce the DataFrame to contain only one row for each unique (team, position) pairing. For example, since Team A has two 'Guard' records (11 points and 8 points), the deduplication process retains only one of them. Similarly, Team B has two 'Guard' records (14 points and 14 points), and one is retained.

The subsequent `exceptAll()` operation then reveals all the rows that were discarded because they shared the same key combination ('team' and 'position') as another record.

#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 now contains four rows. These are the secondary entries for each (team, position) group: the second 'Guard' for Team A, the second 'Forward' for Team A, the second 'Guard' for Team B, and the second 'Forward' for Team B. This successfully identifies all records that violate the uniqueness constraint defined by the 'team' and 'position' columns, regardless of the value in the 'points' column. This flexibility allows developers to precisely define what constitutes a duplicate based on business logic.

Exploring Additional Resources for Data Integrity

Mastering duplicate identification is a critical step in advanced data cleaning workflows. While the `exceptAll()` and `dropDuplicates()` combination is powerful for finding duplicates, other complementary techniques in PySpark can be used to handle them--such as using window functions to assign row numbers and filter based on those, or using aggregation functions like `count()` to identify groups with more than one entry.

For those looking to expand their knowledge beyond simple identification, exploring Spark's documentation on aggregation, grouping, and advanced set operations is highly recommended. These tools offer further control over how duplicates are treated, whether they need to be deleted entirely, merged based on specific criteria, or logged for audit purposes.

The following tutorials explain how to perform other common tasks in PySpark, contributing to a more comprehensive understanding of large-scale data manipulation and integrity management: