PySpark: Drop Duplicate Rows from DataFrame

Authored by

Mohammed looti

November 11, 2025

Introduction to Handling Duplicates in PySpark

Managing data quality is a critical step in any data processing pipeline. One of the most common issues data engineers face is the presence of **duplicate rows**, which can skew analytical results, corrupt training models, and inflate storage requirements unnecessarily. Fortunately, the **PySpark** library, the Python API for Apache Spark, provides robust and highly optimized methods for identifying and eliminating these redundant records efficiently, even when dealing with massive, distributed datasets. Achieving data integrity is paramount, and removing duplicate entries is a foundational step in that process.

The primary function used for this operation is the `dropDuplicates()` method, which is applied directly to a **DataFrame** object. This versatile method offers flexible control, allowing users to define the precise scope of the duplication check--whether the criteria for uniqueness should span across all columns or be restricted to a specified subset of key attributes. Understanding how to leverage this function effectively is essential for maintaining clean and reliable data within the demanding environment of the Spark ecosystem.

This guide will systematically explore the three fundamental ways to utilize the `dropDuplicates()` function to cleanse your data, demonstrating each approach with clear, executable code examples based on a consistent input DataFrame. These methods range from the simplest, which checks for exact matches across the entire row, to more nuanced approaches that target specific combinations of columns to define the necessary uniqueness criteria, offering maximum flexibility for diverse data cleaning needs.

Method 1: Identifying Duplicates Across All Columns

The most straightforward application of the `dropDuplicates()` method is to search for rows that are identical across every single column. When called without providing any arguments, PySpark performs a comprehensive comparison, evaluating all values in a row against all other rows in the distributed DataFrame. If an exact match is found for every field, the subsequent occurrences of that row are considered redundant and are consequently removed, retaining only the first instance encountered during processing.

This technique is generally recommended when you need to enforce strict, comprehensive record uniqueness based on the entirety of the dataset structure. For instance, if you are aggregating logs, auditing transaction records, or validating configurations where every single field must be distinct to ensure data validity, this default behavior provides a high standard of data integrity preservation. The following concise syntax illustrates how to initiate this complete-row deduplication operation:

 # drop rows that have duplicate values across all columns

```
df_new = df.dropDuplicates()
```

It is crucial to remember that PySpark DataFrames operate lazily. The transformation is defined immediately upon calling `dropDuplicates()`, but the actual computation and execution of the deduplication process across the cluster only occur when an action (such as `show()`, `collect()`, or writing to storage) is invoked. This design allows Spark to optimize the execution plan by combining this deduplication operation with other transformations for maximum performance.

Method 2: Filtering Duplicates Based on Specific Columns

In many real-world scenarios, data uniqueness is defined not by the entire row, but by a combination of specific identifying columns, often referred to as a composite key. For example, in a personnel dataset, you might consider a row duplicate only if the combination of `employee_id` and `hire_date` is repeated, irrespective of what the `department` or `salary` fields hold. The `dropDuplicates()` method is designed to accommodate this precise requirement by accepting a list of column names as an argument.

When you specify a list of columns, PySpark intelligently restricts the duplication check solely to those designated fields. If two or more rows possess identical values exclusively within the specified subset of columns, all but the first occurrence of that group are removed. This selective duplication check is vital for tasks such as cleaning datasets based on primary keys or natural keys where other, less critical fields might legitimately vary (e.g., status flags or timestamps that update frequently).

To implement this targeted deduplication, you simply pass the names of the columns you wish to use as the uniqueness criteria within a standard Python list to the function call. Using the variables from our example dataset, if we wanted to find records that shared the same 'team' and 'position', the operational code would be structured as follows:

```
# drop rows that have duplicate values across 'team' and 'position' columns
df_new = df.dropDuplicates()
```

This level of control ensures that data analysts can retain valuable, non-key information in other columns while confidently enforcing the uniqueness constraint required across the core identifying attributes of the dataset.

Method 3: Ensuring Uniqueness Within a Single Column

The third approach is a specialized instance of the selective deduplication method, focusing on enforcing uniqueness across just one column. This is often necessary when the goal is to retain a

single, representative record for each unique entity identifier, such as ensuring that only one record exists for each unique 'user_id' or 'product_SKU'. By passing a list containing only one column name, PySpark guarantees that the resulting DataFrame will contain no repeated values in that specific field.

It is important to fully grasp the implication of this operation: when multiple rows share the same value in the target column (e.g., several rows for Team 'A'), PySpark retains the first row it encounters associated with that value and discards all subsequent rows, regardless of the values in the other columns. This transformation effectively reduces the DataFrame to a set of unique identifiers based on the chosen column, retaining associated data arbitrarily from the first instance found within the cluster's processing order.

For scenarios where the order of records matters, or where you need to define precisely which "first" record is kept (e.g., the one with the highest score or the latest timestamp), advanced techniques involving sorting (`orderBy`) or **Window functions** should be used before the deduplication step. However, for a simple, direct unique filter on one column, the syntax remains remarkably concise:

 # drop rows that have duplicate values in 'team' column
df_new = df.dropDuplicates()

This method is frequently employed during initial data exploration, summary generation, or aggregation steps where the primary focus is on achieving a list of unique entities present in the data rather than preserving every detailed, potentially redundant record.

Prerequisites: Setting Up the PySpark DataFrame

Before diving into the practical demonstrations of deduplication, it is essential to establish the working environment and the input data. All subsequent examples rely on initializing a **SparkSession**, which serves as the fundamental entry point to utilizing all Spark functionality, and defining a source DataFrame containing known duplicate records to observe the precise effects of the transformations being applied.

The sample DataFrame we will utilize represents basic sports data, tracking teams, player positions, and points scored. This dataset is intentionally constructed to include several types of duplicate entries, allowing us to clearly illustrate the functional differences between the three distinct `dropDuplicates()` methods. Analyzing the output of the `show()` command below reveals the specific rows that are exact, complete duplicates (e.g., rows 3 and 4; rows 5 and 6) and those that are only partial duplicates based on key columns (e.g., multiple 'Guard' positions for Team 'A' with varying point scores).

The following code snippet demonstrates the standard procedure for creating this sample **DataFrame** in PySpark, defining the schema using relevant column names, and subsequently displaying the original structure for reference and verification purposes:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

# Define data structure containing duplicates
data = ,
,
,
,
,
,
]

# Define column names for clarity
columns =

# Create dataframe using data and column names
df = spark.createDataFrame(data, columns)

# View the original 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|
+----+--------+------+
```

With the DataFrame successfully initialized and its initial structure understood, we can now proceed to execute the various deduplication methods and carefully analyze the resulting output

DataFrames to see how each method modifies the underlying data structure.

Example 1: Drop Rows with Duplicate Values Across All Columns

In this first practical application, we use the simplest form of the `dropDuplicates()` method without providing any parameters to clean the input DataFrame `df`. This method ensures that only rows where all three columns ('team', 'position', and 'points') are perfectly identical are identified and removed. We are exclusively targeting rows that are exact, byte-for-byte copies of preceding entries.

By examining the original dataset established in the setup phase, we identified two distinct sets of complete duplicates: first, the entry appears twice; and second, the entry also appears twice. When the function executes, it operates on the principle of retaining the first occurrence of each unique row while discarding all subsequent identical copies across the distributed partitions.

The execution of the following code snippet demonstrates the removal of exactly two rows, confirming that only the perfectly matched duplicates were eliminated. Importantly, rows that differed in even a single column--such as the two 'A', 'Guard' rows which had different 'points' values--remain fully intact, as they do not constitute a complete row duplicate.

 # drop rows that have duplicate values across all columns
df_new = df.dropDuplicates()

# view DataFrame without duplicates
df_new.show()

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

The resulting DataFrame successfully retains only 6 unique records out of the original eight, confirming that two redundant rows were dropped. This confirms that the default behavior of `dropDuplicates()` is highly effective for removing redundant, exact copies of records across the entire structure.

Example 2: Drop Rows with Duplicate Values Across Specific Columns

In this second practical application, we shift our focus to targeted deduplication by instructing PySpark to identify and remove rows that share the same combination of the 'team' and 'position' columns, entirely ignoring the variance in the 'points' value. This is a typical requirement when preparing standardized team rosters or ensuring that each unique combination of team and position is represented only once in the resulting analysis dataset.

By passing to the `dropDuplicates()` method, we force PySpark to treat any records sharing these two column values as duplicates. For instance, Team 'A' has two 'Guard' entries (11 points and 8 points). Since the critical 'team' and 'position' columns match for these two records, one of them will be arbitrarily dropped. Similarly, groups of duplicates are identified and reduced for the 'A' 'Forward', 'B' 'Guard', and 'B' 'Forward' combinations based on this specific two-column criteria.

Executing the command below results in a DataFrame where every combination of the specified columns is guaranteed to be unique. It is worth noting that the 'points' column value for the retained row is determined by which row was processed first in the distributed environment; it simply keeps the associated data from the first instance encountered in that unique group.

 # drop rows that have duplicate values across 'team' and 'position' columns
df_new = df.dropDuplicates()

# view DataFrame without duplicates
df_new.show()

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

The resulting DataFrame contains exactly four rows, clearly representing the four unique combinations of team and position (A/Guard, A/Forward, B/Guard, and B/Forward). This powerfully demonstrates the effectiveness and necessity of targeted deduplication based on a user-defined, selective subset of key attributes.

Example 3: Drop Rows with Duplicate Values in One Specific Column

This final example illustrates the most reductive form of deduplication, where we enforce uniqueness based solely on the 'team' column. The objective here is simple: to retain only one single representative row for each unique team identifier present in the entire dataset. This action is extremely useful in preliminary analysis because it drastically reduces the dataset size, prioritizing the unique entity (the team) over associated detailed, potentially redundant data (position and points).

When a list containing only is passed to `dropDuplicates()`, the function scans the DataFrame and ensures that the final 'team' column contains no repeated values. Since our sample dataset only contains two unique team values ('A' and 'B'), the resulting DataFrame must contain exactly two rows. All other rows associated with 'A' and 'B' are discarded, preserving only the first instance of each team encountered during the processing sequence.

As noted previously, the selection of which row is kept is arbitrary based on the internal processing order. If specific, critical data points (e.g., the row with the latest date or the maximum score) need to be retained, preliminary sorting using `orderBy()` must be performed prior to this deduplication step. For the sake of demonstrating the pure, single-column `dropDuplicates()` functionality, we execute the following operation:

```
# drop rows that have duplicate values in 'team' column
df_new = df.dropDuplicates()

# view DataFrame without duplicates
df_new.show()
```

```
+----+--------+------+
|team|position|points|
+----+--------+------+
| A| Guard| 11|
| B| Guard| 14|
+----+--------+------+
```

The output confirms that only one row remains for Team 'A' and one row for Team 'B', successfully enforcing absolute uniqueness within the 'team' column as intended. Notice that the associated 'position' and 'points' values are those belonging to the specific first row encountered for that respective team identifier.

Key Considerations and Advanced Mechanics

Understanding the underlying mechanical behavior of PySpark's deduplication is essential for ensuring predictable data outcomes, especially in distributed environments. When the `dropDuplicates()` transformation identifies multiple redundant rows based on the defined criteria (either all columns or a subset), it adheres strictly to a fundamental rule: only the first duplicate row is kept in the resulting DataFrame while all subsequent duplicate rows are dropped from the partitions.

A critical characteristic of **PySpark** DataFrames is that they do not inherently guarantee order unless an explicit `orderBy()` transformation is applied. Because Spark processes data across numerous distributed partitions simultaneously, the exact definition of the "first" row is determined by the internal ordering within the partitions as they are processed. If strict, deterministic control over which record is retained (e.g., keeping the record with the maximum score, or the entry with the latest timestamp) is required, it is imperative to sort the DataFrame first using `orderBy()` before calling `dropDuplicates()`, or alternatively, utilizing more complex **Window functions** coupled with ranking to precisely rank and filter records based on specific business logic.

In summary, the `dropDuplicates()` method in PySpark offers an extremely efficient and versatile tool for maintaining data hygiene. Whether your requirement is a comprehensive exact match check across all attributes or a selective filter based on a few key columns, mastering these three approaches ensures that your data remains accurate, structurally sound, and optimally prepared for resource-intensive downstream processing and advanced analytics.

Additional Resources

To further enhance your proficiency in distributed data manipulation using the PySpark framework, consider exploring the following advanced topics and tutorials which cover other common data engineering tasks vital for production environments:

Advanced filtering and selection techniques using the `filter()` and `where()` methods in PySpark DataFrames.

Techniques for efficiently joining multiple DataFrames using different join strategies and types.

Comprehensive strategies for handling missing or corrupt data (null values) using PySpark's imputation and dropping functionalities.