PySpark: Drop Duplicate Rows from DataFrame

Authored by

stats writer

November 16, 2025

When working with large-scale data processing using PySpark, handling data integrity is paramount. One of the most frequent tasks is identifying and removing redundant records, commonly known as duplicate rows, from a DataFrame. Duplicate data can skew analytical results, inflate storage costs, and complicate subsequent data transformation steps. PySpark provides the highly efficient `dropDuplicates()` transformation to address this issue, offering flexibility to define the scope of the duplication check. This tutorial explores the three primary methods for leveraging this powerful function to ensure your data remains clean and accurate for Big Data analysis.

Understanding PySpark's dropDuplicates Transformation

The `dropDuplicates()` method is a critical component of the PySpark API, designed specifically for filtering records within a distributed computing environment. Unlike traditional pandas operations, this method is optimized to run efficiently across clusters, making it suitable for massive datasets where memory limitations are a concern. The function operates by comparing records based on the values in specified columns and retaining only the first unique record it encounters, dropping all subsequent identical matches. Understanding this retention mechanism--always keeping the first instance--is crucial for maintaining data consistency, especially if the internal ordering of rows matters before the shuffle process.

The implementation of this operation often involves shuffling data across the cluster nodes. When no specific columns are provided, PySpark must compare every single column value across all rows, which can be computationally intensive and demands high network bandwidth due to the necessary data redistribution. However, by specifying a subset of columns, the scope of comparison is significantly narrowed, leading to faster execution times and reduced resource consumption, thereby improving the overall performance of your **Apache Spark** jobs.

Method 1: Identifying Duplicates Across All Columns

The simplest application of the duplication removal mechanism involves checking for exact matches across every column within the DataFrame. This method is used when you need absolute certainty that every retained row is completely unique based on its entire set of attributes. By calling `df.dropDuplicates()` without any arguments, the function assumes that all columns should participate in the duplicate detection logic.

This approach is highly effective for datasets where every column contributes to the definition of a unique record, such as transaction logs or complete entity records. If two rows are identified where the values in Column A, Column B, and Column C are identical, one of those rows will be retained, and the other will be discarded. This is the default and safest method when you require perfect data singularity throughout the dataset, ensuring no two rows are ever identical across all defined fields.

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

Method 2: Selective Duplicate Removal Based on Specific Columns

In many real-world scenarios, a row might be considered a duplicate even if not all column values match. Often, uniqueness is defined by a specific combination of key columns, such as a user ID and a timestamp, or in this case, a team identifier and a position. To implement this selective criteria, the `dropDuplicates()` function accepts a list of column names as an argument. PySpark will then identify and remove rows that share identical values exclusively across the columns specified in this list.

This feature is immensely valuable when dealing with hierarchical data or records containing ancillary information that is not critical for defining uniqueness. For instance, if you are analyzing player stats, you might want to ensure that for every unique **team** and **position** combination, only one record remains. Any additional columns, like 'points' or 'rebounds', will not factor into the decision of whether a row is a duplicate, meaning that PySpark will simply keep the first row it encounters for that unique key combination, regardless of the values in the non-key columns.

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

Method 3: Enforcing Uniqueness on a Single Key Column

The third utilization involves enforcing a strict uniqueness constraint on a single column. While less common for detailed analytics, this method is crucial for tasks like creating unique dimension tables or obtaining a list of distinct entities. By passing only one column name to the `dropDuplicates()` function, you are instructing PySpark to retain only one instance of each value found in that specific column, effectively condensing the DataFrame down to one representative row per unique value in the key field.

For example, if you execute this operation targeting the 'team' column, the resulting DataFrame will contain only one row for each unique team name present in the original dataset. All other columns associated with that team's subsequent occurrences are discarded. It is important to remember that since PySpark retains the first row encountered, the non-key column values (e.g., 'position' or 'points') retained for the unique team entry will be arbitrarily selected based on the internal ordering of the distributed data partitions before the duplication check.

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

Setting Up the Example PySpark DataFrame

To clearly demonstrate the functionality of each method, we will utilize a sample PySpark DataFrame representing player statistics. This dataset intentionally contains several duplicate records that will be targeted in the following examples. We first initiate a Spark Session and define the structure and content of our sample data before creating and displaying the DataFrame. Note the presence of duplicate entries, specifically the row (A, Forward, 22) and (B, Guard, 14), which will be used to illustrate the filtering logic.

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: Dropping Duplicates Across All Columns

In this initial example, we apply the **`dropDuplicates()`** transformation without specifying any subset of columns. The objective is to identify and remove any rows that are perfectly identical across all three fields: `team`, `position`, and `points`. By executing this function globally, we aim for the strictest form of uniqueness enforcement within the distributed environment provided by PySpark.

Upon reviewing the sample data, we observe two pairs of rows that are exact duplicates: the fourth row (A, Forward, 22) duplicates the third row, and the sixth row (B, Guard, 14) duplicates the fifth row. Therefore, we expect precisely two rows to be dropped from the original DataFrame of eight rows, resulting in a new DataFrame containing six unique records. The resulting output confirms this expectation, showcasing only those records that differ in at least one column value from all other retained records.

#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
+----+--------+------+

As predicted, a total of two rows were successfully dropped from the original DataFrame. Note that the rows (A, Guard, 11) and (A, Guard, 8) are both retained because their `points` values differ, despite the `team` and `position` being identical, thus not qualifying as full duplicates.

Example 2: Dropping Duplicates Based on Team and Position

In this second scenario, our definition of a duplicate is based exclusively on the combination of `team` and `position`. We pass a list containing to the `dropDuplicates()` function, instructing PySpark to treat any records sharing the same team and position as redundant. This means we are only interested in obtaining one representative row for each unique role within each team, regardless of the associated 'points' value.

The original dataset contains multiple records for 'A' Guard, 'A' Forward, 'B' Guard, and 'B' Forward. By running this operation, PySpark retains only the first instance it encounters for each unique combination (A/Guard, A/Forward, B/Guard, B/Forward). The resulting DataFrame is significantly smaller, guaranteeing that no two rows share the same values for the specified key columns, thereby cleaning the dataset based on entity roles.

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

Notice that the resulting DataFrame has exactly four rows, corresponding to the four unique key combinations. Furthermore, observe that for A Guard, the row with 11 points was retained (as it appeared first in the original data), while the row with 8 points was dropped, illustrating the implicit "first-in, kept" principle of `dropDuplicates()`. This method is instrumental when aggregating data or preparing a dataset for joins where uniqueness based on specific entities is mandatory.

Example 3: Enforcing Uniqueness on the Team Column Only

Our final example demonstrates the most restrictive form of duplicate removal by focusing solely on the `team` column. The goal here is to reduce the DataFrame to contain only one representative row for each distinct team identifier present in the dataset. This action is powerful for quickly

generating a list of unique keys or for preparatory steps where all team-specific details are required to be condensed into a single row, albeit arbitrarily selected from the records available for that team.

When we execute **`df.dropDuplicates()`**, PySpark identifies that there are only two unique team identifiers: 'A' and 'B'. Consequently, all rows associated with subsequent occurrences of 'A' and 'B' are immediately dropped, leaving only the very first encountered row for Team A and the very first encountered row for Team B. This severe reduction results in a DataFrame where the retained non-key column values (`position` and `points`) are those belonging to the retained record.

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

Notice that the resulting DataFrame has no rows with duplicate values in the **team** column. For Team A, the retained row is (A, Guard, 11), and for Team B, the retained row is (B, Guard, 14). This highlights an important consideration: when enforcing uniqueness on a subset of columns, the values retained in the non-key columns are those belonging to the specific row PySpark happened to keep, which is generally the first record based on internal execution order.

Important Considerations for Row Retention Policy

A fundamental operational aspect of the `dropDuplicates()` transformation in PySpark relates to row retention policy. The function generally keeps the first encountered record that satisfies the uniqueness criteria and discards all subsequent matching records. This behavior is crucial because it means the retained row is not selected based on any aggregate function (like maximum points or latest timestamp) but purely based on its position in the distributed data stream before the shuffling process.

Therefore, if you need to ensure that a specific version of the duplicate row is retained (e.g., the record with the highest 'points' value, or the row with the latest 'timestamp'), it is essential to preprocess the DataFrame. This preprocessing step usually involves sorting the DataFrame by the

desired retention criteria before applying **`dropDuplicates()`**. For instance, to keep the row with the maximum points for each team and position, you would sort the DataFrame by `team`, `position`, and then descending `points`, ensuring the preferred record appears first before the de-duplication operation is executed.

Note: When duplicate rows are identified based on the specified criteria (all columns or a subset of columns), only the first duplicate row is kept in the DataFrame while all other duplicate rows are dropped. This implicit retention policy must be factored into any data quality pipeline involving de-duplication using **PySpark**.