How to Remove Duplicate Rows from a PySpark DataFrame

Authored by

stats writer

February 9, 2026

The process of eliminating redundant data is fundamental to maintaining high-quality datasets. In PySpark, removing rows that contain identical values across specified columns--or all columns--is achieved efficiently using the native `dropDuplicates()` function. This operation is critical for effective data cleaning and essential for guaranteeing data integrity, especially when handling vast quantities of information characteristic of big data environments. Duplicate records can skew analytical results, inflate storage requirements, and reduce processing performance. Utilizing `dropDuplicates()` on a DataFrame ensures that subsequent analysis is based on unique, reliable data points, thereby significantly improving the accuracy and efficiency of downstream processing tasks within the PySpark framework.

PySpark: Efficiently Drop Duplicate Rows from a DataFrame

Understanding Data Deduplication Strategies in PySpark

Deduplication in PySpark offers flexibility, allowing developers to define the scope of the uniqueness check. Depending on whether you require rows to be unique across all attributes or only a subset of key identifiers, there are three primary strategies implemented using the versatile `dropDuplicates()` method.

Selecting the correct approach is crucial for achieving the desired outcome. If you are dealing with records where minor differences (like timestamps or derived values) are expected, but the core identifier set must be unique, you will specify columns. Conversely, if true replication of the entire record is the target, you use the function without arguments.

The following sections detail these three common methods for removing redundant rows from a DataFrame:

Method 1: Dropping Duplicates Across All Columns

This is the simplest and most encompassing way to perform deduplication. When the `dropDuplicates()` function is invoked without any parameters, PySpark compares every single column value in one row against every column value in all other rows. Only if two rows are byte-for-byte identical will one of them be marked for removal. This is the default behavior and is typically used when seeking true record uniqueness across the entire record schema.

 # Invokes dropDuplicates() without arguments to check all columns for exact matches.
df_new = df.dropDuplicates()

Method 2: Dropping Duplicates Across Specific Columns

Often, a data scientist needs to enforce uniqueness based on a defined set of primary keys, even if non-key columns might differ slightly across records. By passing a list of column names to `dropDuplicates()`, you instruct the DataFrame to only compare the intersection of values within those specified columns. If the combination of values in the defined column subset is identical for two or more rows, the subsequent rows are treated as duplicates and dropped. This is the standard procedure for composite key uniqueness enforcement.

 # Instructs Spark to drop rows where the combination of 'team' and 'position' columns are identical.
df_new = df.dropDuplicates()

Method 3: Dropping Duplicates Based on a Single Column

Using a single column parameter enforces that every entry in that specified column must be unique across the entire DataFrame. This method is highly effective for ensuring that an assumed unique identifier, such as a user ID or a product code, truly holds a one-to-one mapping across the dataset. When duplicates are found in the specified column, only the first observed row corresponding to that value is retained, which means many associated columns' values will be discarded for that entity.

 # Ensures that only one row per unique value in the 'team' column is retained.
df_new = df.dropDuplicates()

Setting Up the Demonstration DataFrame

To effectively demonstrate the results of each deduplication strategy, we first define a sample DataFrame. This dataset contains athlete statistics with columns `team`, `position`, and `points`, and intentionally includes several redundant records to showcase how the different methods process and clean the data. This setup requires importing the necessary `SparkSession` and defining the data and schema before execution.

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

# Define the raw data, containing intentional duplicates
data = ,
,
,

,
,
,
,
]

Define column headers
columns =

# Create the DataFrame using the defined data and schema
df = spark.createDataFrame(data, columns)

# View the initial DataFrame (8 rows total)
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: Eliminating Exact Row Duplicates (All Columns)

We execute the global deduplication strategy by calling `dropDuplicates()` without arguments. This operation strictly searches for rows that are identical in every single column. Based on the initial data, the rows (A, Forward, 22) and (B, Guard, 14) are exact matches that appear twice. This function will retain one instance of each unique record and discard the perfect duplicates, a necessary step for maintaining data cleaning standards.

 # 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 confirms that a total of two rows were dropped, specifically the redundant instances of (A, Forward, 22) and (B, Guard, 14). The remaining six rows are entirely unique across all three columns, demonstrating successful strict deduplication.

Example 2: Deduplication Based on Specified Key Columns

In this example, we define a composite key using the `team` and `position` columns. We instruct the function to retain only one record for every unique pairing of team and position, even if the `points` column differs. This is highly useful when we treat the team-position combination as a unique entity and wish to eliminate redundant statistical entries.

The original dataset contains four unique combinations for this key: ('A', 'Guard'), ('A', 'Forward'), ('B', 'Guard'), and ('B', 'Forward'). Since each combination appears twice, we expect four rows to be retained after deduplication.

```
# 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 team/position combinations. For instance, the 'A', 'Guard' record with 8 points and the 'B', 'Forward' record with 7 points were dropped because a preceding row shared the identical `team` and `position` values, illustrating the mechanism of selective deduplication based on key identifiers.

Example 3: Enforcing Uniqueness on a Single Column Identifier

This method focuses on ensuring absolute uniqueness within the `team` column. By executing `dropDuplicates()`, we enforce that only one row per team is retained, making this a useful operation for sampling or entity-level summarization. Since only two teams ('A' and 'B') exist in the dataset, the expected output is a DataFrame containing only two rows.

 # 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 resulting DataFrame, containing only two rows, confirms that all other records associated with Team A and Team B were dropped. This strict filtering ensures maximum data integrity for the primary identifier column. This process effectively converts the detailed transaction log into a list of unique entities.

Handling Row Retention and Non-Determinism

A critical consideration when performing deduplication in PySpark is the retention rule. When multiple duplicate rows are identified, only the first encountered row is kept in the DataFrame while all others are discarded. However, it is essential to remember that, by default, the Spark execution framework is distributed and parallelized, meaning the definition of the "first" row is often non-deterministic.

If you have a requirement to retain a specific duplicate record--for example, the row containing the maximum `points` or the latest timestamp--you must explicitly sort the DataFrame prior to calling `dropDuplicates()`. Without an explicit `orderBy()` operation, the retained row may vary across

different execution runs due to variations in partitioning and task ordering, potentially compromising data integrity or expected results in sensitive data cleaning pipelines.

Note: When duplicate rows are identified, only the first duplicate row is kept in the DataFrame while all other duplicate rows are dropped. This retention is subject to Spark's internal partitioning unless explicit ordering is applied beforehand.

Further PySpark Data Manipulation Tutorials

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