How to Remove Duplicate Rows from a PySpark DataFrame

Authored by

stats writer

January 1, 2026

Identifying and managing redundant records is a foundational aspect of ensuring robust Data Quality in any analysis project, especially when operating in large-scale environments characterized by Big Data. Within the Apache Spark ecosystem, utilizing the Python API via PySpark DataFrames provides powerful, distributed methods for data manipulation. Finding and isolating these duplicated rows is an essential preliminary step before executing operations like aggregations, complex joins, or training machine learning models, as redundant data can skew results and inflate processing times.

While the conventional approach to cleaning duplicates often involves simple removal using the `dropDuplicates()` method, a crucial requirement for comprehensive data governance is the ability to specifically identify and inspect the rows that constitute the duplication. This inspection allows data engineers to understand the origin and nature of the redundancy, enabling root cause analysis. PySpark offers an elegant and distributed pattern to achieve this isolation, relying on set difference operations performed at scale.

The strategy revolves around defining uniqueness--either across all attributes or based on a specific subset of key identifiers--and then leveraging two core DataFrame transformation methods. First, `dropDuplicates()` generates a unique representation of the data. Second, `exceptAll()` performs a precise set subtraction, retaining only the rows from the original set that were removed by the de-duplication process, thus isolating the specific records that were duplicates.

Conceptual Framework for PySpark Duplicate Isolation

When operating on distributed datasets, the primary challenge is not the identification logic itself, but executing that logic efficiently across numerous partitions. PySpark DataFrames handle this through optimized planning and shuffling. Conceptually, there are two primary classifications of duplication that users typically need to identify, which dictate the specific parameters used in the detection methods. These classifications are the basis for the two main methods explored in this guide.

The first method addresses rows that are **exact matches** across all columns, signifying literal copies of records. This is the most straightforward form of redundancy and is typically solved by calling `dropDuplicates()` without any arguments. The second method tackles **logical duplicates**, where the identity of the record is determined by a subset of key columns, allowing other non-key attributes (like timestamps or measures) to vary, yet still flagging the record as redundant based on the business definition of uniqueness.

In both scenarios, the mechanism for isolation remains consistent: we use the `exceptAll()` transformation. This function is vital because it preserves the count of duplicated rows. If the

original DataFrame (A) contains a row three times, and the de-duplicated DataFrame (B) contains it once, `A.exceptAll(B)` will correctly return that row twice, identifying both duplicated instances.

Method 1: Detecting Exact Duplicate Rows Across All Columns

This technique is designed to find rows that are perfectly identical to at least one other row in the DataFrame. It is the strictest form of duplicate detection and is highly effective for identifying data loaded multiple times or errors in data capture processes. Since we are interested in every single attribute, we do not pass any column arguments to the `dropDuplicates()` method.

The combination of methods is expressed concisely in the PySpark syntax below. The resulting DataFrame will contain only the records that were present in the original DataFrame but absent in the de-duplicated version--meaning only the redundant copies are returned.

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

Method 2: Detecting Logical Duplicate Rows Across Specific Columns

When the business requirement dictates that uniqueness is determined by a defined set of primary keys, we must employ selective duplication checks. This is common when dealing with transactional data where the combination of identifiers (e.g., user ID and purchase date) must be unique, even if other details (e.g., transaction IDs or specific amounts) might differ due to data inconsistencies.

To implement this selective check, we pass a list of column names--the key columns--into the `dropDuplicates()` method. PySpark then performs the uniqueness check solely based on the values in those specified columns. If a combination of values in the key columns is found more than once, it is marked as a duplicate, and all but the first instance are isolated by the `exceptAll()` method.

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

Establishing the PySpark Testing Environment

To properly illustrate both duplicate detection techniques, we must first initialize a Spark environment and define a sample dataset that includes intentional redundancies. The following example sets up a PySpark DataFrame using a list of lists, representing basketball player data with

columns for `team`, `position`, and `points`. This dataset allows us to observe both perfect duplicates (identical rows) and logical duplicates (identical key columns but different point values).

We begin by establishing the `SparkSession`, which is mandatory for executing any PySpark operations. We then structure our data and column definitions to ensure the resulting DataFrame is correctly typed and ready for transformation. Examining the output of `df.show()` before applying any logic confirms the structure and highlights the presence of duplicate instances for analysis.

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

```
+----+--------+------+
```

Observation of the source data reveals that the row `A, Forward, 22` appears twice, and the row `B, Guard, 14` also appears twice. These four rows (two sets of perfect duplicates) are the only exact matches in the dataset. However, if we look only at `team` and `position`, every combination in the DataFrame is repeated, suggesting four sets of logical duplicates.

Example 1: Analyzing Rows with Duplicate Values Across All Columns

We now execute Method 1, applying the logic to identify instances where the redundancy spans all columns. This test rigorously confirms which rows are exact copies of others, which is often crucial for validating source system integrity in Big Data ingestion pipelines.

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

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

The output correctly identifies the two redundant rows that were exact duplicates of other rows in the DataFrame. Specifically, the result displays the second occurrence of the `A, Forward, 22` record and the second occurrence of the `B, Guard, 14` record. If a row had appeared three times, the output would have shown two instances of that row, demonstrating the fidelity of `exceptAll()` in retaining all duplicated copies beyond the first unique instance.

Example 2: Analyzing Rows with Duplicate Values Across Specific Columns

For the second example, we shift our focus to logical duplicates, defining uniqueness based only on the `team` and `position` columns. This approach is more commonly used in practical scenarios where defining a primary key across all available columns is either impractical or incorrect from a data modeling perspective.

#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 four rows, which represent the second occurrence of each unique `(team, position)` pair found in the original dataset. For instance, the pair `A, Guard` appeared first with 11 points, and subsequently with 8 points. The row with 8 points is therefore returned as the duplicate record. Crucially, the row `A, Forward, 22` is returned again, even though it was already identified as an exact duplicate in Example 1, because its `(team, position)` key is duplicated relative to the first `A, Forward` row.

This outcome demonstrates the flexibility and precision of key-based duplication checking. By clearly defining the columns that must be unique, data professionals can selectively enforce Data Quality constraints that align precisely with the requirements of the downstream analytical tasks or business rules. The ability to isolate these specific conflicting records is paramount for effective data cleansing and transformation workflows in PySpark DataFrames.

Summary of Duplicate Detection Mechanisms

To summarize the techniques for identifying duplicates in PySpark DataFrames, the combination of `dropDuplicates()` and `exceptAll()` provides an efficient, distributed means of isolating the redundant records themselves. This approach is highly performant because it leverages PySpark's optimized set operations across the cluster.

The choice of whether to pass column names to `dropDuplicates()` is the defining factor that transitions the check from finding strict, row-level identical copies to finding logical duplicates based on user-defined key constraints. Both methods are indispensable tools in a data engineer's toolkit when preparing large datasets for reliable analysis.

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