Select Distinct Rows in PySpark (With Examples)

Authored by

stats writer

November 16, 2025

RECOMMENDED CITATION

stats writer (2025). *Select Distinct Rows in PySpark (With Examples)*. PSYCHOLOGICAL SCALES. Retrieved from https://scales.arabpsychology.com/?p=92538

ARABPSYCHOLOGY.COM

Understanding Data Duplication in Big Data

In large-scale data processing environments, handling data duplication is a critical task for ensuring the accuracy and integrity of analytical results. Duplicate records often arise during complex data ingestion pipelines, merging operations across various sources, or due to system retries. Utilizing PySpark, which is the Python API for the powerful Apache Spark engine, data engineers gain access to highly robust tools designed to efficiently identify and eliminate these redundant entries, especially when working with petabyte-scale datasets distributed across a computational cluster.

The core concept of selecting **distinct rows** is fundamental to effective data cleaning and preparation. When processing millions or even billions of records stored within a DataFrame, isolating truly unique observations is essential. If data preparation fails to remove duplicate entries, the consequences can include skewed statistical aggregates, severely inaccurate machine learning model training outcomes, and significant wastage of valuable computational time and storage resources. PySpark provides optimized methods that leverage Spark's distributed architecture to perform rapid deduplication, a capability that clearly differentiates it from traditional single-machine processing frameworks.

This comprehensive guide is designed to explore the specific methods available within PySpark to efficiently select either distinct rows or distinct values from specific columns within a DataFrame. We will provide detailed insights into the underlying mechanisms of these functions, meticulously illustrating their practical application using concrete code examples, and discussing the essential best practices necessary for achieving clean, reliable, and high-quality datasets using PySpark's powerful suite of transformations and actions.

The PySpark DataFrame: A Prerequisite for Distinct Operations

Before proceeding into the operational mechanics of deduplication, it is crucial to establish a strong understanding of the data structure we are manipulating: the PySpark DataFrame. A DataFrame constitutes a distributed collection of data organized into named columns, which is conceptually similar to a relational database table or a Pandas DataFrame in a local environment. PySpark DataFrames are architecturally built atop the resilient distributed dataset (RDD) foundation but provide significantly richer optimization capabilities and a much more user-friendly API, primarily through the use of the Spark SQL catalyst optimizer.

Effective interaction with DataFrames necessitates an active SparkSession, which serves as the primary gateway to all Spark functionality. The performance and efficiency of distinct operations are heavily reliant on how Spark internally handles data shuffling across the cluster. To determine uniqueness, Spark is inherently required to compare every record against others across the entire dataset, a task that often mandates substantial data movement and coordination. Therefore,

ARABPSYCHOLOGY.COM

gaining an understanding of the data partitioning strategy is vital for optimizing execution performance when performing distinct operations on DataFrames of enormous scale.

The methods used for identifying distinct records in PySpark are characterized by their simplicity and high readability, typically involving easily understood, chained function calls. These specific methods are categorized as wide transformations. This classification indicates that they inherently trigger a fundamental shuffling of data across the network to ensure that all identical records, regardless of their initial physical location across different nodes, are correctly grouped together for necessary comparison and ultimate elimination. This inherent requirement for global data coordination makes rigorous performance tuning a key consideration during the implementation of large-scale deduplication tasks.

Core Methods for Handling Distinct Values in PySpark

PySpark furnishes developers with three essential methods for managing distinct records, each tailored to satisfy different analytical requirements--whether the need is to retrieve the unique records themselves, isolate the unique values within a single column field, or simply obtain an accurate count of the unique records present in the dataset.

These primary methods include:

Method 1: Selecting Distinct Rows (`df.distinct()`): This represents the most comprehensive approach. It conducts a full scan of the entire DataFrame and returns a new DataFrame containing only those rows where the combined set of values across all columns is unique. This method is the definitive standard for meticulously cleaning a dataset by eliminating all instances of full-row duplicates.

Method 2: Selecting Distinct Values from Specific Columns (`df.select(col).distinct()`): This method strategically focuses on analyzing cardinality within a specific data subset. It operates by first selecting one or more specified columns and then applying the `distinct()` transformation exclusively to those fields, providing crucial insights into the unique categories or identifiers present within those restricted fields.

Method 3: Counting Distinct Rows (`df.distinct().count()`): This method involves applying the `distinct()` transformation immediately followed by the `count()` action. It is immensely valuable for data diagnostic purposes, enabling users to rapidly assess the overall level of duplication present in the dataset without requiring the costly step of retrieving and displaying the resulting unique DataFrame itself.

Each of these methods is implemented to leverage Spark's highly optimized execution planning capabilities. When executed, Spark diligently ensures that the data is partitioned and compared

ARABPSYCHOLOGY.COM

with maximum efficiency, thereby minimizing unnecessary network input/output (I/O) and computational processing overhead. The selection of the appropriate method should be entirely driven by the specific analytical goal: whether the task is full dataset cleaning, focused categorical analysis, or simply an assessment of the overall data quality.

Deep Dive: Method 1 - Selecting Distinct Rows

The `distinct()` transformation, when invoked directly on a DataFrame (as in `df.distinct()`), is specifically designed to preserve only those rows that exhibit absolute uniqueness across the totality of all existing columns. If any two rows share identical values for every single field, one instance of those rows will be systematically eliminated in the resulting output DataFrame. It is fundamentally important to grasp that this operation treats the entire row as a single atomic unit; any observed difference, regardless of how minor, in any column whatsoever prevents that row from being categorized as a duplicate.

The syntax required to perform this operation is remarkably simple, making high-level data cleaning in PySpark highly intuitive. When this transformation is triggered, Spark orchestrates a complex grouping and aggregation operation hidden beneath the surface. It first maps the data, then initiates a shuffle of the resulting intermediate data based on the full content of the row to ensure all identical rows are routed to the same executor node, and finally, it reduces the dataset by retaining only one representative row from each identified group of duplicates.

This method proves highly effective for rigorously ensuring data integrity, especially following complex join or union operations where unintended or systemic duplicates may have been introduced into the dataset. It is universally regarded as the gold standard when the primary objective is to construct a clean, completely non-redundant primary dataset that is ready for subsequent feature engineering or advanced analytical modeling. When implementing this function, users must remember that `distinct()` is a lazy transformation; the actual computation and resource expenditure only occur when an immediate action (such such as `show()` or `count()`) is called subsequently on the resulting structure.

Deep Dive: Method 2 - Isolating Unique Values in a Specific Column

Frequently, the operational requirement is not to find comprehensive unique rows but rather to identify the discrete categories or values present exclusively within a specific attribute column, such as discovering all distinct customer IDs, product SKUs, or team names. To expertly achieve this focused objective, we skillfully combine the `select()` method with the `distinct()` transformation. The precise sequence of these operations is critical: we first narrow the DataFrame down to include only the columns of explicit interest, and only then do we apply the deduplication logic.

ARABPSYCHOLOGY.COM

The standard structure for this operation is articulated as `df.select('column_name').distinct()`. By intentionally selecting only a single column, the resulting DataFrame is streamlined into what is effectively a single-column list of unique values. This approach offers a significant performance advantage over executing the full `df.distinct()` if the analytical interest is centered solely on the uniqueness of a single attribute, primarily because Spark is then only required to shuffle data based on the content of that specific column, drastically reducing the volume of data transferred across the network infrastructure.

This technique is routinely employed during exploratory data analysis (EDA), empowering data scientists to swiftly verify the cardinality of key categorical variables or to check for unexpected variations or inconsistencies in crucial identifiers. For example, if a dataset is expected only to contain team identifiers 'A' and 'B', running this combined command immediately surfaces any other unique identifiers that might strongly suggest underlying data entry errors or corruption. This sophisticated combination of transformations represents one of the most fundamental and vital operations for robust data profiling within the PySpark ecosystem.

Deep Dive: Method 3 - Quantifying Unique Records

While the retrieval of the actual distinct rows is absolutely necessary for thorough data cleaning, the quantification of the extent of duplication is often a critical prerequisite step before making a final decision on whether comprehensive cleaning is warranted. The combined call of `df.distinct().count()` executes the complete deduplication logic internally and subsequently returns a single, precise numerical value representing the total number of unique records successfully identified in the DataFrame. Because `count()` is classified as an action, this entire operation immediately triggers and forces execution across the entire distributed cluster.

This method is exceptionally valuable for proactive quality assurance checks. By directly comparing the total raw number of rows (obtained via `df.count()`) with the total number of distinct rows (obtained via `df.distinct().count()`), we can accurately calculate both the exact number and the overall percentage of duplicate records present in the dataset. This calculated metric provides a highly crucial indicator of the dataset's overall health and effectively helps stakeholders prioritize their data cleaning and remediation efforts.

It is important to acknowledge and internalize the conceptual difference between `df.distinct().count()` and an aggregate function call like `df.agg(F.countDistinct('column'))`. While the former method calculates the count of unique combinations across the entirety of the row, the latter is restricted to counting the number of unique non-null values exclusively within a single, specified column. Understanding this critical distinction prevents common misinterpretations in data quality assessment. For assessing the overall data duplication across the entire record structure, using the full row `distinct().count()` remains the

ARABPSYCHOLOGY.COM

definitive and authoritative approach.

Practical Implementation: Setting Up the Sample DataFrame

To tangibly demonstrate the utility of these three methods, we must first establish a consistent and reliable testing environment. All fundamental PySpark operations necessitate the initialization of a SparkSession, which is responsible for managing the connection to and configuration of the Spark cluster. We will then proceed to define a small, controlled dataset that intentionally incorporates several duplicate rows to clearly and effectively illustrate how the distinct functions operate under expected conditions.

Our constructed sample data is designed to represent statistical performance metrics for players across two fictional teams, designated 'A' and 'B'. It is deliberately configured such that certain combinations of (team, position, points) are repeated in the raw input data, such as the row detailing 'A', 'Forward', 22 and the row for 'B', 'Guard', 14. This intentional redundancy ensures we can definitively verify the successful identification and removal of these duplicates when applying the appropriate functions.

The following code block meticulously sets up the execution environment, accurately defines the schema (column names), and creates the initial DataFrame. This structured setup forms the necessary foundation upon which all subsequent operational examples will be executed and verified:

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

# Define the raw data, containing intentional duplicate rows
data = ,
,
,
,
,
,
]

# Define column names
columns =

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

ARABPSYCHOLOGY.COM

```
# View the initial, raw 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|
+----+--------+------+
```

Step-by-Step Example 1: Eliminating Full Row Duplicates

Our initial, critical objective is the rigorous cleaning of the DataFrame by meticulously removing all rows that constitute exact duplicates across the entire set of defined columns: `team`, `position`, and `points`. Successfully executing this step guarantees that every resulting record represents a statistically unique observation within the dataset. We achieve this essential data cleansing by applying the foundational `distinct()` transformation directly to the base DataFrame.

As carefully observed in the raw data setup performed previously, rows 3 and 4 are identified as absolutely identical (containing the values A, Forward, 22), and rows 5 and 6 are also identical (B, Guard, 14). By applying the simple `df.distinct()` function, PySpark automatically identifies these pairs of redundant records and retains only a single, unique instance of each, thereby effectively reducing the overall record count and achieving a clean data set ready for further analysis.

The following syntax explicitly demonstrates the execution of this cleaning process and displays the resulting, refined DataFrame:

```
# Apply the distinct transformation to eliminate full row duplicates
df.distinct().show()

+----+--------+------+
|team|position|points|
+----+--------+------+
```

ARABPSYCHOLOGY.COM

A	Guard	11
A	Guard	8
A	Forward	22
B	Guard	14
B	Forward	13
B	Forward	7
+----+--------+------+

The resultant DataFrame is now comprised of **6 rows**, confirming conclusively that the two identified duplicate records were successfully isolated and removed from the dataset. Each row present in this output is definitively unique when considering the composite key formed by the combined values of all three columns. This result unequivocally validates the efficiency and operational simplicity of employing `df.distinct()` for comprehensive and high-volume data deduplication across PySpark DataFrames.

Step-by-Step Example 2: Finding Unique Column Entries

In this second example, we strategically shift our analytical focus away from unique rows to the task of identifying unique values exclusively within a single, designated column--specifically, the `team` column. This targeted operation provides us with the capability to rapidly catalogue the various teams represented throughout the dataset without being concerned with the potentially varying associated position or points data for each instance. This goal is elegantly accomplished by chaining the `select()` transformation before the final `distinct()` transformation.

Despite the fact that our original DataFrame contained eight total records, a quick inspection reveals that the `team` column only holds two fundamentally unique identifiers: 'A' and 'B'. The precise objective of this step is to confirm this cardinality and efficiently extract only these unique identifiers, presenting them within a new, much smaller DataFrame structure designed specifically for this purpose.

The code provided immediately below first executes the selection of the 'team' column and then proceeds to apply the required deduplication logic exclusively to the contents of that selected field:

 # Select the 'team' column, then display only the distinct values within it
df.select('team').distinct().show()

+----+
|team|
+----+
| A|

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

The final output is a highly compact DataFrame showcasing only the values 'A' and 'B'. This powerful combination of `select()` and `distinct()` is fundamentally critical for all aspects of categorical analysis and for the rigorous verification of data schemas. It serves as a clear demonstration of how efficiently PySpark can execute highly specific data profiling tasks, enabling developers and analytical teams to concentrate their immediate attention on single attributes rather than being burdened by the complexity of processing entire records.

Step-by-Step Example 3: Calculating the Total Distinct Count

As the concluding step, we utilize the potent combination of the `distinct()` transformation and the subsequent `count()` action to accurately quantify the total number of unique records present, critically doing so without incurring the overhead of retrieving the data itself. This methodology is particularly efficient and highly recommended for processing extremely large datasets where the physical display or materialization of the results would be computationally impractical or analytically unnecessary. The primary goal here is solely to determine the magnitude of duplication within the dataset.

Building directly upon the results established in Example 1, where we successfully identified that the initial 8-row DataFrame contained 2 sets of redundant records, we can confidently anticipate that the numerical result of this calculation will be 6. The `count()` function is an action that forces the immediate execution of the preceding `distinct()` transformation, providing the final result directly as a single, easily interpretable integer.

Executing this command provides immediate, precise diagnostic feedback regarding the cleanliness and structure of the data:

```
# Execute the distinct transformation followed by the count action
df.distinct().count()
```

```
6
```

The resultant output value of **6** definitively confirms that after the systematic removal of all redundant rows, exactly six distinct records remain within the PySpark DataFrames. This methodology delivers a rapid, highly resource-efficient means of measuring data cleanliness, establishing it as a fundamental and non-negotiable step in any serious data quality assurance pipeline constructed using PySpark. Mastery of these three core distinct handling techniques ensures that data engineers can proactively manage data quality and reliably prepare robust, clean

ARABPSYCHOLOGY.COM

datasets for all forms of advanced analytics.

ARABPSYCHOLOGY.COM