PySpark: Find Unique Values in a Column

Authored by

stats writer

November 17, 2025

Working with large datasets often requires analysts and data scientists to identify the unique categories or values present within a specific column. In the world of PySpark, which is the Python API for Apache Spark, the most straightforward and efficient method for retrieving a list of unique values from a column within a DataFrame is by utilizing the powerful **distinct function**. This function is fundamental when performing exploratory data analysis (EDA), ensuring data quality, or preparing data for subsequent aggregation steps, as it efficiently leverages Spark's capabilities for **distributed computing** across large clusters.

Understanding how the **distinct** operation works is crucial. When applied to a selected column, it performs a transformation that eliminates duplicate entries, returning a new DataFrame containing only the unique occurrences. This process is optimized for handling massive volumes of data that characterize modern data processing workflows. This comprehensive tutorial will delve into various practical examples demonstrating the proper syntax and application of this function, along with related methods for sorting and counting unique entries, providing a robust toolkit for data manipulation in Spark environments.

Setting Up the PySpark DataFrame

To illustrate the utility of the **distinct** function and related methods, we will first define a sample DataFrame. This foundational step ensures reproducibility and clarity throughout the examples. We begin by initializing a SparkSession, which serves as the entry point to programming Spark with the DataFrame API. Once the session is active, we define the raw data and column headers to construct our working dataset. This dataset models simple team performance data, including team name, conference affiliation, and points scored.

The structure of the data includes intentional duplicates across several columns, making it an ideal candidate for demonstrating uniqueness extraction. For instance, Team 'A' appears three times, and Team 'B' appears twice. Similarly, the 'conference' column contains duplicates ('East'), and the 'points' column shows a duplicate score of 6. This setup allows us to rigorously test how PySpark identifies and filters these repeated entries using the **distinct function**.

Below is the standard Python code necessary to set up this example DataFrame. We utilize the SparkSession to call `createDataFrame`, combining our list of tuples (data) with the specified column schema. Reviewing the `df.show()` output confirms that the DataFrame has been successfully created and accurately reflects the input data.

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|conference|points|
+----+----------+------+
A	East	11
A	East	8
A	East	10
B	West	6
B	West	6
C	East	5
+----+----------+------+

Example 1: Basic Retrieval of Unique Column Values

The most common application of the **distinct function** is to ascertain all unique entries within a single column. To achieve this, we first use the **select** transformation to narrow the DataFrame down to the specific column of interest. If **distinct** were applied directly to the full DataFrame, it would identify unique rows, which is a different operation than identifying unique values within a column. By coupling **select** with **distinct**, we explicitly instruct Spark to focus the uniqueness check on the values present in the chosen column.

In this initial example, we focus on the `team` column. This column clearly contains duplicate entries (A, B, A, B, C, A), and our objective is to return the resulting set of unique team identifiers, which we expect to be A, B, and C. The syntax is concise and highly readable, reflecting the declarative nature of the Spark API. The final `show()` action triggers the execution of the transformations on the cluster and prints the resulting small DataFrame containing only the distinct entries.

The code snippet below demonstrates the required operations. We chain `select`, `distinct`, and `show` together. This technique is highly efficient because Spark optimizes these chained operations before execution. Observing the output confirms that all duplicate team entries have been successfully collapsed, leaving only one instance of each unique team identifier, thereby fulfilling the objective of finding the distinct values in the **team** column.

df.select('team').distinct().show()

```
+----+
|team|
+----+
| A|
| B|
| C|
+----+
```

The resulting mini-DataFrame clearly shows that the unique values residing in the **team** column are **A**, **B**, and **C**. This approach is fundamental for cleaning data, identifying categories, or preparing lists for validation checks within large-scale data processing pipelines. It is a prerequisite for subsequent analytical steps, such as computing group aggregates or generating comprehensive reports on categorical features.

Example 2: Finding and Sorting Unique Values in a Column

While the basic application of the **distinct function** successfully extracts unique values, the resulting order of these values is generally arbitrary and dependent on how Spark processes the data internally across its partitions. For presentation purposes or if the unique values need to be processed sequentially, it is often necessary to impose a specific sort order. If we examine the unique values for the `points` column, we see the raw output reflects an unsorted list of integers: 11, 8, 10, 6, and 5.

To introduce order into the result set, we integrate the **orderBy** function into our transformation chain. This function allows us to specify the column by which the result should be sorted. When applied after **distinct**, **orderBy** ensures that the final output is both unique and organized according to standard numerical or alphabetical rules. We first retrieve the distinct points, store the result temporarily as `df_points`, and then apply the sorting logic.

To achieve an ascending sort, which is the default behavior in PySpark, we simply apply the **orderBy** method to the derived DataFrame containing the unique points. This transformation is highly valuable when dealing with quantitative data where ranges and sequencing are important

contextually. The resulting output presents the unique point values clearly structured from the lowest value (5) to the highest value (11).

#find unique values in points column
df_points = df.select('points').distinct()

#display unique values in ascending order
df_points.orderBy('points').show()

```
+------+
|points|
+------+
| 5|
| 6|
| 8|
| 10|
| 11|
+------+
```

Furthermore, the **orderBy** function provides flexibility to reverse the sorting direction. By passing the optional argument `ascending=False`, we can easily return the unique values in **descending order**. This capability is frequently used when analysts need to quickly identify the top or highest unique values within a metric column, such as finding the highest unique scores achieved by the teams in our sample data. This refinement significantly enhances the utility of the distinct operation for analytical purposes.

#find unique values in points column
df_points = df.select('points').distinct()

#display unique values in descending order
df_points.orderBy('points', ascending=False).show()

```
+------+
|points|
+------+
| 11|
| 10|
| 8|
| 6|
| 5|
+------+
```

Example 3: Finding and Counting Unique Values in a Column

Often, simply knowing what the unique values are is insufficient; analysts also need to know the frequency or count of each unique occurrence. This analysis is performed to understand the distribution of categorical variables, identify imbalances in the dataset, or prepare for frequency-based aggregations. While the **distinct function** only provides the list of unique items, we can easily calculate their frequencies using the combination of the **groupBy** and **count** functions.

The **groupBy** operation partitions the DataFrame data based on the unique values found in the specified column (in this case, `team`). Once grouped, the **count** aggregation function is applied to each group, effectively calculating the number of records associated with that specific unique value. This technique is statistically equivalent to generating a frequency table for the categorical variable. This process is highly optimized in PySpark, as the grouping and counting happen in parallel across the cluster nodes, making it efficient even for petabyte-scale datasets.

The following code demonstrates this powerful technique applied to the `team` column. Instead of using `select().distinct()`, we immediately call `groupBy('team')`, followed by `count()`. The resulting DataFrame will contain two columns: the grouping column (`team`) and the aggregate count (labeled `count` by default). This method provides a direct and comprehensive view of the composition of the categorical variable.

df.groupBy('team').count().show()

```
+----+-----+
|team|count|
+----+-----+
| A| 3|
| B| 2|
| C| 1|
+----+-----+
```

Analyzing the output, we successfully identify the three unique team values (**A**, **B**, and **C**) and simultaneously obtain the total number of records associated with each team. Team A appears three times, Team B appears twice, and Team C appears once. This combined approach of finding and counting is indispensable for data validation, identifying outliers, and generating key summary statistics required for further statistical modeling or business intelligence reporting.

Advanced Method: Collecting Unique Values to Python

While performing operations within the distributed Spark environment is generally preferred for

performance, there are instances where the final list of unique values must be transferred back into the local Python execution environment, typically as a standard Python list, for use in subsequent non-Spark operations, such as configuration settings, generating internal lookup tables, or feeding into standard machine learning libraries that expect local data structures. This transfer involves a `collect()` action, which should be used cautiously on very large datasets.

To achieve this, we first execute the `select().distinct()` transformations as before, ensuring we have a Spark DataFrame containing only the unique values. We then use the `collect()` action. Since `collect()` returns a list of PySpark `Row` objects, we typically follow up by mapping over this list to extract the actual value contained within the single column of the Row object. This ensures the result is a clean list of primitive Python types (strings, integers, etc.).

For example, to retrieve the unique team names into a Python list named `unique_teams_list`, the following sequence of operations is required:

 # 1. Select distinct values and collect the Row objects
unique_rows = df.select('team').distinct().collect()

# 2. Extract the value from the Row object (index 0)
unique_teams_list = for row in unique_rows]

# 3. Print the result (shows)
print(unique_teams_list)

It is paramount to remember that calling `collect()` forces all the data, even if it is just a list of unique values, to be serialized and moved to the driver node's memory. If the column contains millions of unique values, this can lead to driver memory exhaustion (an "out of memory" error). Therefore, this method is only recommended when the expected number of unique values is guaranteed to be small enough to safely fit within the driver's resources, thus maintaining the stability of the **PySpark** application.

Performance Considerations for Distinct Operations

The **distinct function**, while conceptually simple, is one of the most resource-intensive operations in distributed systems like Spark. This is because determining uniqueness requires a global shuffle operation. A shuffle is the process of redistributing data across the cluster, involving disk I/O, network transfer, and serialization, ensuring that all identical keys (values in the column) end up on the same worker node so they can be accurately compared and counted only once.

When performing `df.select(column).distinct()`, Spark must perform a significant amount of data movement. If the DataFrame is extremely large and contains a high cardinality column (many

unique values), the shuffle operation can become a major bottleneck, impacting latency and resource consumption. Data engineers often analyze the cardinality of columns before running **distinct** at scale, using sampling or descriptive statistics to estimate the performance impact.

In contrast, using `groupBy().count()` often involves a similar, yet slightly different, shuffle and aggregation pattern. In many modern Spark versions, the cost difference between `distinct()` and `groupBy().count()` might be negligible for small-to-medium datasets, but it is important to be aware of the underlying performance implications. For massive datasets, alternative, more specialized techniques like HyperLogLog approximation (often used for estimating distinct counts) might be considered if the exact count is not strictly required, as approximations avoid the costly full shuffle.

Using Distinct on Multiple Columns (Unique Rows)

While the focus of this guide has been finding unique values within a single column, it is important to note the behavior of the **distinct function** when applied across multiple columns or the entire DataFrame. When applied to a DataFrame containing more than one column, the function identifies and returns only the rows that are entirely unique based on the combination of values across all selected columns. This operation is essential for removing exact duplicate records from a dataset.

To demonstrate, if we apply **distinct** to our sample data on the combination of `team` and `conference`, the resulting unique set of rows would filter out the duplicate row where Team 'B' played in the 'West' conference and scored 6 points, as this row appeared twice in the input. If we were to apply it to the entire DataFrame, the row would still be considered unique relative to , but one of the two identical entries would be removed.

The syntax for checking uniqueness across selected columns involves using `select()` with multiple column names before calling `distinct()`. This pattern helps ensure data integrity by cleaning up redundant records that might have been introduced during data ingestion or merging processes. Although similar to the single-column unique retrieval, the interpretation shifts from finding unique categories to finding unique transactional records, which is a key distinction in **DataFrame** manipulation.

Summary of Unique Value Retrieval Methods

Identifying unique values is a core requirement in data analysis across all platforms, and PySpark provides highly optimized mechanisms to handle this task efficiently in a **distributed computing** environment. We have covered three primary methods, each tailored for a specific analytic need:

Simple Uniqueness (Extraction): Utilizing `df.select(column).distinct()` to quickly retrieve

the raw list of unique entries within a single column. This is the fastest method if only the unique values are needed, regardless of their order.

Uniqueness with Ordering: Combining `distinct()` with `orderBy()` to ensure the resulting unique list is presented in a readable, sorted sequence (ascending or descending). This enhances readability and facilitates structured analysis.

Uniqueness with Frequency (Counting): Employing `df.groupBy(column).count()` to simultaneously identify unique values and measure their respective frequencies across the entire dataset. This method yields immediate statistical insights into the distribution of the column's categories.

Choosing the correct method depends entirely on the analytical goal. For simple identification, the basic **distinct function** is sufficient. When structural presentation is required, **orderBy** adds necessary clarity. Finally, for distribution analysis, the **groupBy** technique is unmatched. By mastering these techniques, developers and analysts can effectively manage data quality and perform deep exploratory analysis on massive datasets using the robust capabilities of **PySpark** and Apache Spark's DataFrame API.