Learning PySpark: A Practical Guide to Finding Unique Values in DataFrame Columns

Authored by

Mohammed looti

November 10, 2025

Working with large-scale datasets often requires identifying the cardinality of specific fields--that is, determining the set of unique elements within a column. In the world of big data processing, this task is efficiently handled by frameworks like PySpark. The most straightforward method for obtaining a list of unique values in a PySpark DataFrame column involves leveraging the powerful built-in distinct function. This function is a crucial part of the Spark SQL API, designed for distributed computation, ensuring that even massive datasets can be processed quickly and accurately.

This comprehensive tutorial delves into various practical applications of the distinct function, demonstrating how it can be combined with other DataFrame transformations to achieve specific analytical goals, such as sorting results or performing data aggregation. We will explore several examples, starting with the necessary setup of our sample data environment.

Setting Up the PySpark Environment and Sample Data

Before executing any transformation, we must initialize a SparkSession, which serves as the entry point to programming Spark with the Dataset and DataFrame API. This session manages the connection to the underlying cluster infrastructure (whether local or distributed) and allows us to start processing data. For demonstration purposes, we will construct a small, representative DataFrame containing information about teams, their conferences, and accrued points. This simple structure allows us to clearly observe the effects of the unique value extraction methods.

The code block below illustrates the standard procedure for importing necessary libraries, creating the Spark entry point, defining the raw data structure, specifying column names, and finally, materializing the PySpark DataFrame. Displaying the resulting DataFrame ensures that the initial data state is correctly understood before proceeding with transformations.

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

This DataFrame, labeled **df**, now serves as our foundation. Notice that the data contains duplicate entries. For instance, 'A' appears three times, and the combination 'B', 'West', 6 appears twice. Our goal is to extract only the unique identifiers from specific columns, demonstrating the efficiency of distributed data processing inherent in PySpark operations.

Core Technique: Finding Unique Values Using the Distinct Transformation

The core mechanism for retrieving unique values is the application of the distinct function. It is important to understand that in Spark, **distinct()** is a transformation that operates on the entire DataFrame or a selected subset of columns. When applied to a single column, it effectively eliminates all redundant entries, leaving behind only one instance of each unique value present in that column.

To isolate the unique values within a specific column, such as the **team** column, we first use the **select()** method. The **select()** method projects the DataFrame onto the specified column(s). Once the DataFrame is reduced to contain only the column of interest, we chain the **distinct()** transformation. This sequence ensures that Spark only processes the necessary data subset for uniqueness checking, optimizing resource utilization.

Consider the following syntax applied to the **team** column. This simple, two-step operation--selection followed by distinction--is highly idiomatic in PySpark and yields a new DataFrame containing only the unique teams.

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

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

As demonstrated by the output, the resulting DataFrame clearly shows the unique identifiers in the **team** column: **A**, **B**, and **C**. This process is fundamental for tasks such as calculating the number of distinct users, categories, or geographical regions present in a large dataset. While this example is straightforward, in production environments, the Spark cluster handles the complex shuffling and comparison required to ensure global uniqueness across all partitions of the distributed data.

Refining Results: Sorting Unique Values

While the distinct function successfully identifies unique entries, the resulting order is often arbitrary, depending on how Spark processed the data across its executors. For analytical purposes or presentation, it is frequently necessary to present these unique values in a specific sequence, typically numerical or alphabetical order. If we examine the unique points awarded, we can see this arbitrary ordering immediately.

Applying the **select()** and **distinct()** operations to the **points** column reveals the unique scores, but they are not arranged sequentially:

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

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

The output above displays all unique point totals (11, 8, 10, 6, 5), but they are unsorted. To impose a meaningful order, we must introduce the **orderBy()** transformation. This transformation is applied directly after the **distinct()** operation, ensuring that the unique values are properly collated and

sorted before being displayed or used in subsequent analysis. By default, **orderBy()** sorts the specified column in **ascending order**.

The following sequence of operations first finds the unique points and then explicitly sorts them from the smallest value to the largest. This two-stage approach allows for clear separation between the data cleansing (uniqueness) and data presentation (ordering) steps.

#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, flexibility in ordering is often required. If the analytical requirement demands viewing the highest scores first, we can easily reverse the sorting direction. This is accomplished by passing the optional argument **ascending=False** to the **orderBy()** function. This small modification transforms the output to present the unique values in **descending order**, which is highly useful when focusing on top performers or maximum values within the distribution of unique data points.

#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|
+------+

Advanced Usage: Counting the Frequency of Unique Values

While knowing which values are unique is essential, understanding how frequently each unique value appears provides critical context for data aggregation and distribution analysis. This is where PySpark's aggregation functions come into play. Instead of simply listing the unique entries, we often need to perform a frequency count, effectively creating a summary table where each unique element is paired with its total occurrence count.

To achieve this, we utilize the **groupBy()** transformation, followed by the **count()** aggregation function. The groupBy transformation partitions the DataFrame data based on the unique values in the specified column (in this case, **team**). Once grouped, the subsequent **count()** operation calculates the number of rows belonging to each distinct group, thereby giving us the frequency of each unique team.

The code below demonstrates the efficient use of this aggregation pattern in PySpark:

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

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

The resulting output is highly informative. We can see the three unique values (**A**, **B**, **C**) and the exact number of times each unique value occurs in the original dataset. Team A appears three times, Team B appears twice, and Team C appears once. This technique is invaluable for generating histograms, determining data imbalance, or preparing data for machine learning models where feature distribution matters significantly. It is generally more computationally efficient than first applying **distinct()** and then manually counting, as groupBy and **count()** are optimized to work together in a distributed environment.

Performance Considerations and Best Practices

While the distinct function is syntactically simple, it is crucial to understand its performance implications, especially when dealing with truly massive PySpark DataFrames. Both **distinct()** and the combination of groupBy and **count()** are known as wide transformations in Spark.

A wide transformation requires data shuffling--moving data across the network between different worker nodes (executors) in the Spark cluster. This shuffling is necessary because Spark must ensure that all identical values, regardless of which machine they initially reside on, are brought together onto the same machine for comparison or data aggregation. Shuffling is typically the most expensive operation in Spark due to network latency and disk I/O.

Minimizing Columns: Always use **select()** before **distinct()** when you only need unique values from a subset of columns. Selecting only the necessary columns reduces the amount of data that needs to be shuffled across the network, leading to substantial performance gains.
Choosing the Right Tool: If your primary goal is simply to count the unique values (cardinality), functions like **approx_count_distinct()** can be considered. This function provides an approximate count much faster than the exact count derived from **distinct().count()** or **groupBy().count()**, trading minor accuracy for significant speed improvement, especially suitable for early exploration of large datasets.
Understanding GroupBy vs. Distinct: If you need the actual list of unique values, use **distinct()**. If you need both the list and the frequency of occurrence, use groupBy. While both involve shuffling, groupBy is explicitly designed for the aggregation task and is often the preferred method when frequency analysis is paramount.

Properly managing these transformations ensures that your PySpark jobs execute efficiently, preventing bottlenecks that can arise from unnecessary data movement, particularly in cloud environments where network costs and execution time are critical metrics.

Conclusion and Further Steps

The ability to accurately and efficiently identify unique values is a cornerstone of data quality checks and preliminary statistical analysis in big data environments. PySpark's distinct function, in conjunction with other DataFrame methods like **select()**, **orderBy()**, and **groupBy()**, provides a robust toolkit for handling this essential task across distributed data. We have seen how to isolate unique entries, how to impose order on the results for better readability, and how to combine transformations to calculate the frequency distribution of these unique elements.

Mastering these fundamental transformations within the PySpark DataFrame API is essential for any data engineer or data scientist working with Apache Spark. The declarative nature of the API allows users to focus on the "what" (finding unique values) rather than the "how" (managing

distributed data movement), leaving the optimization to the Spark execution engine.

To further enhance your skills in PySpark and expand upon the concepts of data manipulation and aggregation, consider exploring the following advanced topics and related tutorials:

Exploring Window Functions for complex row-level calculations.
Implementing Joins and Unions to combine multiple DataFrames effectively.
Handling Nulls and Missing Data using techniques like **dropna()** or imputation.
Optimizing Spark performance through caching and partitioning strategies.

Additional Resources

The following tutorials explain how to perform other common tasks in PySpark, building on the foundation established by these uniqueness and aggregation examples: