How to Select the First Row of Each Group in PySpark

Authored by

stats writer

February 6, 2026

When working with large datasets in **PySpark**, a very common requirement is to isolate or select the first entry corresponding to a unique group defined by one or more columns. This operation is fundamental for tasks such as deduplication, identifying the initial record, or performing analytical ranking. While simple aggregation functions like `groupBy().first()` might seem intuitive, they are often insufficient or inefficient in a distributed computing environment like **PySpark**, especially when dealing with complex datasets where row order matters.

To reliably select the first row within each group, we must employ **Window functions**. These specialized functions allow us to perform calculations across a set of table rows that are related to the current row, effectively partitioning the data and assigning ranks based on specific criteria. The method outlined below utilizes the **row_number** function combined with a defined window specification to accurately identify and filter the designated 'first' record for every unique partition.

PySpark: Select First Row of Each Group

Introduction: Why Group Selection is Essential

In data processing pipelines, ensuring data uniqueness and achieving specific analytical views often requires partitioning the data. Selecting the first row of a group means defining an arbitrary or specific ordering criteria and then extracting the row that meets the highest rank (usually rank 1) within its designated partition. This approach is highly flexible and essential when working with large-scale **DataFrame** structures.

The primary reason for using **Window functions** over standard grouping and aggregation methods is the need to retain all column information from the original row. Standard aggregations typically collapse multiple rows into a single summary row, losing the original detail. Window functions, conversely, calculate a value (like a rank) and append it to the existing **DataFrame** rows, allowing for subsequent filtering while preserving data integrity.

The general methodology involves four distinct steps: defining the partition, defining the ordering, assigning a sequential row number within that partition, and finally, filtering the **DataFrame** based on the assigned row number of 1. This ensures that only the desired record is kept for each unique key.

The Core Concept: Utilizing PySpark Window Functions

To successfully select the first row of each group in **PySpark**, you must utilize the `Window` class from `pyspark.sql.window` and the `row_number` function from `pyspark.sql.functions`. The `Window` object defines the scope of the calculation. This object requires two critical specifications: partitioning and ordering.

The **partitionBy** method divides the **DataFrame** into logical groups based on the unique values in specified columns. All rows sharing the same values in these columns form a single partition, and the ranking will reset within each new partition. This step is equivalent to the `GROUP BY` clause in standard SQL queries.

The `orderBy` method is equally crucial as it dictates which row will be considered 'first'. If a specific criterion (e.g., maximum score, latest timestamp) determines the first row, this criterion must be specified within `orderBy`. If the selection is arbitrary (meaning any row from the group will suffice), an arbitrary value must still be provided to satisfy the structural requirements of the **Window functions**, as demonstrated in the practical syntax.

Implementing the Solution: Syntax Deep Dive

The following syntax demonstrates the standard and most efficient way to achieve first-row selection per group using the established **Window functions** methodology in **PySpark**. This structure is highly scalable and optimized for distributed execution across a Spark cluster.

You can use the following syntax to select the first row by group in a **PySpark DataFrame**:

```
from pyspark.sql.functions import row_number,lit
from pyspark.sql.window import Window

#group DataFrame by team column
w = Window.partitionBy('team').orderBy(lit('A'))

#filter DataFrame to only show first row for each team
df.withColumn('row',row_number().over(w)).filter(col('row') ==1).drop('row').show()
```

This particular example selects the first row for each unique **team** value in the **DataFrame**. The critical element here is the use of `over(w)`, which applies the **row_number** calculation across the window defined by `w`. The final filtering step, `filter(col('row') == 1)`, isolates only the entries that were ranked first within their respective team partitions.

Setting Up the Environment and Example Data

To illustrate this technique practically, we will establish a simple **PySpark DataFrame** containing information about basketball players. This dataset includes columns for the player's team, their position, and their points scored, providing multiple rows for each team, which necessitates the grouping operation.

The following code block initializes the Spark session, defines the input data and schema, and

creates the resulting **DataFrame** that we will subsequently operate on. This setup is mandatory for any PySpark operation.

The following example shows how to use this syntax in practice. Suppose we have the following **PySpark** DataFrame that contains information about basketball players on various teams:

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	14

| C| Forward| 23|
| C| Guard| 30|
+----+--------+------+

As observed in the output, teams 'A' and 'B' each have four entries, and team 'C' has two entries. Our objective is to reduce this **DataFrame** so that only one unique row remains for each team.

Executing the Row Selection Logic

Our goal is to select the first row for each unique team. Since we haven't specified a preference (like the highest scoring player or the alphabetically first position), the 'first' row will be determined arbitrarily based on the internal ordering Spark maintains when the data is partitioned. This arbitrary selection is achieved by using the combination of **partitionBy** and a trivial `orderBy` clause.

The code first defines the window specification `w`, partitioning the data by the 'team' column. Then, it uses `withColumn` to generate a temporary column named 'row'. This new column contains the result of the **row_number** function, assigning ranks starting from 1 within each team. Finally, we filter for rank 1 and remove the temporary 'row' column for a clean final output.

We can use the following syntax to do so:

```
from pyspark.sql.functions import row_number,lit
from pyspark.sql.window import Window

#group DataFrame by team column
w = Window.partitionBy('team').orderBy(lit('A'))

#filter DataFrame to only show first row for each team
df.withColumn('row',row_number().over(w)).filter(col('row') ==1).drop('row').show()
```

```
+----+--------+------+
|team|position|points|
+----+--------+------+
| A| Guard| 11|
| B| Guard| 14|
| C| Forward| 23|
+----+--------+------+
```

Interpreting the Results

The resulting **DataFrame** successfully shows only one entry for each unique team identifier ('A', 'B', and 'C'). For team 'A', the row {'A', 'Guard', 11} was selected as the first. For team 'B', {'B', 'Guard', 14} was selected, and for team 'C', {'C', 'Forward', 23} was selected.

It is important to remember that since we used an arbitrary ordering (via `lit('A')`), the specific row chosen for teams 'A' and 'B' might change between different runs if the underlying data is repartitioned or shuffled. If guaranteed determinism is required, a proper ordering column must be used in the `orderBy` clause, such as `orderBy('points', ascending=False)` to select the highest-scoring player on the team.

The resulting **DataFrame** shows only the first row for each unique team value, based on the arbitrary ordering applied within the window specification. This technique is robust and scalable across massive datasets.

Advanced Use Cases: Custom Ordering and Multi-Column Partitioning

The power of the **Window functions** approach lies in its flexibility. Data requirements frequently dictate selecting the "best" or "most recent" row, rather than just an arbitrary first row. For instance, if we wanted to select the player with the highest 'points' for each team, we would modify the window definition to prioritize that column.

To implement this, we would replace `orderBy(lit('A'))` with `orderBy(col('points').desc())`. The `desc()` function ensures that the highest point value receives the lowest rank number (i.e., rank 1), thereby fulfilling the requirement. If there are ties in points, Spark's internal ordering will break the tie arbitrarily unless a secondary sorting criterion is specified (e.g., `orderBy(col('points').desc(), col('position').asc())`).

Furthermore, grouping can be extended beyond a single column. If you wish to partition the data based on a combination of characteristics--for example, selecting the first row for every unique combination of 'team' and 'position'--you simply include multiple columns in the **partitionBy** function. This changes the scope of the window calculation significantly, generating a much finer granularity of groups.

Note #1: If you would like to group by multiple columns, simply include multiple column names in the **partitionBy** function, separated by commas (e.g., `partitionBy('team', 'position')`). This allows for complex hierarchical grouping necessary for detailed analysis.

Understanding the Role of `lit('A')` in Arbitrary Ordering

A common point of confusion when implementing arbitrary first-row selection is the use of `lit('A')` within the `orderBy` clause: `orderBy(lit('A'))`. The `orderBy` clause is technically mandatory when using the **row_number** function, as this function requires a strict ordering to assign sequential integers. Without explicit ordering, the rank assignment would be non-deterministic.

By using `lit('A')`, we are instructing **PySpark** to create a temporary column where every single row within the current partition has the identical, constant value 'A'. When Spark attempts to sort by this constant value, all rows are considered equal in terms of ordering, resulting in an arbitrary but stable assignment of the row numbers based on Spark's internal physical data layout at that moment. This satisfies the syntactic requirement of `orderBy` while avoiding the imposition of a logical order on the data.

Note #2: The syntax **lit('A')** is simply used as an arbitrary constant value. You can replace 'A' with anything you'd like, such as a number or another string literal, and the code will still work identically, ensuring that the ordering requirement for **Window functions** is met without introducing unintended sorting logic.

Summary of Best Practices for PySpark Grouping

Selecting the first row of each group using **Window functions** is the recommended, high-performance method in PySpark for data filtering and deduplication tasks. It offers superior control and scalability compared to less explicit methods.

Determinism is Key: Always use a meaningful column in `orderBy` (e.g., a timestamp or ID) if the definition of "first" is crucial to your business logic.
Efficiency: The use of `row_number()` is generally more efficient for exact deduplication than functions like `rank()` or `dense_rank()`, which are designed to handle ties differently.
Cleanup: Remember to use the `.drop('row')` command to remove the temporary ranking column, maintaining a clean schema for subsequent operations.

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