How to Select the First Row of Each Group in a PySpark DataFrame

Authored by

stats writer

January 20, 2026

Leveraging Window Functions for Grouped Selection in PySpark

One of the most common requirements when performing data analysis in distributed environments like PySpark is the need to select a specific record--often the first, last, or highest-ranked--from each logical group within a large dataset. While standard SQL uses constructs like `GROUP BY`, selecting a non-aggregated row based on a grouping key requires the powerful concept of **Window Functions**. This method allows us to calculate a rank or serial number within partitioned groups, providing precise control over which records are selected while maintaining the distributed nature of Spark processing.

The standard syntax to efficiently select the first row by group in a DataFrame utilizes the `row_number()` function combined with a **Window Specification**. This technique ensures that Spark processes the data in a highly optimized manner across its clusters. The core approach involves partitioning the data by the desired grouping column, assigning a sequential index using ranking, and then filtering for the records where this index equals one. This is the most efficient pattern for achieving arbitrary row selection per group in large-scale data environments.

The Core PySpark Syntax for Arbitrary First Row Selection

To achieve this selection, you must import necessary functions from `pyspark.sql.functions` (specifically `row_number` and `lit`) and `pyspark.sql.window` (for the `Window` object). The following code snippet illustrates the fundamental pattern used to identify and extract the first entry for every group defined by the `team` column:

```
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 powerful example specifically selects the first row encountered for each unique `team` value in the DataFrame. Note the critical use of `lit('A')` in the `orderBy` clause. Since we are interested in an arbitrary first row rather than a value-determined first row (like highest score), we use a literal value to satisfy the syntax requirement of `row_number()`, which mandates an ordering clause. This prevents unnecessary and costly sorting operations.

Understanding the Window Specification Components

To utilize this technique effectively, it is vital to dissect the **Window Specification** object (`w`). This object dictates how the ranking function operates across the distributed data. The specification relies primarily on the `partitionBy()` clause. The `partitionBy()` function defines the boundaries of the groups; conceptually, Spark treats all rows sharing the same value in the partition column(s) as belonging to one independent group, and the ranking calculation resets every time a new partition begins.

The function `row_number()` is perhaps the most critical component for this task. It assigns a sequential integer starting from 1 to each row within its defined partition. Since we are only interested in the record that receives the initial index, we subsequently filter the DataFrame to keep only those rows where this calculated row number is equal to one (1). This elegant combination of partitioning and ranking provides a highly scalable and optimized solution for non-aggregated group selection in PySpark.

Practical Example: Setting up the Sample Dataset

Let us apply this methodology to a realistic scenario involving player statistics. Suppose we have a PySpark DataFrame containing detailed information about basketball players, including their team, position, and recent points scored. Our specific goal is to retrieve one representative row for each unique team present in the dataset, effectively de-duplicating the teams based on their order of appearance.

We begin by setting up the Spark session and defining our sample data structure. This data includes multiple records for each team (A, B, and C) to demonstrate how the partitioning logic successfully isolates only one record per group. The initial setup ensures that the DataFrame is correctly initialized with the schema required for the subsequent **Window Function** transformation.

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

The input DataFrame contains ten rows in total, distributed across three unique teams (A, B, and C). The ensuing process must efficiently reduce this structure down to three rows, retaining all columns, by selecting only the first entry encountered for each team partition.

Applying the Window Logic and Filtering Results

With the necessary imports secured (including `row_number` and `lit`) and the input DataFrame prepared, we now apply the four critical, chained operations:

Define the Window (`w`): We use `Window.partitionBy('team')` to instruct Spark to treat all rows belonging to the same team as a single partition.

Apply the Rank: The `withColumn` operation calculates the rank (using `row_number().over(w)`) and stores it in a temporary column named `row`.

Filter the Result: We retain only those rows where the calculated rank is 1, thus guaranteeing only the "first" row of each partition remains.

Clean Up: The temporary column `row` is removed using `drop()` to ensure the final output structure is clean and relevant.

The complete execution sequence is demonstrated below, yielding the desired filtered result:

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

The resulting DataFrame successfully isolates exactly one entry for each unique **team** value (A, B, and C). The specific rows selected (e.g., team A, Guard, 11 points) are arbitrary, as intended by the use of the literal ordering.

Deep Dive: The Necessity of `orderBy(lit('A'))`

The syntax `orderBy(lit('A'))` is a deliberate technique to achieve non-deterministic selection, which is often crucial for simple deduplication where performance is paramount. Because the `row_number()` function, following standard SQL specifications, mathematically requires an ordering within the window partition to assign a rank, we must provide it.

If we were to omit `orderBy()` entirely, the code would fail. However, if we were to use a real column like `orderBy('points')`, Spark would perform a physical sort on the data within each partition before assigning ranks, guaranteeing that the row with the lowest points receives rank 1. By using `lit('A')`, we introduce a constant value for sorting. Since all rows in the partition have the same value, no actual meaningful sort is performed, and the ranking relies on the existing

physical order of the data on the executor node. You can replace 'A' with any constant value (like `lit(1)` or `lit('X')`) and the resulting arbitrary selection will be the same.

Advanced Grouping: Partitioning by Multiple Columns

The method described is highly flexible and easily extends to scenarios where grouping must occur across a combination of keys. If, for instance, you needed the first row for every unique combination of **team** and **position** (a composite key), the only modification required is within the `partitionBy()` clause definition.

To group by multiple columns, you simply include all necessary column names as arguments to the `partitionBy` function. This instructs Spark to create a distinct partition boundary whenever the value in the combination of specified columns changes. For example, if we applied this logic to the sample data, we would likely get more than three resulting rows (e.g., one row for 'A'/'Guard' and one for 'A'/'Forward').

 # Window definition for grouping by 'team' AND 'position'
w_multi = Window.partitionBy('team', 'position').orderBy(lit('X'))

# Apply and filter
df.withColumn('row',row_number().over(w_multi)).filter(col('row') == 1).drop('row').show()

This functionality proves invaluable for deduplication tasks where uniqueness must be enforced across composite keys in large-scale datasets managed by PySpark, ensuring accurate data representation based on distinct combinations.

Summary of Key Takeaways

Mastering the selection of the first row by group is a foundational skill in PySpark data manipulation, allowing for efficient deduplication and ranked subsetting. The efficacy and performance of this technique rely entirely on the proper definition of the **Window Specification**.

The method utilizes `Window.partitionBy()` to logically separate the data into independent groups.

The `row_number()` function assigns sequential ranks within these defined partitions.

Using `orderBy(lit('constant'))` is the most optimized pattern when seeking an arbitrary first row, as it satisfies the function's requirements without incurring the cost of a deterministic physical sort.

The final `filter()` step isolates the desired ranked row (where rank equals 1).

This approach provides a robust, scalable solution for addressing complex group-wise selection problems in a distributed computing environment.