Learning PySpark: Selecting the First Row in Each Group of a DataFrame

Authored by

Mohammed looti

November 11, 2025

The Challenge of Group-Wise Selection in PySpark

A fundamental requirement in large-scale data analysis and transformation using PySpark is the ability to distill a large dataset down to a single, representative record for each defined group. This is often necessary when dealing with temporal data, transaction histories, or log files where multiple entries exist for the same entity, and we only need the most relevant one--such as the first, last, or highest-scoring record.

Unlike simple aggregation functions (like `SUM` or `AVG`) which mathematically collapse grouped data into a single summary row, selecting the "first" non-aggregated row requires a specialized technique. We must retain the full schema and column structure of the original data while ensuring that only one row passes the grouping constraint. Standard SQL `GROUP BY` clauses are insufficient for this task because they mandate aggregation across all non-grouped columns, which is counterproductive when attempting to preserve the original record structure.

To solve this efficiently in a distributed environment, we need a mechanism to logically segment the overall dataset based on a specific key (the grouping column) and then assign a sequential identifier or rank to every row within those segments. This ranking step is crucial as it allows us to programmatically identify and filter the desired entry--the one assigned the rank of 1--which represents the first record based on either arbitrary arrival order or a strictly defined sorting criterion. The solution must harness Spark's distributed execution engine to maintain performance when dealing with massive DataFrames.

Introducing PySpark Window Functions for Precise Ranking

The most robust and scalable method to select a specific row within a group in PySpark involves leveraging the advanced concept of Window functions. Window functions operate on a set of rows related to the current row, known as a "window," without collapsing them. This capability makes them ideal for calculating running totals, moving averages, or, in our case, assigning sequential ranks for subsequent filtering.

A Window function differs fundamentally from standard aggregation because it returns a calculated value for every input row, rather than just one value per group. This preserves the original row structure while adding necessary metadata--the rank--which determines the row's position relative to its peers within the defined partition. This process allows for group-wise calculations and selection operations that maintain the full detail of the original DataFrame schema.

The mechanism relies on three core concepts working in concert: defining the grouping boundaries, specifying the sorting order to determine "first," and applying a ranking function. By carefully structuring the window, we ensure that the ranking operation is correctly distributed across the cluster, guaranteeing an efficient and accurate result, even when processing petabytes

of data. This approach is highly declarative and is generally the preferred method for complex data sampling and deduplication workflows within the Spark ecosystem.

The Essential Components: Partitioning and Ranking

To reliably define and select the first row for each group, we must construct a specific Window specification. This specification requires two essential PySpark components: the grouping definition and the ranking logic. The grouping boundaries are established using the Window.partitionBy method, which logically segments the DataFrame based on one or more key columns, ensuring that the subsequent calculations only occur among rows sharing the same key value.

Once the partitions are defined, the sequential rank is assigned using the row_number function. This function assigns a unique, sequential integer (starting at 1) to each row within its partition. The row receiving the rank of 1 is, by definition, the first row encountered (or the highest/lowest value, depending on the sort order) within that group.

The overall workflow involves defining the Window specification, applying `row_number()` over this specification to create a new temporary column (often named 'row' or 'rank'), and finally, filtering the DataFrame where this temporary column equals 1. After the filtering is complete, the temporary ranking column is dropped to return a clean output DataFrame containing only the desired first records. This technique is highly flexible and scalable for managing large-scale data filtering tasks.

The following fundamental snippet demonstrates this core logic, selecting the first row encountered for each unique **team** value in the DataFrame after partitioning:

```
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()
```

Practical Implementation: Setting Up the PySpark Environment

To illustrate the power of this group selection technique, we will utilize a sample dataset representing basketball player statistics. This dataset is structured to contain multiple entries for the same team, making it a perfect test case for isolating a single, representative row for each team entity. Our primary objective is to initialize the Spark session, define and load the sample

data, and then apply the Window function to achieve the desired deduplication based on the group key.

We begin by setting up the necessary PySpark environment and defining a list of data rows. Our sample data includes the columns 'team', 'position', and 'points'. Note that 'Team A' and 'Team B' both contain several redundant records, which necessitates the robust grouping mechanism provided by Window functions to select only one row per team.

The initial setup and data creation steps provided below establish a fully functional DataFrame, ready for the transformation process. Running this code ensures that the Spark session is active and the sample data is correctly structured in a distributed format:

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
+----+--------+------+

Applying the Window Function for First Row Selection

With the data loaded, the next step is applying the core transformation logic. We define the Window specification `w`, explicitly instructing PySpark to segment the data by the 'team' column using partitionBy. For this initial example, we use a constant value in the `orderBy` clause (`lit('A')`) which results in an arbitrary selection of the first row encountered by the Spark executor for that partition.

We then introduce a new column named 'row' using the row_number function applied over our defined window `w`. This step assigns the rank 1 to the first record within each team partition. The final step is a simple filter operation, selecting only those rows where the newly created rank column equals 1, thus isolating one unique row per team.

Observe the transformation below. The output clearly validates the objective, showing only a single record for each unique team (A, B, and C). The method successfully manages the distributed nature of the data and efficiently selects the desired subset:

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 perfectly demonstrates the successful application of the Window function approach, yielding only the first row for each unique team value in the dataset while preserving all original column data.

Advancing Selection Logic: Deterministic Ordering and Multi-Column Grouping

While arbitrary selection is useful for basic deduplication, most professional data science tasks require the "first" row to be determined by a specific business logic, such as the most recent timestamp or the maximum score. The true utility of the Window function pattern is its ability to incorporate a precise `orderBy` clause within the Window specification, which dictates exactly which row receives the rank of 1.

For instance, if the requirement is to select the player record with the maximum points for each team, the `orderBy` clause would be modified to sort by 'points' in descending order (`orderBy(col("points").desc())`). This modification ensures that the row with the highest points value receives rank 1 within its team partition, guaranteeing a deterministic and meaningful selection. Conversely, finding the minimum points record simply requires an ascending sort order. Furthermore, the `orderBy` clause can include multiple columns to handle tie-breaking scenarios (e.g., sorting by points descending, then by position ascending).

The grouping mechanism itself can also be extended. If the unique identifier for your use case requires a combination of fields--for example, selecting the first row for every unique combination of 'team' and 'position'--you simply include all relevant grouping columns within the partitionBy function. This flexibility is critical for managing complex hierarchical data structures where sub-grouping is necessary to correctly scope the ranking operation.

Grouping by Multiple Columns: To partition the data based on multiple unique identifiers, include all relevant column names as arguments in the **`partitionBy`** function. For example: `Window.partitionBy('team', 'position').orderBy(...)`.

Understanding Arbitrary Ordering: In cases where the selection must be truly non-deterministic (i.e., you genuinely do not care which row is selected first), the syntax **`orderBy(lit('A'))`** can be used. This tells PySpark to apply a constant value for ordering, yielding an arbitrary selection based on how Spark encounters the data internally. However, for production systems requiring consistent, reproducible results, always use a meaningful column in **`orderBy`** to define the 'first' row explicitly.

Conclusion: Best Practices for Robust Data Filtering

Selecting a representative row from a distributed group is a foundational task in data engineering, and it is handled elegantly and efficiently in PySpark via the combination of Window functions and the row_number function. This methodology is superior to standard SQL aggregation techniques when the primary goal is to preserve the full structural information of the selected record.

When implementing this technique, data engineers must always ensure the `orderBy` clause accurately reflects the business definition of "first." While using `lit()` might suffice for quick, arbitrary sampling, defining a specific sort order is essential for producing reliable and reproducible results across distributed execution environments, especially when dealing with time-series or high-stakes metric data.

A crucial best practice involves maintaining data cleanliness: always remember to drop the temporary rank column (`.drop('row')`) in our examples) immediately after the filtering step. This ensures the output DataFrame contains only the original, required fields, minimizing unnecessary data overhead and preparing the result set for subsequent transformations or storage. Mastering this Window function pattern unlocks significant capabilities in advanced data manipulation within the Spark ecosystem.

Additional Resources

The following related tutorials provide further insights into common data manipulation tasks in PySpark:

PySpark: How to Group By Multiple Columns

PySpark: Selecting the Last Row of Each Group

Introduction to PySpark Window Functions