How can I select the first row by group using dplyr?

Authored by

stats writer

December 20, 2025

In modern data analysis using R, the ability to efficiently manipulate and summarize data is paramount. One common requirement in data preprocessing is selecting a specific row--often the first, last, or Nth row--from subsets defined by a grouping variable. This task is streamlined perfectly by the dplyr package, a cornerstone of the Tidyverse ecosystem.

The dplyr package provides a powerful and readable grammar for data manipulation. When dealing with grouped operations, the combination of several key functions allows users to precisely isolate the required record within each defined subset. Whether you are looking for the earliest transaction, the smallest score, or simply the first entry based on the original data structure, the methodology remains consistent and highly efficient.

To select the very first observation within each group, we leverage a sequence of operations: first defining the groups, then establishing a clear order within those groups, and finally, filtering down to the row that meets our criteria. The syntax is concise, reflecting the clean and expressive nature of the R language when powered by dplyr.

The Core Syntax for Selecting the First Row

The fundamental technique for extracting the leading observation per group involves chaining three essential verbs from the dplyr toolkit: **`group_by()`**, **`arrange()`**, and **`filter()`**. This combination ensures that we partition the data frame, sort the data according to a specified variable, and then select the top record based on its rank within its respective group.

The **`group_by()`** function is critical; it temporarily transforms the data structure so that subsequent operations are applied "by group" rather than to the entire dataset. Following this, **`arrange()`** dictates the internal order of the rows within each group. This ordering is crucial because the "first" row is only meaningful once a sort order is established. Finally, the **`filter()`** function uses the special window function **`row_number()`** to identify and keep only the first row (where the row number equals one) from each group.

The generalized syntax for this operation is shown below. Here, `group_var` represents the column used to define the groups, and `values_var` represents the column used to sort the rows within those groups. By default, **`arrange()`** sorts in ascending order, meaning we are effectively selecting the row with the minimum value for `values_var` in that group.

df %>%
group_by(group_var) %>%
arrange(values_var) %>%
filter(row_number()==1)

This powerful sequence of functions is the bedrock of many data aggregation and cleaning tasks. The pipe operator (`%>%`) ensures a clean, sequential flow of data manipulation, making the code highly readable and maintainable for complex data workflows.

Understanding Grouping and Ordering

Before diving into the specific example, it is essential to appreciate how grouping and ordering interact. The **`group_by()`** function does not alter the underlying data order; it merely annotates the data structure with grouping information. The real sorting happens with the **`arrange()`** function.

When **`arrange()`** is applied to a grouped data frame, it sorts the rows independently within each group. This sorting determines which row becomes the 'first' or 'top' row for that group. If we sort by 'points' ascending, the row with the minimum points will be designated as row number one. If we sort by 'points' descending, the row with the maximum points will be row number one.

The final piece, **`row_number()`**, is an analytic function that assigns a sequential integer rank to each row within its current group, starting at 1. By combining **`filter(row_number() == 1)`**, we explicitly tell dplyr to retain only those rows that hold the rank of one after sorting. This methodology ensures robust and predictable row selection across diverse datasets.

Practical Demonstration: Setting Up the Data Frame

To illustrate this process clearly, let us establish a sample dataset. This data frame, named `df`, contains two variables: `team` (the grouping variable) and `points` (the variable used for sorting and selection). This structure is typical of many datasets where we need to find the extreme or ranked values associated with specific categories.

The creation of the sample data involves standard R syntax, utilizing the **`data.frame()`** constructor. Notice how the teams A, B, and C have varying numbers of observations and corresponding point values. The goal is to select just one row per team based on the point values.

Below shows the code used to generate the dataset and the resulting structure, highlighting the heterogeneity within the groups before any manipulation begins.

#create dataset
df <- data.frame(team=c('A', 'A', 'A', 'B', 'B', 'B', 'C', 'C', 'C', 'C'),
points=c(4, 9, 7, 7, 6, 13, 8, 8, 4, 17))

#view dataset
df

team points

1 A 4
2 A 9
3 A 7
4 B 7
5 B 6
6 B 13
7 C 8
8 C 8
9 C 4
10 C 17

Implementation 1: Selecting the First Row (Minimum Value)

Our first concrete example demonstrates how to select the row associated with the lowest score for each team. Since **`arrange()`** defaults to ascending order, sorting by `points` will place the minimum point value at the top (row number 1) of its group.

The process begins by loading the necessary library, **`dplyr`**. We then pipe the `df` object through the **`group_by()`** function, specifying `team` as the grouping variable. This partitions the data into three logical subsets (A, B, and C). The subsequent **`arrange()`** call sorts these subsets based on `points`.

Finally, **`filter(row_number() == 1)`** extracts the newly designated "first" row from each of the three sorted groups. The resulting output is a smaller data frame (or tibble, as returned by dplyr operations) containing only one record per team, representing the lowest point score achieved by that team.

library(dplyr)

df %>%
group_by(team) %>%
arrange(points) %>%
filter(row_number()==1)

A tibble: 3 x 2
Groups: team
team points

1 A 4
2 C 4
3 B 6

Implementation 2: Selecting the First Row in Descending Order (Maximum Value)

What if the requirement is to find the row corresponding to the **maximum** score for each team? The methodology is nearly identical, but we introduce the special helper function **`desc()`** within the **`arrange()`** call.

The **`desc()`** function forces the sorting to occur in descending order. This means the row containing the highest value of `points` will be moved to the top position (row number 1) within its respective team group. By keeping the subsequent **`filter(row_number() == 1)`** step unchanged, we successfully isolate the maximum value row for every team.

This minor modification demonstrates the flexibility and power of combining dplyr verbs. We are effectively solving the "find the maximum value and the corresponding row by group" problem using the same foundational structure designed for selecting the first row.

df %>%
group_by(team) %>%
arrange(desc(points)) %>%
filter(row_number()==1)

A tibble: 3 x 2
Groups: team
team points

1 C 17
2 B 13
3 A 9

Advanced Selection Techniques: Targeting the Nth Row

The beauty of using the **`row_number()`** function is that it provides access to every row's rank within the group, not just the first. This allows for simple and elegant modification to select the Nth row, where N is any positive integer. If you want the 2nd row, simply change the filter condition to `row_number() == 2`.

It is crucial to remember that the Nth row selection still depends entirely on the sorting applied by **`arrange()`**. If you are sorting in descending order of points, selecting the 2nd row will give you the second highest score for that team. This method grants granular control over ranked data extraction.

For example, if we maintain the descending sort order (highest points first) but modify the filter to select the 2nd row, we retrieve the second-highest score for each team:

```
df %>%
group_by(team) %>%
arrange(desc(points)) %>%
filter(row_number()==2)
```

Targeting the Last Row within a Group

Selecting the last row is a common variation of selecting the first. While one could achieve this by sorting in reverse and then taking the first row, dplyr provides an even more intuitive and robust method using the window function `n()`.

The `n()` function, when used within a grouped context, returns the total number of rows (observations) in the current group. Therefore, setting the filter condition to `row_number() == n()` ensures that we only select the row whose rank matches the size of its group, effectively selecting the very last row defined by the current sorting scheme.

If we continue using the descending sort on `points`, selecting `row_number() == n()` will identify the row with the lowest point score for each team, since the lowest score is placed last in the descending sort order. This flexibility ensures that users can easily isolate either extreme end of the ordered data.

```
df %>%
group_by(team) %>%
arrange(desc(points)) %>%
filter(row_number()==n())
```

Comparison to Alternative Methods

While the combination of `group_by()`, `arrange()`, and `filter()` is the most expressive method in the R ecosystem, it is worth noting that other methods exist. For instance, some users might try to achieve similar results using base R functions or alternative Tidyverse functions like `slice()` or `slice_min()`/`slice_max()`.

The `slice()` function offers a shorthand alternative to the `filter(row_number() == ...)` pattern. When used on a grouped data frame, `slice(1)` achieves the exact same result as selecting the first row after sorting. Similarly, `slice(n())` selects the last row. However, understanding the `filter(row_number())` method provides a deeper insight into how ranking and

filtering work under the hood, offering more flexibility for complex ranking conditions (e.g., selecting rows 1 through 3, or filtering based on relative rank).

Ultimately, the `group_by()` + `arrange()` + `filter()` approach is often favored by expert practitioners for its explicit control over the ordering variable, ensuring that the selection criterion is crystal clear to anyone reading the code. It is an indispensable technique for robust data wrangling.

Further Reading on Data Manipulation in R

How to Arrange Rows in R
How to Count Observations by Group in R
How to Find the Maximum Value by Group in R