How to Filter DataFrames in dplyr While Keeping Rows with NA Values

Authored by

stats writer

November 21, 2025

Data cleaning and manipulation are foundational tasks in any data science workflow, and the **dplyr** package in **R** provides powerful, readable tools for these operations. One of the most frequently used functions is **filter()**, which allows users to subset rows based on specific logical conditions. However, when working with real-world datasets, encountering **NA values** (Not Available or missing data) is inevitable, and these missing observations often complicate standard filtering operations.

A common requirement is to exclude rows based on a specific criterion (e.g., excluding 'Team A'), yet simultaneously ensure that rows containing **NA values** in that filtering column are retained. Standard logical filtering in R, particularly within **dplyr**, typically evaluates any comparison involving an NA as NA itself. Since **dplyr::filter()** only keeps rows where the condition evaluates to **TRUE**, any row resulting in **NA** is silently dropped. This behavior, while logical in computational terms, often runs contrary to the analyst's intent when dealing with partially complete records.

This guide explores a robust and clean method to overcome this challenge, allowing you to selectively exclude non-matching entries while explicitly preserving all rows where the criterion column contains missing data. This technique leverages the complementary strengths of the **tidyr** package, specifically the **replace_na()** function, integrated seamlessly into the **dplyr** pipeline.

Understanding Default Behavior in `dplyr::filter()`

Before implementing the solution, it is essential to understand why standard filtering drops rows containing **NA values**. In **R**, logical operations follow the rules of three-valued logic (TRUE, FALSE, and NA). When you apply a condition like `column != 'Value'`, if the value in `column` is **NA**, the result of the comparison is also **NA**.

The **filter()** function is designed to retain only those rows for which the supplied logical expression evaluates strictly to **TRUE**. Consequently, any row where the condition resolves to **FALSE** or **NA** is automatically discarded from the resulting **data frame**. This default behavior ensures that the output contains only observations that definitively satisfy the filter criteria.

However, for data cleaning or imputation tasks, dropping these missing records prematurely can lead to information loss or biased analysis. Our goal is to manipulate the result of the logical comparison specifically for **NA values**, ensuring that they are converted into **TRUE** within the context of the **filter()** function, thereby forcing their retention.

Prerequisites: Required Packages (`dplyr` and `tidyr`)

To execute the robust filtering technique demonstrated here, you must have two core packages from the Tidyverse installed and loaded: **dplyr** and **tidyr**. The **dplyr** package provides the filtering

mechanism itself, while the **tidyr** package offers the crucial function necessary to handle the **NA values** within the logical vector created by the filter condition.

The **tidyr** function we will rely on is **replace_na()**. This function is specifically designed to replace missing values (**NA**) in a vector with a specified replacement value. When applied to the logical vector generated by the **filter()** condition, we can tell **R** to interpret the **NA** results as **TRUE**, effectively bypassing the default dropping mechanism of **dplyr**.

The following basic syntax outlines how to filter a **data frame** without losing rows that contain **NA values** using functions from the **dplyr** and **tidyr** packages in R:

```
library(dplyr)
library(tidyr)

#filter for rows where team is not equal to 'A' (and keep rows with NA)
df <- df %>% filter((team != 'A') %>% replace_na(TRUE))
```

Note that this formula uses the **replace_na()** function from the **tidyr** package to convert **NA values** generated by the comparison to **TRUE** so they aren't dropped from the data frame when filtering. This critical step ensures that missing records are retained in the output.

Practical Demonstration: Setting up the Sample Data Frame

To illustrate this technique practically, we will construct a small sample **data frame** representing basketball player statistics. This dataset includes three columns: `team`, `points`, and `assists`, and critically, it incorporates intentional **NA values** in the `team` column to mimic real-world data imperfections.

The creation process involves using the base **R** function `data.frame()`, ensuring we explicitly include `NA` entries where appropriate. This setup allows us to precisely track how missing values are handled during different filtering processes. We are primarily interested in filtering based on the `team` column.

The following example shows how to use this syntax in practice, beginning with the creation of the sample dataset:

```
#create data frame
df <- data.frame(team=c('A', NA, 'A', 'B', NA, 'C', 'C', 'C'),
points=c(18, 13, 19, 14, 24, 21, 20, 28),
assists=c(5, 7, 17, 9, 12, 9, 5, 12))
```

#view data frame
df

team points assists
1 A 18 5
2 <NA> 13 7
3 A 19 17
4 B 14 9
5 <NA> 24 12
6 C 21 9
7 C 20 5
8 C 28 12

Scenario 1: Default Filtering (The Problem)

Now suppose we use the **filter()** function from the **dplyr** package to filter the data frame to only contain rows where the value in the **team** column is not equal to A. This is the default, problematic behavior when dealing with missing data.

As anticipated, when **dplyr** evaluates the condition `team != 'A'` on rows 2 and 5 (where team is **NA**), the result is NA, leading to these rows being automatically dropped from the output. This illustrates the data loss that occurs when missing values are not explicitly handled during filtering.

The resulting dataset below clearly shows that both Team 'A' entries and the **NA values** have been eliminated:

library(dplyr)

#filter for rows where team is not equal to 'A'
df <- df %>% filter(team != 'A')

#view updated data frame
df

team points assists
1 B 14 9
2 C 21 9
3 C 20 5
4 C 28 12

Notice that each row where the value in the **team** column is equal to A has been filtered out,

including the rows where the value in the **team** column is equal to NA.

Scenario 2: Applying the `replace_na()` Technique (The Solution)

If we would like to filter out the rows where **team** is equal to A and *keep* the rows with **NA values**, we must integrate the **tidyr** package's **replace_na()** function into our **dplyr** pipeline. This ensures that any time the filtering condition results in **NA**, it is immediately converted to **TRUE** before being evaluated by **filter()**.

The key modification is wrapping the logical comparison `(team != 'A')` and then piping the resulting logical vector into `%>% replace_na(TRUE)`. This strategy provides a clean, expressive, and reliable method for retaining missing data during exclusion filtering.

The resulting dataset now correctly excludes Team 'A' while preserving the records where the team affiliation was unknown:

library(dplyr)
library(tidyr)

#filter for rows where team is not equal to 'A' (and keep rows with NA)
df <- df %>% filter((team != 'A') %>% replace_na(TRUE))

#view updated data frame
df

team points assists
1 <NA> 13 7
2 B 14 9
3 <NA> 24 12
4 C 21 9
5 C 20 5
6 C 28 12

Notice that each row where the value in the **team** column is equal to A has been filtered out, but we kept the rows where the value in the **team** column is equal to NA.

Conclusion and Further Resources

The combination of **dplyr's** powerful chaining operator (%>%) and **tidyr's** specialized handling of missing data provides a clean and highly effective solution for managing **NA values** during conditional filtering. By leveraging **replace_na(TRUE)**, data analysts gain precise control over

which records are kept, ensuring that valuable information contained within partially complete records is not inadvertently lost due to the technicalities of three-valued logic in R.

Mastering this technique is crucial for maintaining data integrity during complex data preparation stages. It ensures that the subsequent analysis, visualization, or modeling steps receive a **data frame** that accurately reflects the intended population, including those observations where the filtering variable is missing.

Note: You can find the complete documentation for the tidyr **replace_na()** function online.

The following tutorials explain how to perform other common functions in **dplyr**: