In R, how do you group data by hour of day?

Authored by

stats writer

November 21, 2025

Introduction to Time-Based Grouping in R

Analyzing time-series data often requires grouping observations into meaningful temporal bins, such as grouping data by the hour of day, day of week, or month. In the statistical programming language, R, achieving this precise hourly aggregation demands specific functions that can properly truncate or floor the timestamp to the desired interval. While base R offers methods like the `cut()` function, which segments data based on specified intervals, these methods can sometimes be cumbersome when dealing with continuous time variables and require careful management of factor levels. A more efficient and modern approach leverages powerful packages from the **Tidyverse** ecosystem, notably **dplyr** and **lubridate**, to streamline the process of temporal data manipulation and aggregation.

The challenge in grouping by hour lies in converting a precise datetime stamp (e.g., `2022-01-01 01:14:00`) into a standardized hourly marker (e.g., `2022-01-01 01:00:00`). This standardization ensures that all events occurring within that 60-minute window are correctly allocated to the same group. Once standardized, standard data aggregation techniques, such as summing or averaging, can be applied seamlessly. We will focus on utilizing `lubridate`'s functions in conjunction with `dplyr`'s powerful `group_by()` and `summarize()` capabilities to execute this task cleanly and efficiently.

This sophisticated method not only simplifies the grouping logic but also maintains the integrity of the time variable, keeping it in the appropriate datetime format--often the **POSIXct** class in R--which is essential for subsequent analyses or visualizations. Understanding the interaction between these packages is fundamental for anyone working extensively with temporal data sets, offering substantial improvements in code readability and processing speed compared to older, more convoluted techniques.

Why Standard Aggregation Requires Temporal Manipulation

When working with transaction or sensor data, observations are rarely perfectly aligned on the hour; they possess minute and second resolution. If we attempt to group raw datetime columns directly using standard functions, R will treat every unique timestamp as its own separate group, effectively preventing any meaningful aggregation. For example, a sale recorded at 1:05 PM is distinct from one recorded at 1:15 PM if the full timestamp is used for grouping, even though both belong logically to the 1 PM hour bin.

To overcome this issue, we must perform a crucial data preprocessing step: truncating the time variable. Truncation, or "flooring," involves rounding down the timestamp to the nearest specified interval--in this case, the nearest hour. By replacing the minutes and seconds component of the datetime stamp with zeros, we create a canonical hourly marker. All events that fall between

1:00:00 and 1:59:59 will map to the 1:00:00 timestamp, allowing **dplyr** to correctly identify them as belonging to the same group.

This technique is vital in various fields, especially those relying on high-frequency data, such as financial trading, environmental monitoring, or e-commerce analytics. It transforms raw, highly granular data into manageable time intervals suitable for reporting and trend analysis. Without this intermediate step, detailed temporal analyses would be impossible, as the data would remain too fragmented for effective summarization.

Leveraging the Tidyverse: dplyr and lubridate

The most robust solution for time-based aggregation in R involves combining two highly effective packages from the Tidyverse: dplyr for data manipulation and lubridate for dealing with dates and times. **dplyr** provides the scaffolding for efficient data processing using functions like `group_by()` and `summarize()`, which are central to any aggregation task.

The key innovation comes from the **lubridate** package, specifically the `floor_date()` function. This function is designed explicitly to round down a date-time object to the nearest boundary of a specified unit (e.g., hour, day, month). When we pass the time column to `floor_date()` and specify `"1 hour"` as the unit, it instantly normalizes all timestamps within that hour to the start of the hour, creating the required grouping key.

The combination of these tools results in elegant, readable, and highly efficient code, adhering to modern R programming best practices. The syntax is intuitive, clearly communicating the intent: "Take this data frame, group the time column by flooring it to the nearest hour, and then calculate a summary statistic." This seamless workflow is preferred over lower-level base R manipulations for complex data frame operations.

The Core Syntax for Hourly Aggregation

To execute time-based grouping and subsequent aggregation, the following structure is utilized. This example demonstrates how to group the values by hour in a column named `time` and then calculate the total sum of values in the `sales` column for each hourly period. This approach is highly flexible and can be adapted for any metric (e.g., mean, count, maximum) simply by changing the function within `summarize()`.

The pipe operator (`%>%`) chains the operations together: first, the data frame is passed to `group_by()`, where the time variable is transformed using `floor_date()`; subsequently, the grouped data is passed to `summarize()` to compute the desired statistic for each newly created hourly group.

library(dplyr)
library(lubridate)

#group by hours in time column and calculate sum of sales
df %>%
group_by(time=floor_date(time, '1 hour')) %>%
summarize(sum_sales=sum(sales))

In this specific syntax block, the `floor_date(time, '1 hour')` function call is instrumental. It creates a new grouping variable named `time` (overwriting the original column for the purpose of grouping, but not in the original data frame) where every timestamp is rounded down to the beginning of the hour. Once the groups are established, the summarize() function calculates `sum_sales`, providing a concise summary of the volume of activity observed during each complete hour.

Practical Demonstration: Setting Up the Sales Data

To illustrate this methodology concretely, let us consider a hypothetical scenario where we track the number of sales made at a retail store across a few hours of operation. The initial data frame, which we will call `df`, contains two columns: `time`, which stores the precise moment of the transaction, and `sales`, which records the quantity sold in that transaction. It is critical that the `time` column is properly formatted as a date-time object, specifically the POSIXct class, for `lubridate` functions to work correctly.

The following R code block demonstrates the creation of this sample data frame, ensuring the time column is correctly coerced using `as.POSIXct()`. This step is often necessary when importing raw data that might initially treat time stamps as plain character strings.

#create data frame
df <- data.frame(time=as.POSIXct(c('2022-01-01 01:14:00', '2022-01-01 01:24:15',
'2022-01-01 02:52:19', '2022-01-01 02:54:00',
'2022-01-01 04:05:10', '2022-01-01 05:35:09')),
sales=c(18, 20, 15, 14, 10, 9))

#view data frame
df

time sales
1 2022-01-01 01:14:00 18
2 2022-01-01 01:24:15 20
3 2022-01-01 02:52:19 15

4 2022-01-01 02:54:00 14
5 2022-01-01 04:05:10 10
6 2022-01-01 05:35:09 9

Observe the variance in the minutes and seconds component of the `time` column. For instance, the first two entries occur within the first hour (starting at 1:00:00), while the next two occur within the second hour (starting at 2:00:00). If we were to use the raw `time` column for grouping, we would end up with six distinct groups, which is not the desired outcome for hourly data aggregation. This setup perfectly demonstrates the need for temporal flooring.

Step-by-Step: Calculating Total Sales Per Hour

With the data frame established, we can now apply the core methodology combining **dplyr** and **lubridate** to group the `time` column by hour and subsequently calculate the sum of `sales` for each resulting group. This process effectively converts high-granularity transactional data into hourly throughput metrics, providing immediate insight into periods of peak activity.

We begin by loading the necessary libraries: dplyr for piping and summarizing operations, and lubridate for time manipulation. The primary operation resides within the `group_by()` function, where `floor_date(time, '1 hour')` standardizes all timestamps. Finally, the summarize() function collapses the groups and computes the total sales volume (`sum(sales)`), renaming the resulting column `sum_sales`.

library(dplyr)
library(lubridate)

#group by hours in time column and calculate sum of sales
df %>%
group_by(time=floor_date(time, '1 hour')) %>%
summarize(sum_sales=sum(sales))

`summarise()` ungrouping output (override with `.groups` argument)
A tibble: 4 x 2
time sum_sales

1 2022-01-01 01:00:00 38
2 2022-01-01 02:00:00 29
3 2022-01-01 04:00:00 10
4 2022-01-01 05:00:00 9

The output is a new data frame (specifically, a tibble in the Tidyverse context) containing the aggregated results. Notice that the original six rows have been consolidated into four rows, corresponding to the four distinct hours during which sales occurred. The `time` column in the resulting tibble now exclusively shows the start of the hourly interval, confirming the successful application of the `floor_date()` function in creating the accurate temporal bins for aggregation.

Interpreting the Aggregated Results

The summarized output provides a clear, high-level view of sales activity broken down by the hour of the day. Each row represents a complete hour, starting at the time indicated in the first column. For analytical purposes, this aggregated view is far more useful than the raw transaction log, allowing for immediate comparisons between different operational periods.

Based on the summation output, we can draw the following key conclusions about the store's performance during these hours:

A total of **38** sales were made during the hour starting at 2022-01-01 01:00:00. This is the sum of the 18 and 20 units sold in the first two transactions.

A total of **29** sales were made during the hour starting at 2022-01-01 02:00:00. This combines the 15 and 14 units sold during that interval.

A total of **10** sales were made during the hour starting at 2022-01-01 04:00:00.

A total of **9** sales were made during the hour starting at 2022-01-01 05:00:00.

This visualization of hourly throughput immediately highlights that the earliest operating hours (01:00 and 02:00) generated significantly higher sales volumes than the later hours, suggesting a potential early surge in customer activity or perhaps a targeted promotional period. Such insights are crucial for operational planning, inventory management, and staffing decisions.

Expanding Analysis: Calculating the Mean Sales Per Hour

While calculating the total sum is often the first step in time-series aggregation, the same robust hourly grouping structure can be easily adapted to compute any other required metric, such as the mean, median, or standard deviation. For instance, calculating the **mean** number of sales per hour provides insight into the average transaction size or performance rate within that interval.

To switch from summing to averaging, only the function within the summarize() call needs modification, replacing `sum(sales)` with `mean(sales)`. The grouping logic remains identical, ensuring that the mean is calculated only across transactions correctly assigned to that specific hourly bin.

```
library(dplyr)
library(lubridate)

#group by hours in time column and calculate mean of sales
df %>%
group_by(time=floor_date(time, '1 hour')) %>%
summarize(mean_sales=mean(sales))

`summarise()` ungrouping output (override with `.groups` argument)
# A tibble: 4 x 2
time mean_sales

1 2022-01-01 01:00:00 19
2 2022-01-01 02:00:00 14.5
3 2022-01-01 04:00:00 10
4 2022-01-01 05:00:00 9
```

The resulting output confirms the new metric calculation. For the first hour, the mean sales were calculated as (18 + 20) / 2 = **19**. For the second hour, the mean sales were (15 + 14) / 2 = **14.5**. In contrast, the fourth and fifth hours saw only one transaction each, meaning the mean sales equal the single recorded sales value for those hours (10 and 9, respectively).

The mean sales made in the first hour were **19**.

The mean sales made in the second hour were **14.5**.

The mean sales made in the fourth hour were **10**.

The mean sales made in the fifth hour were **9**.

This adaptability underscores the power of using the `floor_date()` function within the `group_by()` framework. Researchers and analysts are encouraged to experiment by modifying the metric in the summarize() function to calculate any specific metric necessary for their data visualization or modeling requirements, whether it is maximum latency, count of events, or weighted averages.

Conclusion: Mastering Time-Series Grouping in R

Grouping time-series data by the hour of the day in R is a fundamental requirement for effective temporal analysis. By integrating the functionality of the lubridate and dplyr packages, data scientists can perform this complex task using concise and highly readable code. The `floor_date()` function serves as the central mechanism, transforming granular timestamps into

standardized hourly bins, which subsequently enables accurate and efficient data aggregation via the `group_by()` and `summarize()` sequence.

This methodology ensures robustness, regardless of whether the goal is to calculate cumulative metrics like total sales or descriptive statistics such as the mean or median. Mastering this approach provides a solid foundation for handling more sophisticated time-based analyses, including rolling aggregations or comparisons across varying temporal granularities. This technique is applicable across diverse datasets, from financial market movements to server log analysis, making it an essential skill for any advanced R user dealing with temporal data.

Feel free to apply this powerful methodology to your own data frame. By simply modifying the time interval (e.g., changing `'1 hour'` to `'30 minutes'` or `'1 day'`) and adjusting the function within `summarize()`, you can tailor the aggregation to meet precise analytical objectives for your specific business or research needs.