Learning to Benchmark R Code: Measuring Execution Time with the microbenchmark Package

Authored by

Mohammed looti

March 3, 2026

In the world of data science and statistical computing using R, code efficiency is not merely an academic concern; it directly impacts resource consumption, processing speed, and the scalability of analytical pipelines. When analysts develop complex scripts or functions, they often encounter situations where multiple programming approaches yield the same final result. However, the internal efficiency--the true speed at which the computation is performed--can vary wildly between these approaches. Identifying the fastest method is critical, especially when working with massive datasets where even minor differences in execution time compound rapidly.

While simpler tools like `system.time()` offer a basic measurement of elapsed time, they often lack the precision and rigor required for accurate comparative analysis. These basic functions are susceptible to system noise, garbage collection cycles, and other environmental variables, leading to measurements that are often inconsistent and unreliable. When comparing two highly optimized functions that execute in milliseconds, a single trial measurement is insufficient to draw robust conclusions about performance superiority.

This is where specialized benchmarking tools become indispensable. A robust benchmarking process requires executing each target expression numerous times within a controlled environment, collecting a statistical distribution of the timing results, and then summarizing these metrics (such as minimum, mean, and median). Utilizing the **microbenchmark** package in R allows developers to make evidence-based decisions regarding algorithm selection and code optimization, ensuring that their production models run as quickly and efficiently as possible. The package provides the necessary structure to compare the execution time of various expressions with high precision and statistical reliability.

Introduction to the microbenchmark Package in R

The **microbenchmark** package is the leading solution in R for precise, repeatable timing measurements. Unlike methods that rely on single-run timing, **microbenchmark** ensures that each expression is evaluated multiple times (100 times by default), generating a statistical distribution of elapsed times rather than just a single point estimate. This approach significantly increases the confidence in the comparison between different computational strategies.

To begin utilizing this powerful tool, the package must first be loaded into the R session. The fundamental syntax involves calling the `microbenchmark()` function and passing the expressions you wish to compare as arguments. Each expression should be separated by a comma. The function handles the setup, repetitive execution, and collection of detailed timing statistics automatically, standardizing the comparison process.

The general structure for conducting a performance comparison is intuitive, allowing practitioners to quickly set up and run tests on any set of functions or code snippets. Understanding this basic structure is the first step toward rigorously evaluating the performance implications of different

coding choices in R:

library(microbenchmark)

#compare execution time of two different expressions
microbenchmark(
expression1,
expression2)
)

This framework is ideal for comparing anything from vectorized operations versus loop-based approaches to evaluating the speed differences between functions provided by Base R and those available in specialized packages like dplyr. We will now move to a practical example showcasing this comparison.

Setting Up the Comparative Example Data Structure

To illustrate the practical application of **microbenchmark**, we will construct a common scenario in data analysis: calculating summary statistics grouped by a categorical variable. For this demonstration, we will create a simulated dataset focusing on points scored by players across hypothetical basketball teams. Ensuring the reproducibility of data creation is a crucial preliminary step in any scientific or analytical exercise, which we achieve using `set.seed()`.

Our sample dataset, implemented as an R **data frame**, contains 1,000 observations, evenly split between two teams ('A' and 'B'). The 'points' variable is generated using a normal distribution centered around 20. This structure mimics typical real-world data where performance comparisons are most relevant--specifically, operations involving grouping and aggregation across a moderately sized dataset. The primary goal is to calculate the mean points scored for each team using two distinct methods.

Below is the R code used to generate and inspect the initial structure of the data frame. This setup ensures that both methods we compare--the Base R approach and the dplyr approach--operate on identical input data, guaranteeing a fair performance comparison:

#make this example reproducible
set.seed(1)

#create data frame
df <- data.frame(team=rep(c('A', 'B'), each=500),
points=rnorm(1000, mean=20))

#view data frame
head(df)

team points
1 A 19.37355
2 A 20.18364
3 A 19.16437
4 A 21.59528
5 A 20.32951
6 A 19.17953

With the data structure in place, we define the two methods for calculating the mean points segmented by team:

Method 1: The Base R Approach. This method relies on the powerful and native `aggregate()` function, a classic tool for splitting data into subsets, applying a function (like `mean`), and combining the results efficiently.
Method 2: The Tidyverse Approach. This method utilizes functions from the highly popular dplyr package, specifically chaining `group_by()` followed by `summarise_at()`. This approach is often favored for its readable, pipe-based syntax.

Executing the Benchmark: Comparing Base R vs. dplyr

The primary objective is to quantify which of these two aggregation strategies offers superior performance. We must load both necessary libraries (`microbenchmark` and `dplyr`) and then pass both expressions directly into the `microbenchmark()` function. By executing each expression 100 times, we ensure a statistically robust measurement of their inherent speed.

The code block below demonstrates the execution block. The first line uses the concise `aggregate()` function from Base R, while the second line uses the typical dplyr piping mechanism. The resulting output is a comprehensive table summarizing the performance statistics collected across all 100 trials.

library(microbenchmark)
library(dplyr)

#time how long it takes to calculate mean value of points by team
microbenchmark(
aggregate(df$points, list(df$team), FUN=mean),
df %>% group_by(team) %>% summarise_at(vars(points), list(name = mean))
)

Unit: milliseconds
expr
aggregate(df$points, list(df$team), FUN = mean)
df %>% group_by(team) %>% summarise_at(vars(points), list(name = mean))
min lq mean median uq max neval cld
1.307908 1.524078 1.852167 1.743568 2.093813 4.67408 100 a
6.788584 7.810932 9.946286 8.914692 10.239904 56.20928 100 b

The unit of measurement, reported in milliseconds, immediately signals that we are dealing with very small differences in time, emphasizing why repeated trials (`neval=100`) are essential. The table provides a complete statistical profile of the function performance, moving far beyond a simple single timing measurement and allowing for a nuanced understanding of speed consistency.

Interpreting the Detailed microbenchmark Output Metrics

The output generated by the `microbenchmark()` function is rich with statistical information, providing insight into both average speed and performance stability. Understanding each column is fundamental to accurately interpreting the performance profile of the benchmarked code.

The detailed metrics reported for each expression are as follows:

min: The minimum time recorded across all executions. This represents the "best-case scenario" performance.
lq: Lower quartile (25th percentile) time. 75% of the recorded runs were slower than this value.
mean: The arithmetic average execution time across all trials. While commonly used, the mean can be sensitive to rare, high-latency outliers.
median: The 50th percentile. This is generally considered a more robust measure of typical performance than the mean, as it is less affected by extreme outliers.
uq: Upper quartile (75th percentile) time. 25% of the recorded runs were slower than this value.
max: The maximum time recorded, highlighting the worst-case scenario performance.
neval: The number of times the expression was evaluated (100 in this case).
cld: The Comparison Letter Diagram, which indicates statistical differences. Expressions marked with different letters (e.g., 'a' vs. 'b') are statistically significantly different in performance.

For most practical purposes, performance assessment usually focuses on the **mean** or **median** time. By examining the results table, we can directly compare the typical performance of the two grouping methods:

The `aggregate()` function from Base R registered a mean time of **1.852 milliseconds**.

The dplyr pipeline recorded a substantially slower mean time of **9.946 milliseconds**.

Based purely on these numerical results, we conclude that the highly optimized `aggregate()` function from Base R is significantly faster than the equivalent dplyr workflow for this type of aggregation task on this dataset size. This difference is statistically confirmed by the distinct letters ('a' vs. 'b') in the `cld` column.

Visualizing Performance Differences Using Boxplots

While the tabular output provides precise numerical metrics, visualizing the full distribution of execution time is often the most effective way to communicate performance differences and stability. The **microbenchmark** object is designed to integrate seamlessly with R's graphical capabilities, enabling the creation of boxplots that summarize the minimum, quartiles, median, and maximum values for each tested expression.

Creating a boxplot from the **microbenchmark** results allows analysts to instantly grasp the magnitude of the difference in speed, and it visually highlights the variance or spread of the execution times. A method with a tighter, shorter boxplot indicates more consistent performance, which is valuable information for production systems where predictable timing is often as important as raw speed.

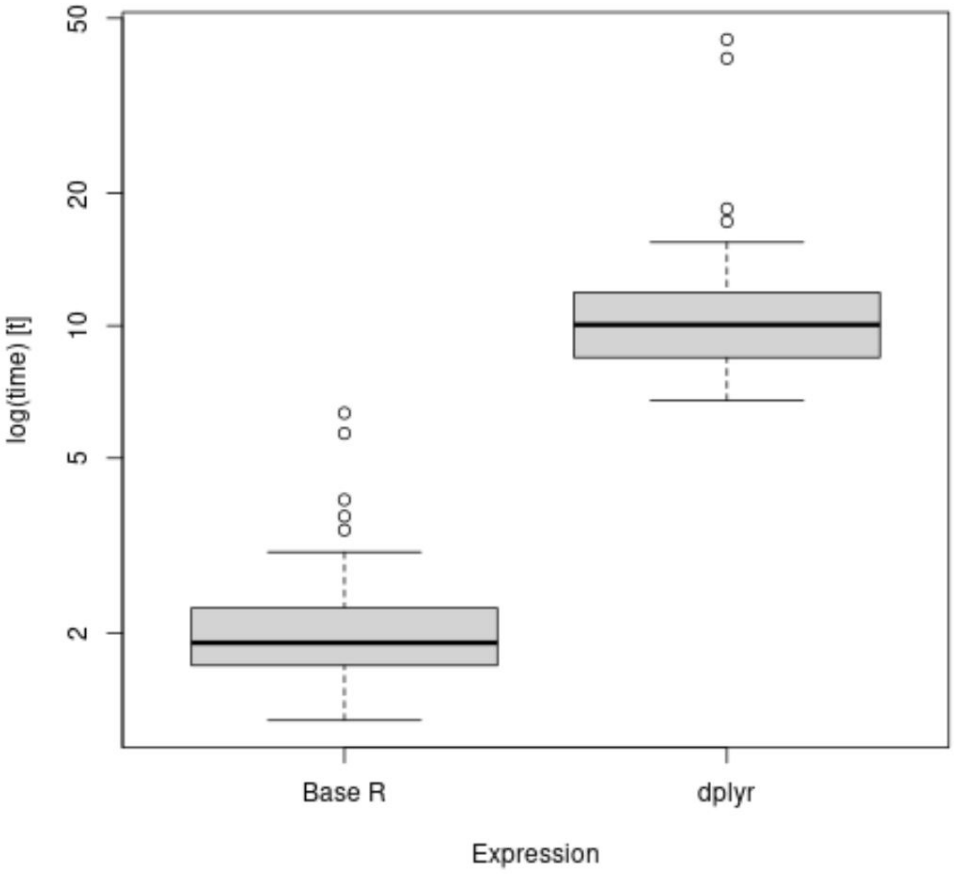
The following R code snippet demonstrates how to capture the results of the benchmark execution into a variable (`results`) and subsequently generate a comparative boxplot:

```
library(microbenchmark)
library(dplyr)

#time how long it takes to calculate mean value of points by team
results <- microbenchmark(
aggregate(df$points, list(df$team), FUN=mean),
df %>% group_by(team) %>% summarise_at(vars(points), list(name = mean))
)

#create boxplot to visualize results
boxplot(results, names=c('Base R', 'dplyr'))
```

The generated visualization clearly reinforces the quantitative findings.



From the boxplot, we observe that the Base R method's box is significantly lower and narrower than the dplyr method's box. This visual confirmation underscores that the Tidyverse approach not only takes longer on average but also exhibits a wider range of execution times, suggesting slightly less predictable performance compared to the highly optimized core function. Performance optimization is an ongoing task, and tools like **microbenchmark** provide the necessary empirical evidence to guide those decisions.

Additional Resources

The following tutorials explain how to perform other common tasks in R: