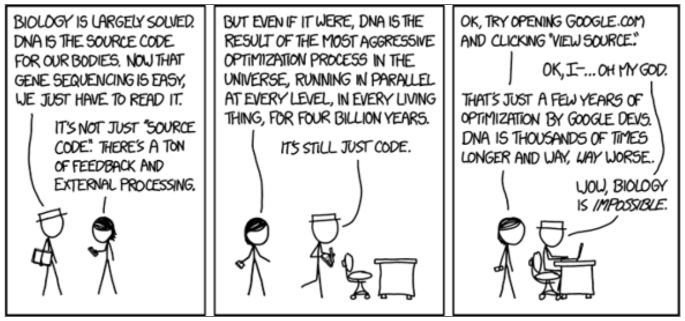
Maximizing Performance: Strategies for Code Optimization

Valeria Duran

What Is Optimization?

> An act, process, or methodology of making something (such as a design, system, or decision) as fully perfect, functional, or effective as possible. – Merriam-Webster

Code optimization is the process of enhancing code quality and efficiency.


BIOLOGY IS LARGELY SOLVED. DNA IS THE SOURCE CODE FOR OUR BODIES. NOW THAT GENE SEQUENCING IS EASY, WE JUST HAVE TO READ IT.
IT'S NOT JUST "SOURCE CODE." THERE'S A TON OF FEEDBACK AND EXTERNAL PROCESSING.
BUT EVEN IF IT WERE, DNA IS THE RESULT OF THE MOST AGGRESSIVE OPTIMIZATION PROCESS IN THE UNIVERSE, RUNNING IN PARALLEL AT EVERY LEVEL, IN EVERY LIVING THING, FOR FOUR BILLION YEARS.
IT'S STILL JUST CODE.
OK, TRY OPENING GOOGLE.COM AND CLICKING "VIEW SOURCE."
OK, I—... OH MY GOD.
THAT'S JUST A FEW YEARS OF OPTIMIZATION BY GOOGLE DEVS. DNA IS THOUSANDS OF TIMES LONGER AND WAY, WAY WORSE.
WOW, BIOLOGY IS IMPOSSIBLE.

xkcd comics

Pre-Optimization Steps

- Make the code work.
- Don’t repeat yourself.
- Write clean code and document it well.
- Don’t try to reinvent the wheel.

> “Make it work, then make it beautiful, then if you really, really have to, make it fast. **90 percent of the time, if you make it beautiful, it will already be fast.** So really, just make it beautiful!” –Joe Armstrong

Optimization should be the **final** step of your programming practice. Performance can be high following the pre-optimization steps. Only optimize when it’s necessary!

Steps to Optimize Code

1. Make sure code runs as expected, is clean, is well documented, and is created with the **end user** in mind.
2. Profile code (identify where the slow code is)
3. Find other solutions
    1. Vectorize code when possible
    2. Use parallelization techniques
    3. Cache frequently used data
    4. Manage memory
    5. Find a faster package/function
4. Benchmark code (compare code with your solution)
5. Execute!

Useful R Tools

- Profiling packages: `{profvis}` and `{profile}`.
- Benchmarking package: `{microbenchmark}` and `{bench}`.
- Caching packages: `{memoise}` (non-persistent by default) and `{R.cache}` (persistent).
- Parallel computing packages: `{snow}` and `{parallel}`.
- Out-of-memory data packages: `{ff}`, `{bigmemory}`, and `{feather}`.
- Use `gc()` to release memory (not needed, but it doesn’t hurt to use after removing large objects).
- Use `{data.table}` for faster computations.

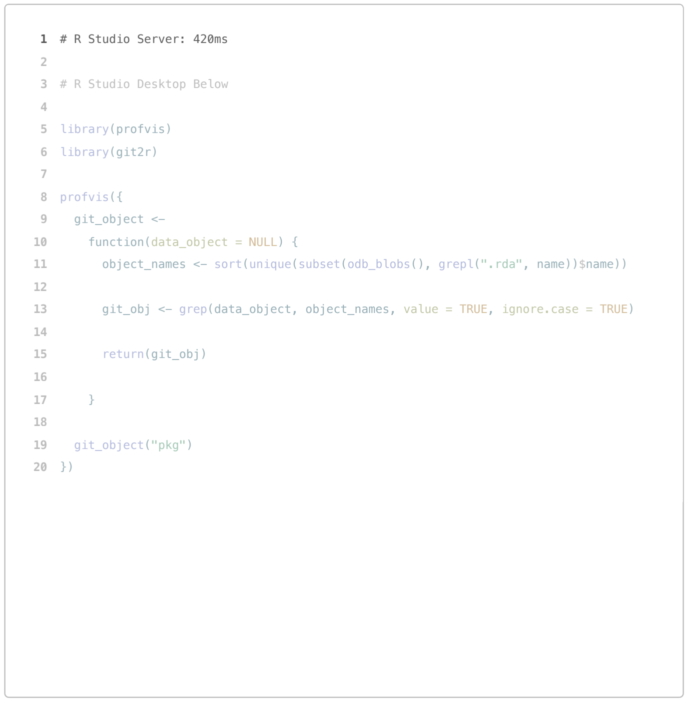
Scenario 1: Working With Big Data!

Goal: build a model using the insurance claims of ~25 million lives with two years' worth of data (i.e., billions of records!) to estimate the cost of a procedure.
Steps:

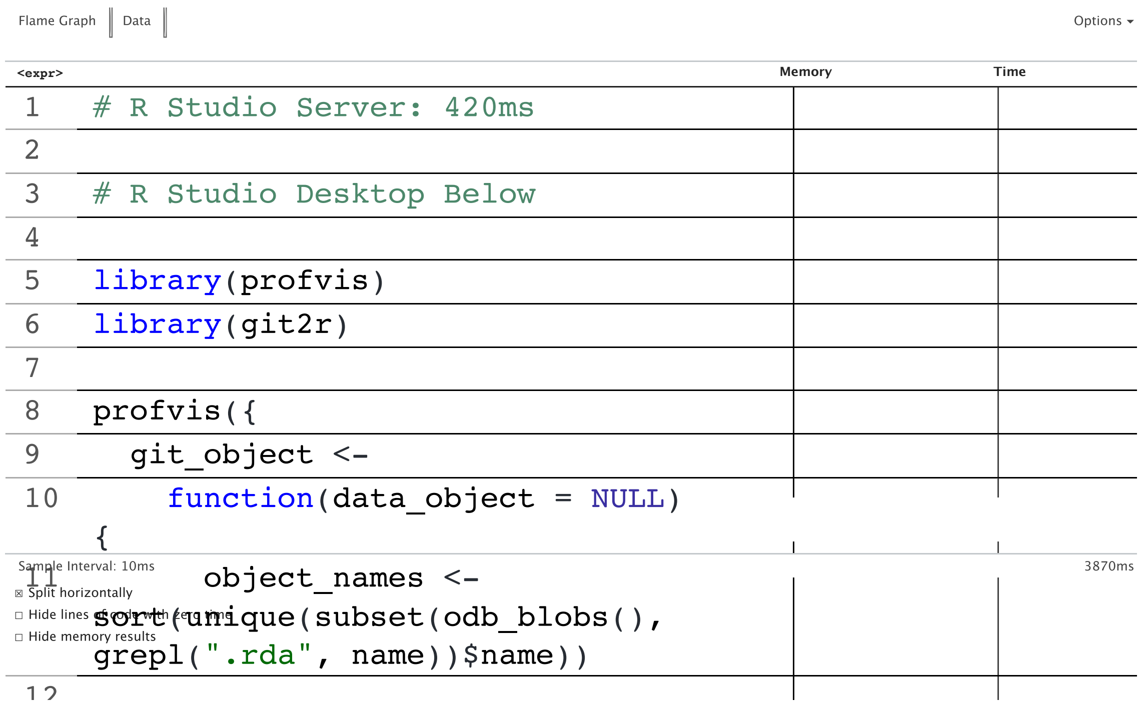
1. **Partition** data into categories/sections (body region). 👍
2. Create R scripts with tidyverse on a small dataset (Houston, Texas population for 1 year). 👍
3. Apply same scripts to a larger dataset (Texas population for 1 year). 👎
4. Update scripts to use data.table instead of tidyverse. 👍
5. Increase Windows instance (increasing RAM from 16GB to 32GB). 👍 👎
6. Run scripts on body regions (nationwide, and on two years of data). 👍

Took **months** to complete…
…Other solutions are also possible!

Scenario 2: End User in Mind



```r
# R Studio Server: 420ms

# R Studio Desktop Below

library(profvis)
library(git2r)

profvis({
  git_object <-
    function(data_object = NULL) {
      object_names <- sort(unique(subset(odb_blobs(), grepl(".rda", name))$name))

      git_obj <- grep(data_object, object_names, value = TRUE, ignore.case = TRUE)

      return(git_obj)

    }

  git_object("pkg")
})
```



<expr>	Memory	Time

```r
# R Studio Server: 420ms

# R Studio Desktop Below

library(profvis)
library(git2r)

profvis({
  git_object <-
    function(data_object = NULL)
{
      object_names <-
sort(unique(subset(odb_blobs(),
grepl(".rda", name))$name))
```

Sample Interval: 10ms 3870ms

☒ Split horizontally
☐ Hide lines of code with zero time
☐ Hide memory results

```r
library(microbenchmark)

microbenchmark(git_object <- sort(unique(subset(odb_blobs(), grepl(".rda", name))$name)),
               git_object2 <- sort(unique(gsub(".*/", "", system2("git" , "ls-files *.rda" , stdout = TRUE)))),
               times = 10)
```

```
Unit: milliseconds
                                                                                               expr
                          git_object <- sort(unique(subset(odb_blobs(), grepl(".rda", name))$name))
 git_object2 <- sort(unique(gsub(".*/", "", system2("git", "ls-files *.rda",      stdout = TRUE))))
       min        lq      mean    median        uq       max neval cld
 3232.7150 3699.6561 3977.4452 4029.9945 4328.2491 4557.8492    10  a
  233.1812  235.5819  249.6366  239.2543  243.3913  346.6082    10   b
```

… using `base::system2()` produces much faster results than `git2r::odb_blobs()`. Sometimes, what you want to use isn't always the best option for the end user!

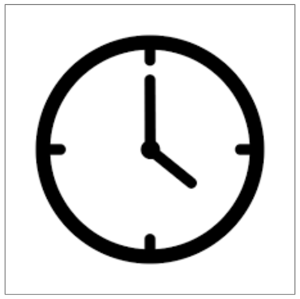
Can Optimizing Code Be a Bad Thing?

“The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; **premature optimization is the root of all evil (or at least most of it) in programming.”**
- Donald Knuth, Computer Programming as an Art

When Is Optimizing Code Bad?

- When code becomes less readable.
- When performance improvement is minuscule.
- When the time needed to optimize is longer than the task at hand.
- When code is not used frequently enough.

TIME!



Conclusion

- When writing code, consider the end user. What the majority will use might not be what you use. This will save you a lot of future rework.
- Trustworthy code is oftentimes better than fast code.
- Don’t optimize unless you absolutely must.
- Consider the biggest trade-offs: **time** and **effort**. Is it worth it?

Resources

- **Efficient R Programming**
- **Advanced R: Improving Performance**
- **97 Things Every Programmer Should Know**
- **Best Coding Practices in R**