Chapter 2
Profiling

2.1 Overview

It's very tempting as a developer of code, whether simply R code or shiny applications, to make a guess at where our applications are running slowly and spend time fixing it. It is possible that we might get lucky and happen to fix the right thing, but it is much more likely that we will spend lots of time trying to fix the wrong thing. Thankfully, there are tools that we can use to help us identify exactly where our code is slow, in particular the **profvis** package.

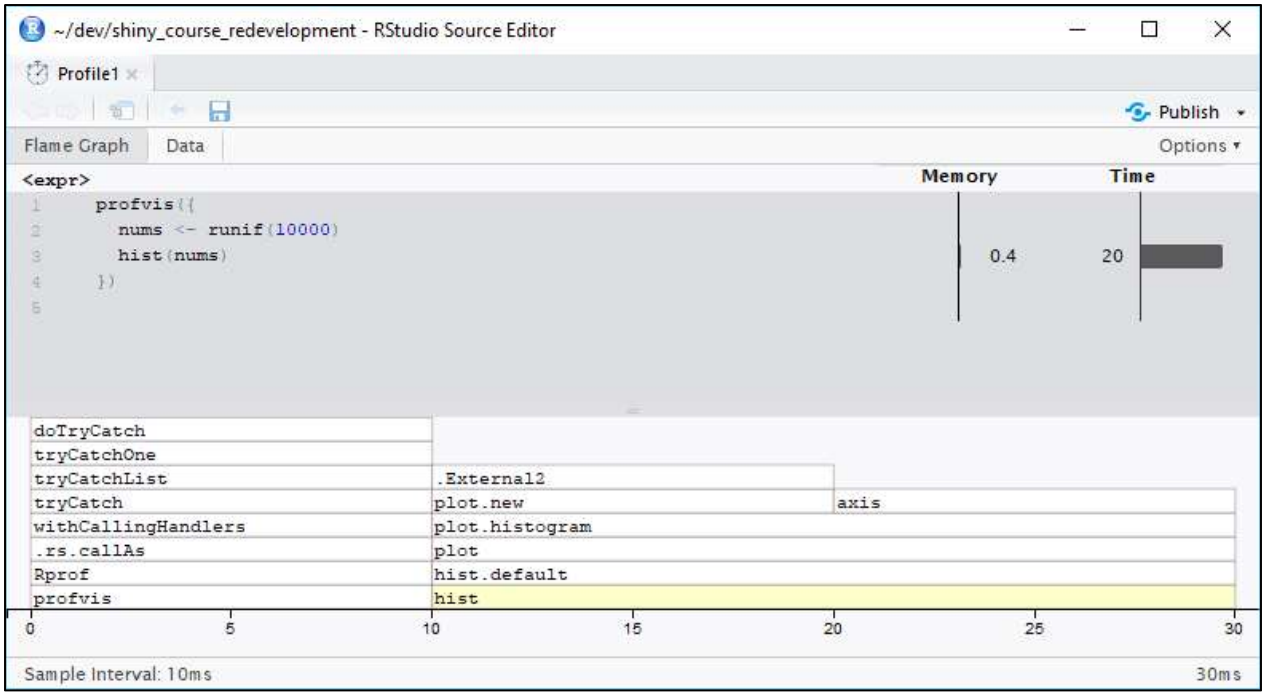
2.2 Running the Profiler

In RStudio profiling code is as simple as highlighting the code you want to profile and selecting from the menu Profile Selected Line(s).

We can also run the profiler by directly calling the `profvis` function, passing the code we want to evaluate as the first argument, e.g.

```
library(profvis)
profvis({
  nums <- runif(10000)
  hist(nums)
})
```

Once the code has finished profviz will open a code and “flame graph” visualisation that describes how much time was spent on each function call.



The top half of the chart shows bars next to each line of code representing how long each line took to complete. Here we can see that the `hist` function took 20ms to run.

The lower half of the chart shows horizontally how much time was spent on each function call, and shows vertically the callstack for each function.

The underlying data is produced by `Rprof`, which records the currently running function many times a second, but this is only a sample of what was running, so functions that complete very quickly may not even be captured.

We can see in the example above, runif ran so quickly it was not captured by the output. Usually this is not a problem since we are only looking for slow functions!

2.3 Profiling Shiny Apps

The **shiny** framework makes many function calls on our behalf to manage reactivity, making the call stack very noisy and hard to see what is going on. The `profvis` function now has functionality to turn off much of the shiny framework code so we can focus on what is specific to our app.

We can profile a shiny app by simply profiling the call to `runApp`.

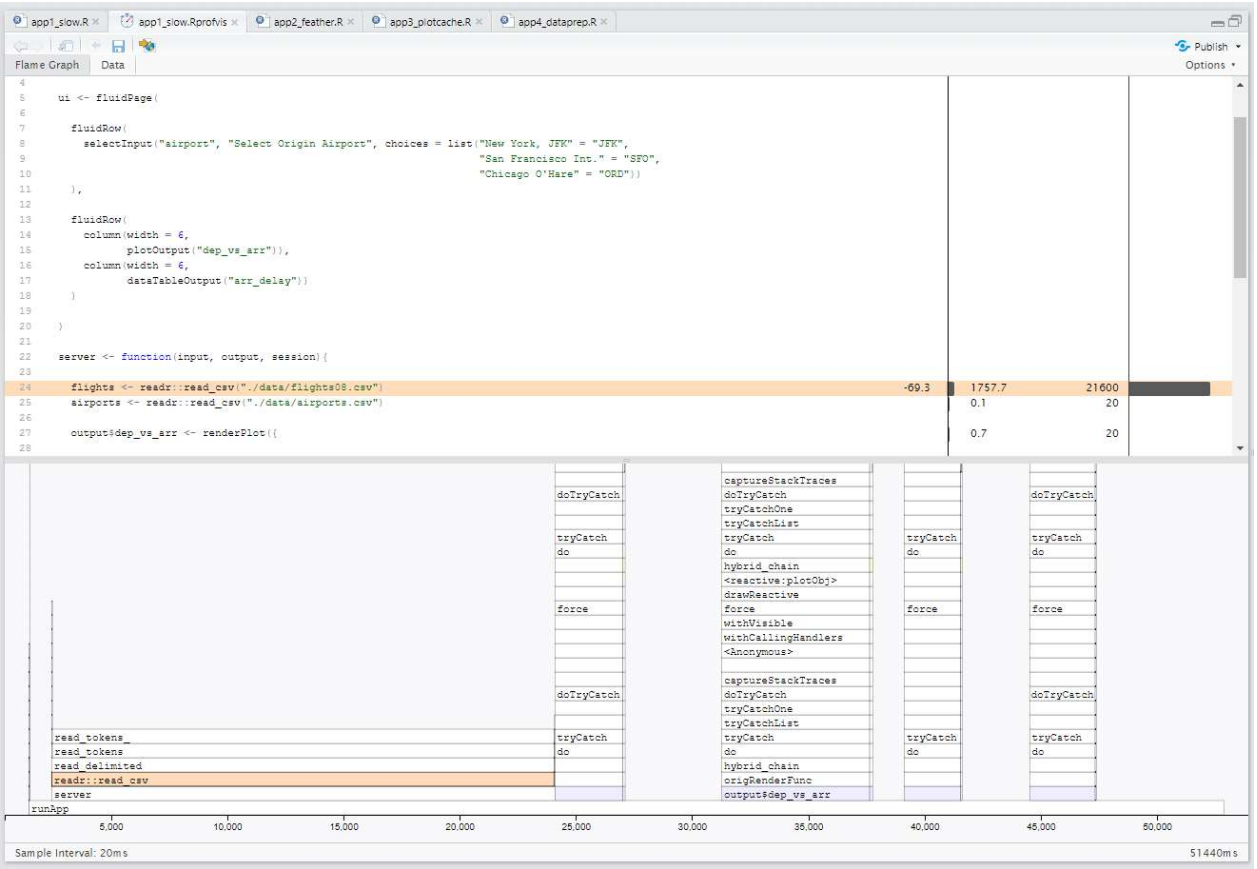
```
> profvis::profvis(runApp())
```

We can also start the profiler from the "Profile" menu and start our app using the "Run App" button in the interface, this is the same as running the above code.

The profiler will then start the **shiny** app and you can run as normal. When the app is closed the profiler will then generate the results and launch the interactive profile. Note, shiny outputs (`renderPlot`, etc.) are coloured blue in the flame graph.

As an example, the output below shows the profile from running our flights data application. In this example, we have loaded the application and then changed the airport choices a couple of times, before closing the application.

It is very easy to see here that the most time consuming element of the application is importing the data.



2.4 Data Import in Shiny Applications

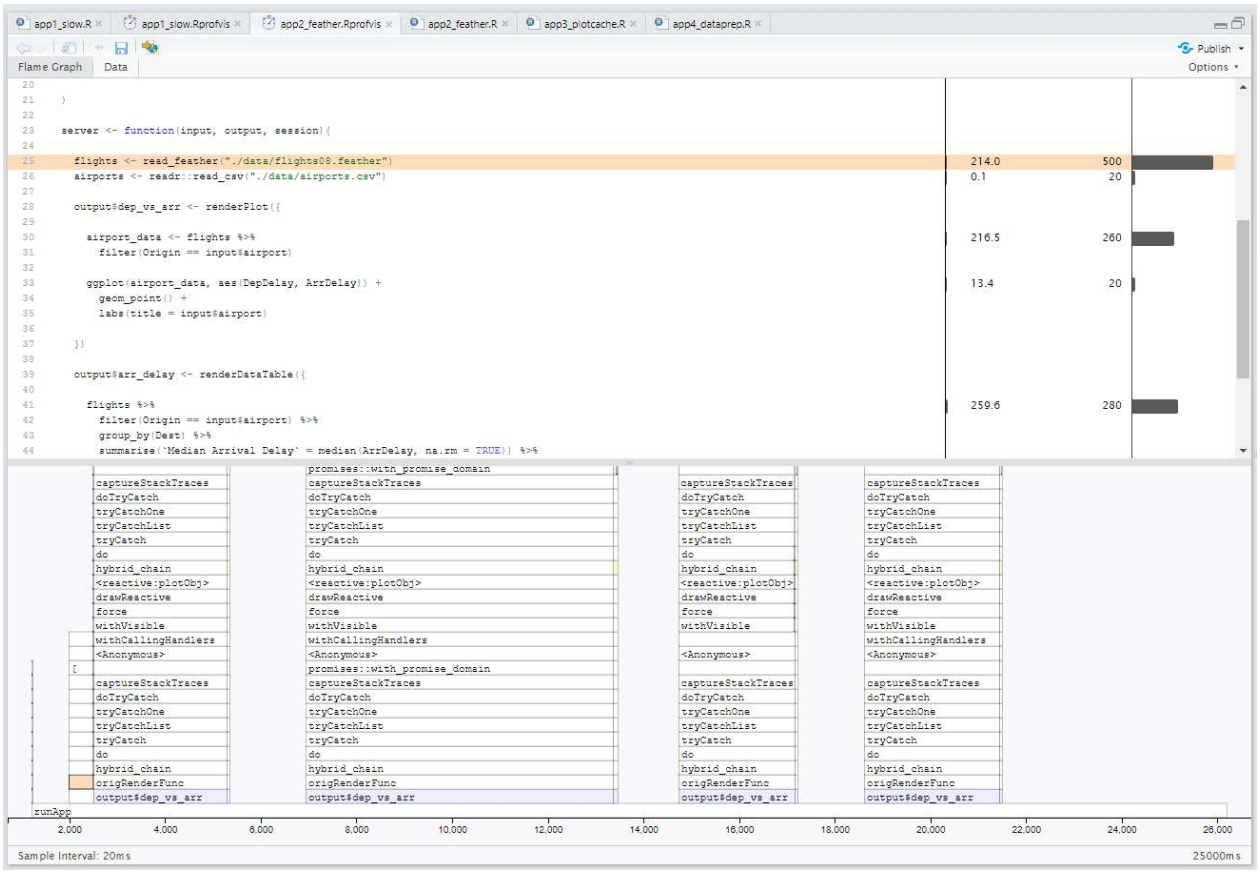
When it comes to shiny applications we will almost always be loading data in some form. If the data is large, this can be the most time-consuming element of our application. However, there are strategies that we can use to improve this.

- Change the file type to one that can be read faster, such as feather
- Only load the data that you need, if you have 100 columns but only use two only save two in your feather file
- Perform all of your manipulations and work with the raw data (if needed) outside of your shiny application (take a look at RStudio Connect for scheduling execution of RMarkdown documents).

In the example above, suppose that we had been able to pre-manipulate the data before we even got to running our application in a scheduled process. By saving only the 5 columns of the flights data that we will be working with and writing to the feather format (using `write_feather` from the **feather** package), we can change our data import to become:

```
flights <- read_feather("./data/flights08.feather")
airports <- readr::read_csv("./data/airports.csv")
```

Note that the airports data is sufficiently small that import takes no significant time, as can be seen in the profile above. Making only this change we can perform the profiling again to see the impact this has had:



As you can see from the orange highlighted row of code and in the flame graph at the bottom, this has significantly reduced the loading time, reducing to 500 from 21600. Whilst this may mean additional pre-processing, it significantly reduces time the end user is spent waiting.

1. Using the "Exercise" application, profile the application taking the following steps:
   a. Launch the application and allow to load
   b. Change the model coefficient to "DepTime"
   c. Change the number of samples to 17,000
   d. Change the model coefficient to "AirTime"
   e. Set the samples back to 10,000
   f. Set the coefficient back to "Distance"
2. Review the profile that is created. What is the slowest element of the application?
3. Make any changes that you think may improve performance and perform the profile again. Are there any further changes that you think may benefit the application?