Chapter 3
Load Testing

3.1 Overview

After initial profiling of our application, but before deploying to a production environment, the next step is to load test our application. Load testing is focused on seeing how our application performs when multiple users are accessing it at the same time. A typical requirement of an in production application. This can help us to determine:

- Additional changes to the code of the application itself to account for multiple users
- Changes to the set-up of our production environment (such as number of R processes available to the application)

3.2 Setting up

To perform the load testing we need to be able to run a tool called shinycannon. This tool is designed to mimic the effect of a given number of users running a common workflow in the application many times. This tool is not an R package, and installation and usage varies dependent upon the operating system.

Details of how to get set-up and run the tool on all common operating systems can be found on the shinyloadtest webpages:

https://rstudio.github.io/shinyloadtest/#shinycannon

For the remainder of this material, we will assume that you have already followed these instructions. As we expect that most people will be working with Windows machines, the examples throughout this material will replicate that of Windows users.

3.3 Recording an Example Session

So that we can replicate multiple user sessions we first of all need to record a typical session. To do this, we can either use an application that has already been deployed or an application running locally. It is this second option that we will follow here, for which we will need to open two R sessions at the same time.

3.3.1 Session 1: Launch the Application

From the first R session that you are using, launch the shiny application that you want to load test. You can do this in the usual way, either using the "Run App" button in the RStudio interface or running as code. Ensure that the application is running in a browser.

Leave this app running whilst recording and simulating users, taking note of the URL of the application.

3.3.2 Session 2: Record a Typical User Workflow

In the second R session you are running you can load the shinyloadtest package and set up the recording of a typical user workflow.

Suppose that the application that we started from our first R session is running at the URL http://127.0.0.1:4396/ .

```
> library(shinyloadtest)
> record_session("http://127.0.0.1:4396/")
```

If we were load testing a deployed application, we would replace the URL with that of the URL for the deployed app.

This will proceed to launch a new instance of the application. Run the application as you would expect a user to interact, changing options and viewing elements of the application.

For the example app that we have been considering, we changed the airport option to cycle through each of the airports, and return to others previously viewed (i.e. New York, Chicago, New York, San Francisco, Chicago).

When you have completed the workflow, close the browser and the recording will stop. This will then save a "recording.log" file.

This recording process does not need to be repeated for each change we make to an app, this log file defines the workflow and can be re-used for re-testing after making changes to the original application.

1. Using the "Exercise" application, record the following workflow:
   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"

3.4 Simulating Multiple Users

To simulate multiple users of our application, we now need to use the shinycannon tool, mentioned earlier. This is run at the command line, not as R code, so we need to either:

- Switch to the "Terminal" tab (in RStudio, requires RStudio >= 1.2)
- Open an independent command prompt

The tool requires two mandatory arguments and three optional, but recommended, arguments.

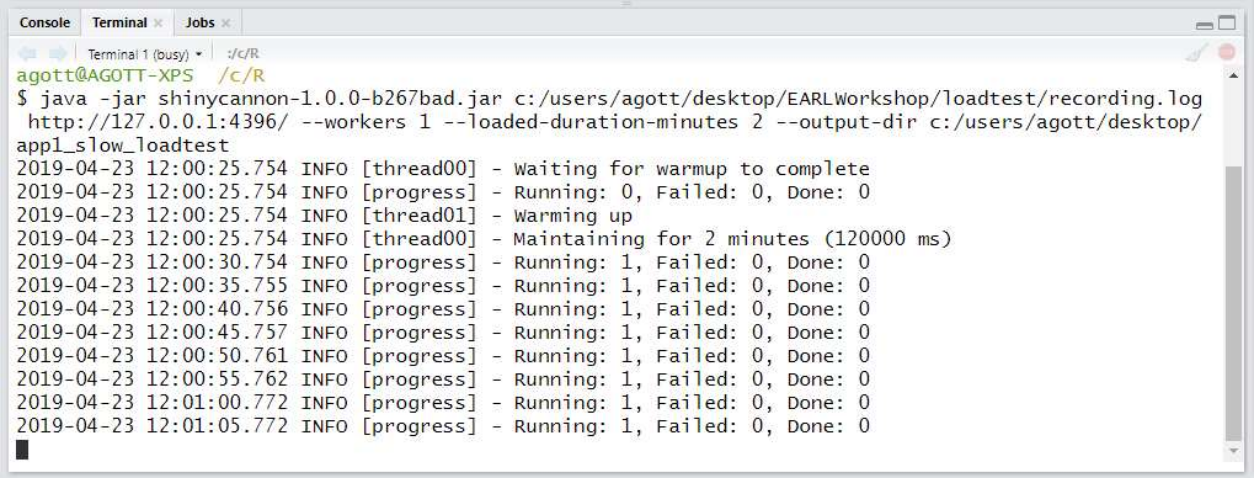
Argument	Use
[Recording Path]	The file path to the "recording.log" file, created during the recording process
[App URL]	The URL to the deployed or local application (as used in the recording step)
-workers	The number of simultaneous users of the application
-loaded-duration-minutes	How long the application should be loaded for and the sample workflow run. The workflow will be re-run during this time if it is longer than it takes for the workflow to run
-output-dir	The directory into which output files should be saved, this should be different for each test

Running on Windows, this would look something like the following:

```
java -jar shinycannon-1.0.0-b267bad.jar
c:/Users/agott/Desktop/recording.log http://127.0.0.1:4396/ --workers
1 --loaded-duration-minutes 2 --output-dir
c:/users/agott/Desktop/app1_slow_loadtest
```

When using the command line, we don't put commas between arguments and we don't specify the value that arguments take with "=", we simply use a space for both cases.

In this example, we are starting with just a single user to give us a baseline and running for 2 minutes, which should be longer than it takes to complete the workflow with the slowest version of the application.



This will create a series of files in the specified directory, that we can load into R and analyse.

1. Using the recording that you created in the last exercise (or the provided "recording.log" file), run a load test with the following options:
   a. Workers = 1
   b. Duration = 2 minutes
2. Try re-running the test, increasing the concurrent users to 5 (remember to change the output directory to a new location)

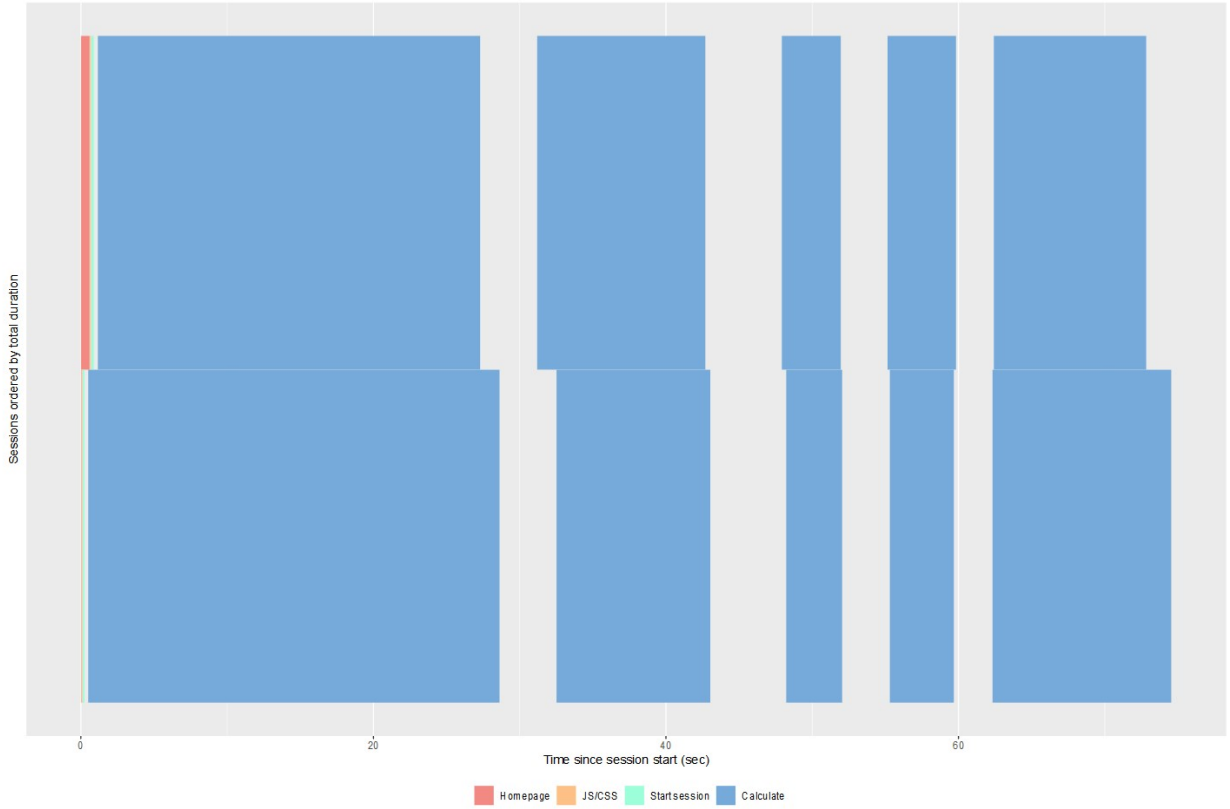
3.5 Reviewing Load Test Results

The final step in the load testing process is reviewing the data that has been produced. The shinyloadtest package comes with some further utility functions to not only read this data in, but also generate HTML reports that summarise the information that we need to review.

```
> runs <- load_runs("slow" = "./Desktop/app1_slow_loadtest")
> shinyloadtest_report(runs, "app1_slow.html")
```

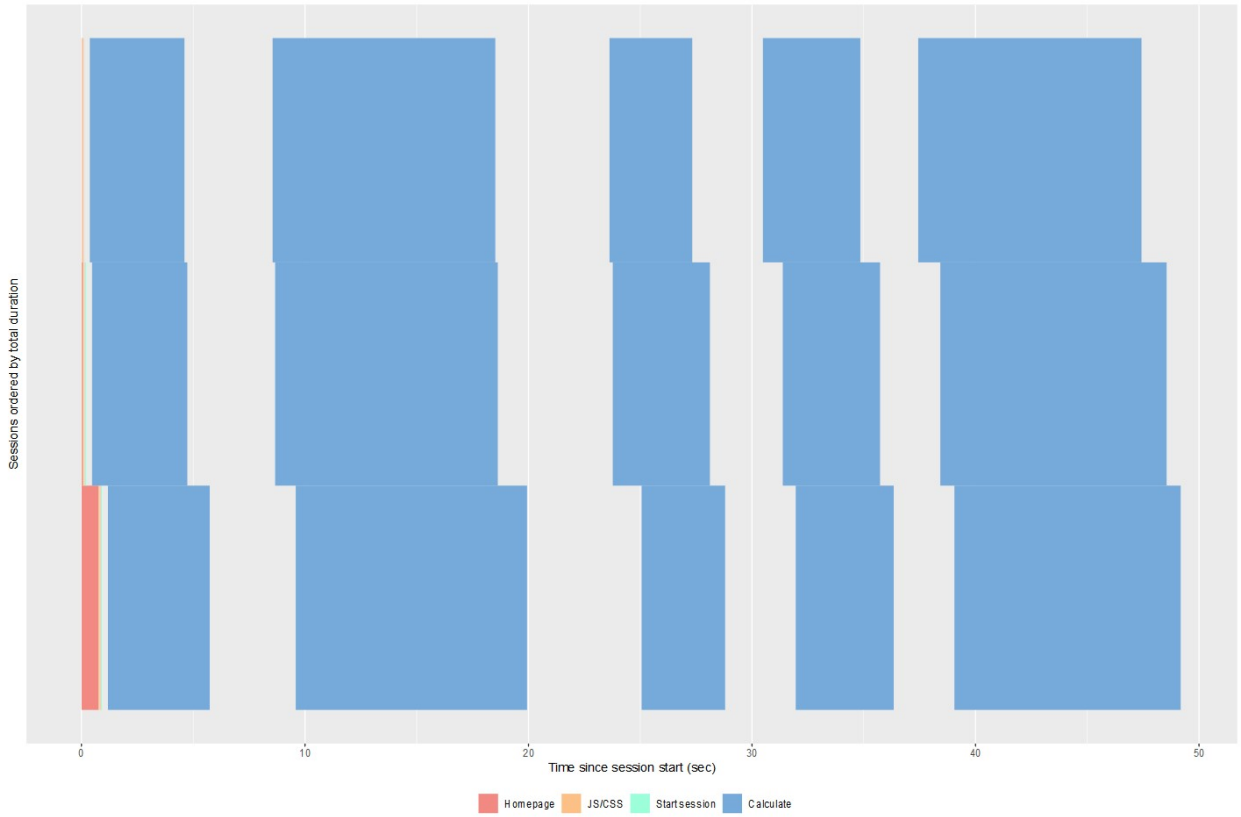
This generates a number of visualisations of the generated data that we can review. The first that is worth looking at more closely is the "Session Duration" tab, which shows all of the times that the workflow was completed.

In this example, we have tested the slowest version of our sample application (before we improved the file read time), and we can clearly see the large amount of time taken after the app initially loads for that to happen



Running the same load test after changing the file type (see the section on profiling for more details), you can see that in the same two minute period, more sessions were able to be completed and the initial load time has dropped from nearly 30 seconds to around 5.

But, what we can clearly see is that there is still a lot of time wasted when the graphics are generated. Here only one user is accessing the application, but it is clear that this will be problematic if we were to increase the number of users.



1. Using the output from one of your load tests from the previous exercise (or using the provided outputs), generate the load test report.
2. Which parts of the application are the slowest? Does this match your expectations from profiling the application?

3.6 Rendering Graphics in Shiny Applications

We previously looked at how we could improve the load time for data in an application, but what about graphics?

In the example that we are considering here, there are 3 possible graphics that can be created. Each time a user opens the application the graphics will remain the same (the data is not random or updating very frequently), but they are very slow to load due to the size of the data being rendered.

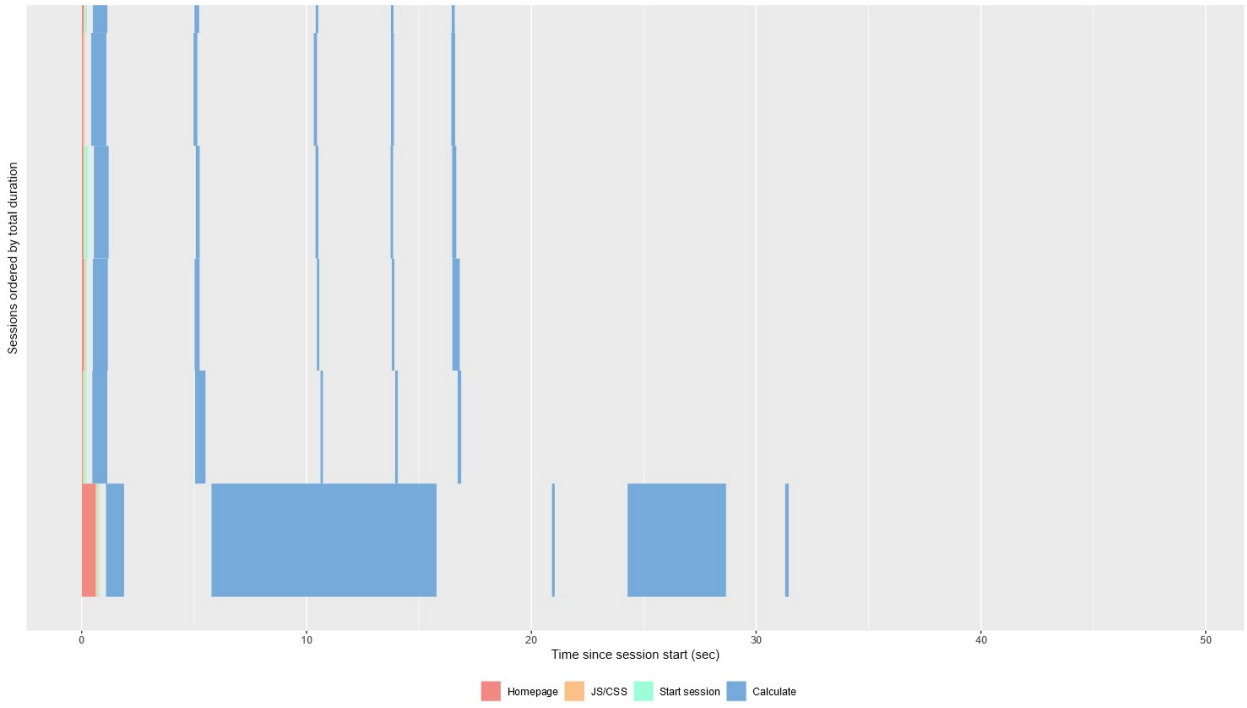
This makes this a good case for plot caching. Essentially this means saving a version of the graphic after it is created the first time, so that the next time the same graphic is required the cached version can be used instead of re-rendering the plot.

Making this change is, thankfully, very straightforward. The UI to our shiny app remains exactly as it was before, the only change comes in the server, where we now need to call `renderCachedPlot`.

```
output$dep_vs_arr <- renderCachedPlot({

    airport_data <- flights %>%
      filter(Origin == input$airport)

    ggplot(airport_data, aes(DepDelay, ArrDelay)) +
      geom_point() +
      labs(title = input$airport)

  }, cacheKeyExpr = input$airport)
```

The only other change is the argument `cacheKeyExpr`, which is the input that defines a unique graphic in the application. In this case it is the airport dropdown that makes a plot unique.

If we re-run the same load test as before (with one worker for two minutes), we can see the difference that this makes.



Whilst the first time the app generates each graphic still takes time, each subsequent run is significantly faster. We can start to see the real impact of this when we increase the number of workers.

3.7 Re-Running Load Tests

When we re-run a load test on an updated application we follow most of the same process as initially described, however we do not need to re-record the test. To compare across changes to the application it is best to use the same recording, as this will ensure the same options are pressed, in the same order, with the same gaps between.

However, we need to ensure that the application running in the first R session is the correct one. If we have made changes we need to ensure that this is the version of the application we are testing.

Otherwise we can run in the same way as already shown. Suppose we want to re-run our load test, but now for 5 workers (i.e. 5 concurrent users).

```
java -jar shinycannon-1.0.0-b267bad.jar
c:/Users/agott/Desktop/recording.log http://127.0.0.1:4396/ --workers
5 --loaded-duration-minutes 2 --output-dir
c:/users/agott/Desktop/app4_cache_5_loadtest
```

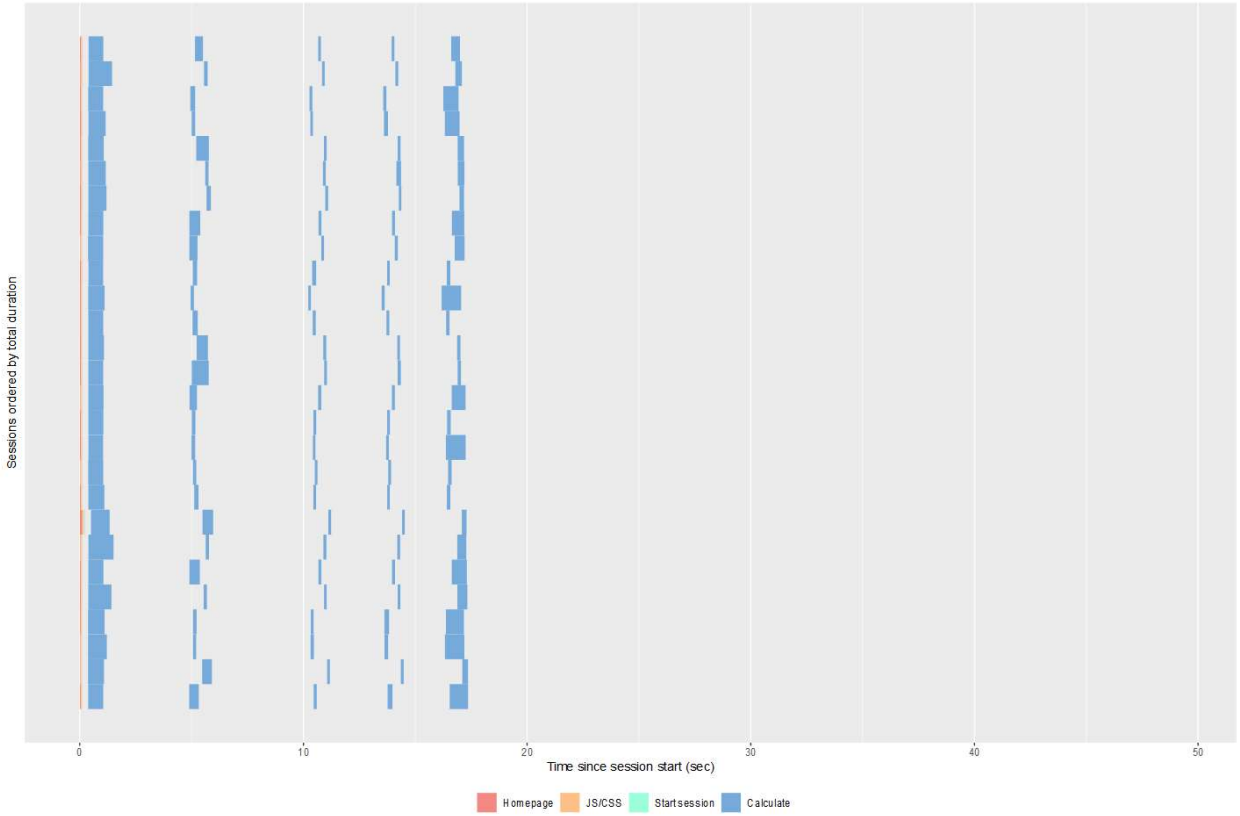
Running for multiple workers will now include a warm-up period, in which each worker is started at a given interval. Sessions are only recorded once that period is complete, so we will no longer see the long run time for the first instance.

3.7.1 Comparing Multiple Runs

Comparing multiple runs requires us to load multiple run outputs and generate a single report for all of the runs together.

```
> runs <- load_runs("feather" = "./Desktop/app2_feather_loadtest".
+                   "cache_5" = "./Desktop/app4_cache_5_loadtest")
> shinyloadtest_report(runs, "app4_cache.html")
```

When we compare runs in this way, it ensures that the scales on the analysis graphics are all the same, making it easy to compare how durations are changing.



1. The exercise application has two graphics in it. Which of these graphics would be ideal for plot caching and why?
2. Update the application to make use of plot caching for the relevant graphic
3. Re-run your previous load test and compare how the application performs with and without plot caching.