

# Using the `cachier` Package

Roger D. Peng <rpeng@jhsph.edu>

September 21, 2007

## 1 Introduction

The purpose of the `cachier` package is to provide tools for “caching” statistical analyses and for distributing these analyses to others in an efficient manner. Once distributed, a statistical analysis can be reproduced, modified, or improved upon. The `cachier` package is an implementation of the distributed reproducible research ideas described in Peng and Eckel (2007).

This tutorial will demonstrate how to use the `cachier` package for caching statistical analyses and distributed these analyses over the web. Specifically, we will demonstrate how to create and explore a cache database as well as download and explore cache databases created by others.

## 2 Caching Statistical Analyses

To illustrate some of the features of the `cachier` package we will use the following simple statistical analysis of the `airquality` dataset from the `datasets` package which comes with R. The code for the entire analysis is printed below.

```
library(datasets)
library(stats)

data(airquality)

fit <- lm(Ozone ~ Wind + Temp + Solar.R, data = airquality)
summary(fit)

## Plot some diagnostics
par(mfrow = c(2, 2))
plot(fit)

## Interesting non-linear relationship
temp <- airquality$Temp
ozone <- airquality$Ozone

par(mfrow = c(1, 1))
plot(temp, ozone)
```

The code is contained in a file called “sample.R” which comes with the `cachier` package. The above analysis is fairly simple and not very time-consuming so it is easily reproduced by anyone who can

run R, without any need for caching. Nevertheless, it is useful for demonstrating how the `catcher` package works.

The first step is to install the `catcher` package from the Comprehensive R Archive Network (CRAN) and load it into R.

```
> library(catcher)
> setConfig("verbose", TRUE)
```

For now, we also set the global `verbose` option to be `TRUE`, making `catcher` be somewhat more “chatty”.

The primary function is called `catcher` and it accepts a file name as its first argument. This file should contain the code for the analysis that you want to cache. Other arguments include the name of the cache directory (defaults to `.cache`) and the name of the log file (defaults to `NULL`). If `logfile = NULL` then messages will be printed to a file in the cache directory. Setting `logfile = NA` will send messages to the console.

The “sample.R” file containing the above analysis comes with the `catcher` package and can be copied into your working directory. Given a file containing the code of an analysis, you can call the `catcher` function as

```
> catcher("sample.R")
```

Call:

```
lm(formula = Ozone ~ Wind + Temp + Solar.R, data = airquality)
```

Residuals:

	Min	1Q	Median	3Q	Max
	-40.485	-14.219	-3.551	10.097	95.619

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	-64.34208	23.05472	-2.791	0.00623
Wind	-3.33359	0.65441	-5.094	1.52e-06
Temp	1.65209	0.25353	6.516	2.42e-09
Solar.R	0.05982	0.02319	2.580	0.01124

Residual standard error: 21.18 on 107 degrees of freedom

(42 observations deleted due to missingness)

Multiple R-Squared: 0.6059, Adjusted R-squared: 0.5948

F-statistic: 54.83 on 3 and 107 DF, p-value: < 2.2e-16

The `catcher` function evaluates each expression in the file and prints any resulting output to the console. For example, the summary of the fitted linear model is printed to the console while the two plots are sent to the appropriate graphics device. The log messages are written to a file in `.cache/log/sample.R.log` which contains information about each expression evaluated by `catcher`. We will discuss the contents of the log file in Section 2.2.

When `catcher` evaluates each code expression, the results of the evaluation are cached to the database and lazy-loaded back into the workspace. After running the “sample.R” analysis, we can see that there are the following objects now in the workspace:

```
> ls()
```

```
[1] "airquality" "fit"          "ozone"      "temp"
```

Since the objects are lazy-loaded, they do not occupy any memory until they are accessed. The lazy-loading is not as important on the first evaluation but can reduce the amount of evaluation time required on subsequent analysis

For example, take the following very simple set of expressions.

```
x <- rnorm(1000000)
s <- summary(x)
print(s)
```

The first expression creates a vector of 1 million standard Normal random variates and the second computes a summary (a five-number summary plus the mean). The amount of time to evaluate these expressions the first time is

```
> systime <- system.time(cacher("bigvector.R"))

      Min.   1st Qu.   Median     Mean   3rd Qu.     Max.
-4.790000 -0.678500 -0.003163 -0.003113  0.672900  4.794000

> print(systime)

      user  system elapsed
 2.160   0.108   2.271
```

In this case, the evaluation time is about 2.2 seconds. After running `catcher`, the objects `x` and `s` reside in the workspace and have been cached to the database. Subsequent evaluations of the same code should take much less time since we can simply load `x` and `s` from the cache.

```
> rm(x, s)
> systime <- system.time(cacher("bigvector.R"))

      Min.   1st Qu.   Median     Mean   3rd Qu.     Max.
-4.790000 -0.678500 -0.003163 -0.003113  0.672900  4.794000

> print(systime)

      user  system elapsed
 0.036   0.000   0.036
```

Now the evaluation only takes about 0.036 seconds. In fact, the analysis here is particularly quick because we do not need the `x` vector at all. We can simply print the summary object `s`.

Even if we did need to access the vector `x`, loading from the cache is often faster than regenerating all of the random Normals using `rnorm`. For example, if we wanted to calculate the 95th percentile of the data, then we could simply do

```
> systime <- system.time(q95 <- quantile(x, 0.95))
> print(q95)

      95%
 1.642029

> print(systime)

      user  system elapsed
 0.488   0.024   0.513
```

## 2.1 Exploring a cached analysis

Once an analysis has been cached using `cache`, it can be explored using the utilities provided in the `cache` package. Since you can cache analyses from multiple files (as we have done above), we can show which analyses have already been cached using the `showfiles` function.

```
> showfiles()

[1] "sample.R"      "bigvector.R"
```

Here we see the file names corresponding to the two files that we analyzed in the previous section. If you want to examine a particular analysis, you can use the `sourcefile` function to choose that analysis and `showcode` will simply display the raw source file.

```
> sourcefile("bigvector.R")
> showcode()

x <- rnorm(1000000)
s <- summary(x)
print(s)
```

You can also use the `code` function to display the code in a summary form

```
> sourcefile("sample.R")
> code()

source file: sample.R
1  library(datasets)
2  library(stats)
3  data(airquality)
4  fit <- lm(Ozone ~ Wind + Temp +
5  summary(fit)
6  par(mfrow = c(2, 2))
7  plot(fit)
8  temp <- airquality$Temp
9  ozone <- airquality$Ozone
10 par(mfrow = c(1, 1))
11 plot(temp, ozone)
```

The `code` function truncates expressions to a single line and also shows the sequence number assigned to each expression in the order that the expression is encountered in the source file. In order to see the full code for each expression, you can set the `full = TRUE` option to `code`.

The first thing you might do when exploring a cached analysis is to explore the elements of the cache database itself. You can list the objects available using the `showobjects` function, which returns a character vector of the names of each object in the database. Passing an expression sequence number to `showobjects` via the `num` argument shows the objects created by that expression.

```
> showobjects()

[1] "airquality" "fit"          "temp"         "ozone"

> showobjects(8)

[1] "temp"
```

```
> showobjects(1)
```

```
character(0)
```

These objects can be lazy-loaded into the workspace using the `loadcache` function.

```
> loadcache()
```

```
> ls()
```

```
[1] "airquality" "fit"          "ozone"        "temp"
```

Now, we can print the linear model fit (without actually fitting the model) by calling

```
> print(fit)
```

```
Call:
```

```
lm(formula = Ozone ~ Wind + Temp + Solar.R, data = airquality)
```

```
Coefficients:
```

```
(Intercept)      Wind      Temp      Solar.R  
-64.34208    -3.33359    1.65209    0.05982
```

The `loadcache` function takes a `num` argument which can be a vector of indices indicating code expression sequence numbers. For example, if you want to load only the objects associated with expression 4 (i.e. the `fit` object), then you can call `loadcache(4)`.

In addition to exploring the objects in the cache database, you may wish to run the analysis on your own computer for the purposes of reproducing the original results. You can run individual expressions or a sequence of expressions with the `runcode` function. The `runcode` function accepts a number or a sequence of numbers indicating expressions in an analysis. For example, in order to run the first four expressions in the “sample.R” analysis, we could call

```
> rm(list = ls())
```

```
> code(1:4)
```

```
source file: sample.R
```

```
1 library(datasets)
```

```
2 library(stats)
```

```
3 data(airquality)
```

```
4 fit <- lm(Ozone ~ Wind + Temp +
```

```
> runcode(1:4)
```

```
evaluating expression 1
```

```
evaluating expression 2
```

```
loading cache for expression 3
```

```
loading cache for expression 4
```

```
> ls()
```

```
[1] "airquality" "fit"
```

In this case, expressions 1 and 2 are evaluated but expressions 3 and 4 are loaded from the cache. By default, `runcode` does not evaluate expressions for which it can load the results from the cache. In order to force evaluation of all expressions, you need to set the option `forceAll = TRUE`.

## 2.2 Understanding the log file

As each expression is being evaluated, `catcher` keeps track of which expressions result in the creation of new objects (including modification of existing objects) and which expressions have side effects. Expressions with side effects cannot be cached and therefore must always be evaluated. The primary operation falling into this category is plotting, which launches a graphics device and makes changes to that device. One exception is `lattice` plots which can be stored as objects and therefore cached. The log file contains information about each expression and whether it needs to force evaluation. Here we print the first few lines of the log file for this analysis.

```
1: library(datasets)
   eval expr and cache
   expression has side effect: 6e2f63b75ebfb78f5447d5c3198fa200
2: library(stats)
   eval expr and cache
   expression has side effect: d1f85d2a81165655d9fb907c50f05bb1
3: data(airquality)
   eval expr and cache
   -- loading expr from cache
4: fit <- lm(Ozone ~ Wind + Temp +
   eval expr and cache
   -- loading expr from cache
```

Understanding the log file output is not critical to using `catcher` but it is occasionally useful to know what the function is doing for a given expression. Each expression is assigned a number based on when it is encountered in the source file and a snippet of the expression is printed immediately after the number. Below, `catcher` will indicate if the expression needs to be evaluated and cached and will try to determine if the expression resulted in a side effect. The check for side effects is rudimentary and will not catch all cases. Once the expression has been cached, `catcher` will reload the results from the cache into the global environment (i.e. workspace) and move to the next expression.

Running the analysis a second time with `catcher` results in the following log file being generated.

```
1: library(datasets)
   force expression evaluation
2: library(stats)
   force expression evaluation
3: data(airquality)
   -- loading expr from cache
4: fit <- lm(Ozone ~ Wind + Temp +
   -- loading expr from cache
```

Here we see that expressions 1 and 2 were forced to be evaluated because the `library` function results in a side effect (i.e. altering the search list). Expressions 3 and 4 create objects in the workspace so they can be lazy-loaded from the cache. Note here that although the `airquality` dataset is loaded from the cache, it is not needed if you are primarily interested in examining the `fit` object from the `lm` call. This is where lazy-loading is very useful. However, if you want to fit a different model, say, with some interactions, then of course the original data will be loaded into the workspace the first time it is accessed.

## 2.3 Caching Individual Expressions

Individual expressions can be cached using the `cc` function. For example, you might want to cache just a single critical section of an analysis rather than the entire analysis itself. In this case, you can wrap the expression with the `cc` function and subsequent evaluations of the expression will be loaded from the cache (assuming the expression hasn't changed in the meantime).

For example, the following artificial expression takes at least 2 seconds to evaluate before the value of 5 is assigned to `x`.

```
> cc({
+   x <- local({
+     Sys.sleep(2)
+     5
+   })
+ })
```

Caching the expression with `cc` allows the value of `x` to be loaded from the cache on subsequent evaluations. The cache database corresponding to this expression can be explored as in the previous examples.

```
> showfiles()

[1] "5e9a701cf987e92e132b68ae9e892453"
```

Note that since there is no source file to use, a file name is generated based on a digest of the expression being cached.

```
> sourcefile(showfiles())
> showcode()

{
  x <- local({
    Sys.sleep(2)
    5
  })
}
```

It is probably not so useful to explore the cache associated with a single expression. More likely, you will want to cache an expression so that subsequent evaluations of the same expression will be loaded from the cache and therefore not take as long.

## 3 Distributing Cached Analyses Over the Web

There are two options for user who wishes to distributed a cached statistical analysis over the web. If the user has access to a local webserver, then the cache directory can be posted on that webserver and be downloaded by others using the `clonecache` function. The other option, if a local webserver is not available, is to make use of the Reproducible Research Archive at <http://penguin.biostat.jhsph.edu/> which stores reproducibility packages and makes them available to others.

### 3.1 Posting a cache directory

If you have access to a webserver you can post your cache directory directly on the webserver for others to access. Once made available on a webserver, others can access your cache directory by using the `clonocache` function in the `clonercache` package and the URL of the directory on your webserver. For example, we can download the analysis corresponding to the “bigvector.R” file by calling

```
> clonocache("http://www.biostat.jhsph.edu/rr/bigvector.cache")
```

```
downloading source file list
downloading metadata
downloading source files
downloading cache database file list
```

This call to `clonocache` downloads all of the relevant cache files related to the analysis except for the cache database files. In order to download the cache database files, the option `all.files = TRUE` must be set.

Once a cache package has been downloaded using `clonocache` you can use all of the tools described in the previous sections to explore the cache and the run some of the analyses.

```
> showfiles()
```

```
[1] "bigvector.R"
```

```
> sourcefile("bigvector.R")
```

```
> code()
```

```
source file: bigvector.R
1 x <- rnorm(1000000)
2 s <- summary(x)
3 print(s)
```

```
> showobjects()
```

```
[1] "x" "s"
```

```
> loadcache()
```

```
> print(s)
```

```
/ transferring cache db file 44060b00bf1360bc8a5499cf14c7fb11
      Min.   1st Qu.   Median     Mean   3rd Qu.    Max.
-4.661000 -0.675000 -0.002147 -0.001026  0.675100  5.008000
```

By default, `clonocache` does not download the cache database files until they are needed in order to minimize the amount of data that is transferred. Cache database files are only transferred from the remote host when the objects associated with them are first accessed.

In the above example, the database file corresponding to the object `s` is only transferred when we call `print(s)`. When a database object has to be downloaded from the remote site, a message will be printed to the screen indicating the transfer.

## 3.2 Using the Reproducible Research Archive

In order to use the Archive, one must first create a “cache package” using the `package` function. The `package` function simply zips up the cache directory with the `zip` utility (which must be available) and creates a zip archive whose filename is based on the MD5 digest of the zip file itself. This MD5 digest is used to identify the package once it is posted on the Archive.

Other users can download the cache package by simply passing the package ID (i.e. the MD5 digest) to the `clonecache` function via the `id` argument. For example, in order to download the statistical analyses conducted in Section 2, you can call

```
> clonecache(id = "ccf4cb9a8995afcb74e39b1caa80d77e")
```

```
downloading source file list
downloading metadata
downloading source files
downloading cache database file list
downloading metadata
downloading source files
downloading cache database file list
```

which will download all of the cache files except the cache database files.

The `cacher` package sets the Archive at <http://penguin.biostat.jhsph.edu/> as the default repository from which to download cache packages. If another archive is to be used, the repository can be changed using the `setConfig` function to point the “archive” option to the appropriate URL.

This particular cache package comes with both materials for both analyses conducted in Section 2 and hence there are two source files.

```
> showfiles()

[1] "sample.R"      "bigvector.R"

> sourcefile("sample.R")
> code()

source file: sample.R
1  library(datasets)
2  library(stats)
3  data(airquality)
4  fit <- lm(Ozone ~ Wind + Temp +
5  summary(fit)
6  par(mfrow = c(2, 2))
7  plot(fit)
8  temp <- airquality$Temp
9  ozone <- airquality$Ozone
10 par(mfrow = c(1, 1))
11 plot(temp, ozone)

> showobjects()

[1] "airquality" "fit"          "temp"         "ozone"

> loadcache()
> print(fit)
```

```
/ transferring cache db file 4a2278092572524d879cb54ac5a5a875
```

```
Call:
```

```
lm(formula = Ozone ~ Wind + Temp + Solar.R, data = airquality)
```

```
Coefficients:
```

```
(Intercept)      Wind      Temp      Solar.R  
-64.34208     -3.33359     1.65209     0.05982
```

Again, in this case, since the `fit` object has not been previously downloaded, it is downloaded on demand when we try to print it.

## 4 Verifying a Cached Analysis

Once you have cloned an analysis conducted by someone else, you may wish to verify that the computation that you run on your computer leads to the same results that the original author obtained on his/her computer. This can be done with the `checkcode` function. The `checkcode` function essentially evaluates each expression locally (if it can) and compares the output with the corresponding value stored in the cache database.

If the locally created object and the cached object are the same, then that expression is considered verified. If an expression does not create any objects, then there is obviously nothing to compare. If the locally created object and the cached object are different, the verification fails and `checkcode` will indicate which object (if there was more than one) it could not verify.

For example, we can run the `checkcode` function on the analysis of the `airquality` dataset from before. Here we will only check the first four code expressions.

```
> sourcefile("sample.R")  
> showobjects(1:4)  
  
[1] "airquality" "fit"  
  
> checkcode(1:4)  
  
checking expression 1  
= no objects to check, OK  
checking expression 2  
= no objects to check, OK  
checking expression 3  
/ transferring cache db file a05108daaa93248d2ddc7608f92a64c7  
+ object airquality OK  
checking expression 4  
+ object fit OK
```

In the first four expressions, there are two objects created: the dataset `airquality` and the linear model object `fit`. The `checkcode` function compares each of those objects with the version stored in the cache database (which we previously cloned from the web). In this case, the objects match and the computations are verified. Notice that in expression 3, the database file for the `airquality` object had to be downloaded so that it could be checked against the locally created version.

We can check the code in the “bigvector.R” analysis also. In this analysis there are two objects that need to be verified: `x`, the vector of standard Normals and `s` the “summary” object.

```

> sourcefile("bigvector.R")
> checkcode()

checking expression 1
/ transferring cache db file 7daf09148498b5f85ed5603efcf3794a
- object x not verified, FAILED
checking expression 2
/ transferring cache db file bcf7ca8479925f3b6b0283c8a073caa1
- object s not verified, FAILED
checking expression 3
= no objects to check, OK

```

Notice that expressions 1 and 2 failed for a common reason (expression 3 had no objects to verify). Since the analysis did not set the random number generator seed in the beginning, the generation of the Normal random variates on the local machine is not the same as that for the original analysis. Therefore, the object `x` is not reproducible (nor is `s`).

Of course, there are limitations to verifying statistical analyses. Analyses may take a long time to run and therefore it may take a long time to verify a given computation. If one does not have the necessary external resources (i.e. hardware, software) then it may not be possible to verify an analysis at all. Currently, verification of analyses is limited to R objects only. We cannot verify the output of summary or print functions nor can we verify plots (although lattice plots can be verified if they are stored as R objects).

Certain analyses may load external datasets or inputs which will generally not be available to the other users. A typical analysis might be of the form

```

data <- read.csv("faithful.csv")
with(data, plot(waiting, eruptions))

library(splines)
fit <- lm(eruptions ~ ns(waiting, 4), data = data)

xpts <- with(data, seq(min(waiting), max(waiting), len = 100))
lines(xpts, predict(fit, data.frame(waiting = xpts)))

```

This analysis reads in the the “Old Faithful” dataset which contains eruption times and waiting periods for the Old Faithful geyser in Yellowstone National Park. Although this dataset is available from the R installation, we have exported it here to a comma-separated-value file for demonstration.

The original author of this analysis can run the `cachef` function on this analysis file and distributed it to others.

```

> cachef("faithful.R")

```

However, another user (presumably on a different computer) will not be able to verify all of the code in this analysis

```

> sourcefile("faithful.R")
> checkcode()

checking expression 1
- problem evaluating expression, FAILED
- simpleWarning: 'cannot open file 'faithful.csv', reason 'No such file or directory''
- loading objects from cache

```

```
/ transferring cache db file 9e0c8598b70949b92808ad05b173a761
checking expression 2
= no objects to check, OK
checking expression 3
= no objects to check, OK
checking expression 4
/ transferring cache db file bf2586a466547cf574bfbef114403daa
+ object fit OK
checking expression 5
/ transferring cache db file dd6cbd37575c4e23b151e9f301da5002
+ object xpts OK
checking expression 6
= no objects to check, OK
```

Here, the first expression, which reads the dataset in via `read.csv` cannot be verified because the “faithful.csv” file is not available. However, the other expressions can be run on the local machine and are verifiable since they can use the cached copy of the dataset.

## References

Peng, R. D. and Eckel, S. P. (2007), “Distributed reproducible research using cached computations,” Tech. Rep. 147, Johns Hopkins University Department of Biostatistics, <http://www.bepress.com/jhubiostat/paper147/>.

## A Session Information

- R version 2.5.1 (2007-06-27), x86\_64-unknown-linux-gnu
- Locale: LC\_CTYPE=en\_US.UTF-8; LC\_NUMERIC=C; LC\_TIME=en\_US.UTF-8; LC\_COLLATE=en\_US.UTF-8; I
- Base packages: base, datasets, graphics, grDevices, methods, splines, stats, utils
- Other packages: cacher 0.1, cacheSweave 0.5, lattice 0.15-11, RColorBrewer 1.0-1