Stored Object Caches for **R**

Bill Venables, CSIRO Mathematics, Informatics and Statistics
Brisbane, Australia

April, 2010

Contents

1 Introduction

When an object is created in **R**, by default it remains in memory. If the objects are large, or numerous, memory management can become an issue even on a modern computer with large central memory. At the end of a session the objects in the global environment are usually kept in a single binary file in the working directory called `.RData`. When a new **R** session begins with this as the initial working directory, the objects are loaded back into memory. If the `.RData` file is large startup may be slow, and the memory can be under pressure again right from the start of the session.

Objects need not be always held in memory. The function `save` may be used to save objects on the disc in a file, typically with an `.RData` extension. The objects may then be removed from memory and later recalled explicitly with the `load` function. The `SOAR`[1] package provides simple way to store objects on the disc, but in such a way that they remain visible on the search path as *promises*, that is, if and when an object is needed again it is automatically loaded into memory. It uses the same *lazy loading* mechanism as packages, but the functionality provided here is more dynamic and flexible.

The `SOAR` package is based on an earlier package of David Brahm called `g.data`. This earlier package was briefly described in *R News*. (See [1]).

2 Local stored object caches

The *working directory* for any **R** session is the directory where by default **R** will look for data sets or deposit text or graphical output. It can be found from within the session using the function `getwd()`. By a *local stored object cache* we mean a directory, usually a sub-directory of the working directory, which will be used by **R** to contain saved object `.RData` files. Users of **S-PLUS**, (a programming environment not unlike **R**), will be familiar with the `.Data` sub-directory of the working directory which acts as a local stored object cache in precisely this sense. These caches are created and used by **R** itself, not the user directly.

[1] `SOAR` is an acronym for the four main functions of the package, `Store`, `Objects`, `Attach`, and `Remove`

To specify the way a cache works it is helpful to give an example.

```
> ## create some dummy data
> X <- matrix(rnorm(1000*50), 1000, 50)
> S <- var(X)
> Xb <- colMeans(X)
> Store(X)
```

At this point, if the initial workspace is empty, the global environment contains the two objects `S` and `Xb`, and the matrix is stored on the disc in a local stored object cache.

```
> objects()   ## or ls()

[1] "S"  "Xb"

> find("X")

[1] ".R_Cache"
```

Several things have happened.

1. A special sub-directory, called `.R_Cache`, of the working directory has been created, if necessary,

2. The object `X` has been saved in the cache as an `.RData` file, (with an encoded name, see later), and *removed* from the global environment,

3. An image of the cache has been placed at position 2 of the search path, also under the name "`.R_Cache`",

4. An object called "`X`" has been placed in position 2 of the search path which is really a promise to load the real `X` into position 2, as needed.

The `SOAR` package provides a slightly enhanced version of the `search` function, for inspecting the search path. As well as the entries, it shows the enclosing directories, where applicable. Such an enclosing directory is called the `lib.loc` for packages and similar entries. We now continue the example:

```
> Store(Xb, S)
> Search()

   name              lib.loc
01 .GlobalEnv
02 .R_Cache          .
03 package:stats     R_HOME/library
04 package:graphics  R_HOME/library
05 package:grDevices R_HOME/library
06 package:utils     R_HOME/library
07 package:datasets  R_HOME/library
08 .R_Data           ~
09 .R_Utils          ~
10 package:SOAR      C:/DOCUME~1/ven037/LOCALS~1/Temp/Rinst509676455
11 package:methods   R_HOME/library
12 Autoloads
13 package:base      R_HOME/library

> Objects()

[1] "S"  "X"  "Xb"
```

The `SOAR` function `Objects` is like the standard function `objects` (or equivalently `ls`), but applies only to stored object caches. In addition to listing objects, it locates the cache on the search path beforehand using both its name and `lib.loc`. As the position of the cache on the search path may wander as packages are added, and as there may be several caches on the search path under the same name, this is a useful feature.

2.1 Memory issues

An important (but not the only) reason to use saved object caches is to release memory as much as possible. It is useful to keep track of memory usage as an object is stored and retrieved by various operations in **R**, which we now do with another small, artificial example.

Memory usage is largely tracked by looking at the number of “`Vcells`” currently in use. This is an element of the matrix which forms the output message of the garbage collector:

```
> vc <- gc(); vc ; v0 <- vc["Vcells", "used"]
```

```
         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 140652  3.8     350000  9.4   350000  9.4
Vcells  92145  0.8     786432  6.0   381261  3.0
```

At this point there are 92145 `Vcells` in use. Now create a large **R** object and manipulate it, keeping track of what happens to the `Vcells` in use, as a difference from this base level. We begin by defining a tiny function to pick out the relevant number from the `gc` output.

```
> Vcells <- function()
+   c(Vcells = gc()["Vcells", "used"])             ; Vcells()-v0
```

```
Vcells
    81
```

```
> bigX <- matrix(rnorm(1000^2), 1000, 1000)        ; Vcells()-v0
```

```
 Vcells
1000075
```

```
> Store(bigX)                                      ; Vcells()-v0
```

```
Vcells
   103
```

```
> d <- dim(bigX)                                   ; Vcells()-v0
```

```
 Vcells
1000140
```

```
> bigX[1,1] <- 0                                   ; Vcells()-v0

 Vcells
2000138

> Attach()                                         ; Vcells()-v0

 Vcells
1000160

> Store(bigX)                                      ; Vcells()-v0

Vcells
   166
```

It is worth looking at this in some detail.

- Creating `bigX` in the global environment increases memory usage by about 10^6 cells, as expected.
- Storing the object reduces the cells in use to around the base level again.
- Finding the dimension of `bigX` requires it to be loaded into the cache, honoring the promise to do so. Cell usage increases by about 10^6 cells again.
- Modifying `bigX`, in this case replacing its first entry by zero, causes a *second copy* of the object to be made in the global environment. Cell usage increases by a further 10^6 cells.
- Using the `SOAR` function `Attach` re-establishes the cache on the search path. The object `bigX` at position 2 of the search again becomes a promise to re-load rather than the object itself, resulting in a *reduction* in memory usage of about 10^6 cells.
- Finally, storing the modified `bigX` takes it out of memory and places the new version in the cache. The promise is now to re-load the modified version and the original version is lost.

We can clear the cache on the disc using:

```
> Remove(bigX)                                    ; (Vcells()-v0)
```

```
Vcells
   115
```

The effect on memory is insignificant as only the promise object has been removed. The main effect is to remove an `.RData` file from the disc.

2.2 Specifying objects for storage or removal

The functions `Store` and `Remove` take as their main arguments a specification of objects to be stored or removed respectively. There are four ways to do this, namely:

1. As *unquoted names* of objects, such as `X`, `bigX` and the like.
2. As *explicit character strings* using any of the three quotation styles allowed in **R**, namely single, double or backtick quotes,
3. As *an expression evaluating to a character string vector*, which gives the names of objects to be removed, e.g. `objects(pattern = "^X")`, which would specify all objects whose name started with `X`.
4. As a *character string vector* value for the argument `list`.

Hence

```
> Store(objects())
```

when issued at the command line would store all objects in the global environment whose names did not begin with a period. This is a common idiom. An almost equivalent way to do this would be

```
> objs <- objects()
> Store(list = objs)
```

In this case, however, the object objs would remain in the global environment. Specification styles can be mixed, so a fully equivalent way would be

```
> objs <- ls()
> Store(objs, list = objs)
```

Also

```
> Remove(Objects())
```

would remove from the cache all objects whose names did not begin with a period. This is seldom necessary but holds a certain appeal for some obsessively tidy minds.

2.3 lib.loc and lib

All four functions, Store, Objects, Attach and Remove need to know where the cache is located on the disc. This is done in two parts, similar to the way that package locations are specified, namely by giving the *enclosing directory*, known as the lib.loc and the *cache name*, or lib.

For local stored object caches, the default lib is usually .R_Cache. More precisely, the default is either the value of the environment variable R_LOCAL_CACHE, or .R_Cache if this is unset. Environment variables for **R** can be set in the **R** session using Sys.setenv, or more conveniently (if the setting is intended to be generally made) by placing an entry in a .Renviron file in the user's home directory. Thus the following step from within **R** itself

```
> cat("\nR_LOCAL_CACHE=.R_Store\n",
+     file = "~/.Renviron", append=TRUE)
```

will add a line to the .Renviron in the user's home directory (or create one if none currently exists) which will change the default local cache name from ".R_Cache" to ".R_Store" permanently and generally for all four functions.

Of course, lib names can be specified as additional named arguments in each of the functions, but some care needs to be given. For convenience, the lib argument may be given either as an *unquoted name* or as an *explicitly quoted* character string. Hence

```
> Attach(lib = ".R_Store")
> Attach(lib = .R_Store)
```

are equivalent, but

```
> lib <- ".R_Store"
> Store(X, Y, Z, lib = lib)
```

would store three objects in a local cache called literally "`lib`".[2]

For local caches the `lib.loc` would normally be the current working directory, as given by `getwd()`, but users are free to vary this. The actual default is the value of the environment variable `R_LOCAL_LIB_LOC` or `getwd()` if this is unset. Again the user may change the default for all four functions by an entry in the file `~/.Renviron`, but this would be unusual.

Since directory names are usually *not* syntactic **R** names, the option of specifying them in the argument list as unquoted names is not available.

The main reason to change the `lib.loc` for a local cache is to add the objects from some other working directory cache to the present session. For example

```
> Attach(lib.loc = "..")
```

would attach the `.R_Cache` directory from the parent of the current working directory, making those stored objects accessible to the present session as well.

3 Centrally stored object caches

In some cases objects can usefully be made available for multiple projects. We term such caches as *central* stored object caches, though they do not differ from local caches in any way other than in their preferred location.

[2] There is no particular reason to have the cache name begin with a period, but since they are NOT to be accessed directly by the user, doing so makes them conveniently out of sight in many contexts.

3.1 Data and Utils variants

Each of the four primary functions as two variants, namely one with "Data" and the other with "Utils" added to the name. These are purely convenience functions which differ from the primary counterparts only in the default value for their lib and lib.loc. The default values for these are as follows:

lib.loc For all variants, the default lib.loc is the value of the environment variable R_CENTRAL_LIB_LOC or the user's home directory if this is not set. The user's home directory is found using path.expand("~").

lib This differs for the two variant kinds.

- For Data variants the default lib is the value of the environment variable R_CENTRAL_DATA, or .R_Data if this is unset, and
- For Utuls variants the default lib is the value of the environment variable R_CENTRAL_UTILS, or .R_Utils if this is unset.

The motivation for providing these variants is to give the user a convenient way of saving objects and making them generally visible across their **R** session. We envisage that the Data forms will be used for data sets and the Utils forms for utility functions.

One function we may wish to store and make generally available is the Vcells memory checking function we used above.

A simple but useful function to have available is the converse of the binary operator %in%, to identify which elements are *not* members of the set. One way to make such an operator is:

```
> `%ni%` <- Negate(`%in%`)
```

If for some reason the user preferred to use lsCache instead of Objects a simple way to do this without butchering the source package[3] woule be

```
> lsCache <- Objects
```

[3] Users are free to butcher the source package, of course, but if you do so, please *do not re-distribute it*, there's a sport.

These little utility functions may now be stored in the central utilities cache using:

```
> StoreUtils(Vcells, `%ni%`, lsCache)
```

Then in any future **R** session

```
> AttachUtils()
```

would make `Vcells`, `%ni%` and `lsCache`, in particular, available on demand.

Similarly

```
> AttachData()
```

would make the central data object cache visible and available on demand as well. Central data and utility object stores can be quite large without having an appreciable effect on memory, unless, of course, many large objects are required simultaneously.

The motivation for using environment variables to specify the default values is purely one of convenience. The user's home directory may be an appropriate `lib.loc` for the centrally stored object caches, but many users would already have a reserved directory for **R** related resources, in particular the add-on packages (as opposed to those which come with the release of **R** itself). A suitable place for the centrally stored object caches might be alongside this package directory. Thus a typical `~/.Rprofile` might include entries such as

```
R_LIBS_USER=~/R/lib/library
R_CENTRAL_LIB_LOC=~/R/lib/cache
```

3.2 Tricks with `.Rprofile`

In addition to `.Renviron` if there is a file `.Rprofile` in the user's home directory it contains **R** commands that are performed at startup for every **R** session.[4]

[4] Unless there is a `.Rprofile` in the current working directory, which will override one in the home directory.

The .Rprofile is intended to customise the working **R** environment, but this should be done with some care, particularly if the user is working as a member of a team and has to share **R** scripts. It is very easy to make **R** scripts that work in some customised contexts but fail in puzzling ways on another machine where the customisations are different.

For users who will want to use SOAR in most sessions it is inconvenient to have to remember to put library(SOAR) at the head of every script. One way round this is to add the line

```
options(defaultPackages = c(getOption("defaultPackages"), "SOAR"))
```

to ~/.Rprofile. This will mean that SOAR is included in the list of packages to be loaded at startup.

A more conditional way to add SOAR automatically to the search path is to use **R** autoloads. If we add lines such as

```
autoload("Store", "SOAR")
autoload("Objects", "SOAR")
autoload("Attach", "SOAR")
autoload("Remove", "SOAR")
autoload("Search", "SOAR")
```

to ~/.Rprofile, then if any of these five functions are used, the SOAR package is automatically attached to the search path, no questions asked. This is done by placing dummy autoload objects in the Autoload entry on the search path. These are promises like delayed assignments, but rather than loading objects into memory on demand, they attach the specified package on to the search path when the function in question is invoked.

With these autoloads in place we can add further lines to ~/.Rprofile such as:

```
SOAR::AttachUtils()
```

which will ensure that the central utilities cache is part of the search path at startup. We need the double colon construction here to let the interpreter know where to fine the function AttachUtils. However these variants on the main functions with Data or Utils tacked on to the end of their names are front-ends to the basic functions, and actually morph into a call to the relevant basic function. In turn, the basic function causes the package to be loaded through the autoload. So a side-effect will be that SOAR will be loaded on the search path at startup as well, just before the .R_Utils cache.

For the more intrepid user it is easy to make and add a full list of possible autoloads for any package and add it to ~/.Rprofile from within **R** itself using, for example:

```
> if(require(SOAR)) {
+     lst <- paste('autoload("', objects("package:SOAR"),
+                  '", "SOAR")\n', sep="")
+     cat("\n", lst, sep="", file = "~/.Rprofile", append = TRUE)
+ }
```

Now using any (exported) function from SOAR would cause the package to be loaded automatically, if not already. This sort of dodge can rapidly cause .Rprofile to become very long and bulky, of course!)

4 Scripts or saved data objects?

While it is convenient to carry objects over from one session to another, particularly during the period where an analysis is being developed, it can be a mistake to rely on **R** data objects gradually severing the link with the primary sources of the data. We would encourage users to make and keep scripts which construct all important data sets and analyses from primary sources and to be able to re-construct the entire process from them.

Stored object caches are a convenience intended to provide a way of managing memory, primarily, but also for sharing objects between sessions.

Objects placed in the central utilities directory would usually include functions on test prior to collecting them into coherent groups and making packages from them. As yet there is not neat way of documenting the objects in stored object caches, which is one reason to aim for packages as a more satisfactory and permanent way of holding such information.

A Some technical details

Structure of the cache. A cache directory consists of `.RData` files each corresponding to a single stored **R** object. The name of the file is related to the name of the object itself, but is *encoded* as some file systems have restrictions on file names. For example in WINDOWS file names are case insensitive whereas in **R** object names are case sensitive. Details of the encoding are given below.

Users are strongly advised *not* to access the files in the cache directory other than through **R**. Manual changes to the cache, and in particular, extra files or sub-directories in the cache directory will almost certainly cause problems when the cache is used again by the `SOAR` package, from which recovery may be very difficult.

Operation of the cache. When an existing cache is attached to the search path, as may be done explicitly by `Attach` or implicitly with `Store`, `Objects` or `Remove`, the following steps take place.

- The names of the files in the cache directory, apart from "`.`" and "`..`", are decoded into object names,
- An initially empty **R** environment is attached to the search path, into which objects of the same name as those stored in the cache are placed. These objects are promises, created by calls to `delayedAssign`, to load the corresponding entire object into the environment on demand.

Any reference to an object in the cache thus precipitates a `load` operation, bringing the entire object into central memory and replacing the promise in the attached cache.

Any change to an object in the cache causes a further copy of it to be loaded into the appropriate working environment to accept the changes. If, for example, the object is changed at the command line, this extra copy will be in the global environment.

Using `Attach` to re-attach a cache will reinstate all objects as promises again, thus freeing any memory that has been taken up by automatic loading of entire objects.

Storing an object with `Store` will (by default) remove it from the current working environment, store it in the cache directory and reinstate the object in the attached cache as a promise.

Removing objects from a cache using `Remove` clears both the promise from the attached cache *and* the corresponding `.RData` file from the cache directory. Thus the removal is permanent.

Birth and death of caches. If reference is made by any of the four main functions to a cache that does not presently exist, the cache directory is created (or, more precisely *will be created* when any object is stored there) and an empty cache is attached to the search path. If, however, `Attach` is used for this purpose a warning is issued that an empty cache has been attached. This is because it is never necessary to create a cache with `Attach`, so the user has most likely made a typo.

If all objects are removed from a cache using, for example,

```
> Remove(Objects())
```

the cache directory is *not* removed. Removing empty cache directories, if need be, should be done using normal file system operations outside **R** itself.

File name encoding The names for `.RData` files in the cache directory are encoded from the names of the objects themselves as follows:

- We assume that object names consist only of printable characters.
- Lower case letters are encoded as themselves.
- Upper case letters are encoded by preceding them with an `@` character.
- There are 10 other characters which are known to be problematical if used in file names on some operating systems. These are encoded as `@0`, . . . , `@9`. The correspondence is as shown below in **R** code:

  ```
  > bad <- c(" ", "<", ">", ":", "\"", "/", "\\", "|", "?", "*")
  > rpl <- paste("@", 0:9, sep = "")
  ```

- The `@` character itself is encoded as `@@`.
- All other printable characters, including the digits, are encoded as themselves.
- Finally the extension tag "`@.RData`" is added to the name without further `@`-modification.[5]

This rather simple encoding has proved to be adequate for all genuine cases, at least in UTF-8 locales. It has the virtue that most **R** objects in the cache can be easily recognised from the file name at a glance, which was a distinct advantage during debugging.

B Some packages with a similar functionality

As mentioned previously, David Brahm's `g.data` package ([1]) was antecedent to the present package. It offers effectively the same functionality as `SOAR`, but the usage is rather different. Users may wish to compare the two.

Roger Peng's package `filehash`, which provides **R** with hash files also allows objects to be stored on the disc and recalled automatically as needed. It uses the `makeActiveBinding` mechanism rather than the `delayedAssign` mechanism used by `SOAR` and `g.data`, which has some advantages, but at a slightly increased overhead time cost. There are strengths and weaknesses in both approaches and future versions of SOAR may offer a `makeActiveBinding` mechanism as an alternative to `delayedAssign` particularly for very large objects. For a discussion of the `filehash` package, see [3].

Mark Bravington's intricate and ingenious package `mvbutils` also offers a `makeActiveBinding` mechanism to cache objects through the function `mlazy`. Users should consult the help information for `mvbutils` for further details. This package has not yet been described in *R News* or the *R Journal*, but for an example of the Bravington style, readers may wish to look at his very effective `debug` package, described in [2].

[5] As well as being useful, this turns out to be necessary on WINDOWS where some very simple file names, such as e.g. "`con`" are actually illegal, *even if given an extension.*

C Links with ASOR: help for old friends

A precursor to the SOAR package was the package ASOR, which was never released through CRAN, but has been in fairly widespread trial use for some time. The name-change was made for the officially released package to draw attention to the fact that there are some important differences between it and ASOR, though there is a large degree of backward compatibility.[6]

The main differences with ASOR are as follows.

- There has been an extension to the file name encoding, as described in Appendix A above. This was needed to overcome some deficiencies in the old encoding leading to failures. For example objects such as `con` and `foo<-` can now be cached on WINDOWS which was not the case under the ASOR encoding.

  IMPORTANT NOTE: If a cache created under ASOR is attached, directly or indirectly with SOAR, *the file names will be re-encoded under the new scheme*, with a warning that this is taking place. At this point the cache will not be readable with ASOR functions: there is no easy road back. However this is a quick and simple operation and has proved to be very reliable. Nevertheless users should take to heart the advice given in sub-section 4 on page 13 and make sure that they have scripts available to re-create all important **R** objects rather than relying solely on stored object caches.

- There has been a change to the default local stored object cache from `.R_Store` to `.R_Cache`. This is partly to reflect the change in terminology, but also to make it easier for people to operate with ASOR for a while longer while feeling their way with SOAR, if they so wish. It could become very confusing if both ASOR and SOAR packages were in use in the same **R** session. Users are warned against this.

  There has been effectively no change to the default `lib` names for the centrally stored object caches.[7]

[6] Another reason to change the name is that speakers with a sufficiently broad Australian accent used to pronounce the old package name as "eyesore".

[7] The `Data` and `Utils` variants did exist in ASOR but were largely unadvertised features and as far as I know I am the only person who knew how to use them!

- There has been a change to the way the default `lib` and `lib.loc` names are specified, now using environment variables. This is a more flexible system than the previous one and offers a way for users to prescribe their own preferences in this regard in a simple and global way.

- The functions `Save`, `SaveData` and `SaveUtils` have been removed. These were complete aliases for the corresponding forms with `Store`. This is because there is already a function `Save` in the `Hmisc` package.[8]

- The function `Search` has been added, mainly to provide a way for users to separate multiple stored object caches on the search path with the same primary name, but with different `lib.loc`s.

References

[1] David E. Brahm. Delayed data packages. *R News*, 2(3):11–12, December 2002.

[2] Mark Bravington. Debugging without (too many) tears. *R News*, 3(3):29–32, December 2003.

[3] Roger D. Peng. Interacting with data using the filehash package. *R News*, 6(4):19–24, October 2006.

[8] Originally `ASOR` only had the `Save` forms, but the `Store` forms were added as a preferable alternative when the author became aware of the clash with `Hmisc`. The `Save` forms have now passed effectively from deprecated to defunct.