Lazy Loading and Packages in R 2.0.0

by Brian D. Ripley

Lazy Loading

One of the basic differences between R and S is how objects are stored. S stores its objects as files on the file system, whereas R stores objects in memory, and uses *garbage collection* from time to time to clear out unused objects. This led to some practical differences:

1. R can access objects faster, particularly on first use (although the difference is not as large as one might think, as both S and the file system will do caching).
2. R slows down the more objects that there are in memory.
3. R's performance is more sensitive to the number and size of packages that are loaded.

These differences are blurred considerably by the advent of *lazy loading* in R 2.0.0. This is optional, but is used by all the standard and recommended packages, and by default when a package with more than 25Kb of R code is installed (about 45% of those currently on CRAN). This is 'lazy' in the same sense as *lazy evaluation*, that is objects are not loaded into memory until they are actually used. This leads to some immediate differences:

1. R uses much less memory on startup: on my 32-bit system, 4.2Mb rather than 12.5Mb. Such a gain would have been very valuable in the early days of R, but nowadays most of us have far more RAM than those numbers.
2. The start-up time is much shorter: 0.4s on my system. This is almost entirely because many fewer objects have been loaded into memory, and loading them takes time.
3. Tasks run a little faster, as garbage collection takes much less time (again, because there are many fewer objects to deal with).
4. There is much less penalty in loading up lots of packages at the beginning of a session. (There is some, and loading R with just the **base** package takes under 0.1s.)

For data, too

Another R/S difference has been the use of `data()` in R. As I understand it this arose because data objects are usually large and not used very often. However, we can apply lazy-loading to datasets as well as to other R objects, and the **MASS** package has done so since early 2003. This is optional when installing packages in 2.0.0 (and not the default), but applies to all standard and most recommended packages. So for example to make use of the dataset `heart` in package **survival**, just refer to it by name in your code.

There is one difference to watch out for: `data(heart)` loaded a copy of `heart` into the workspace, from the package highest on the search path offering such a dataset. If you subsequently altered `heart`, you got the altered copy, but using `data(heart)` again gave you the original version. It still does, and is probably the only reason to continue to use `data` with an argument.

For packages with namespaces there is a subtle difference: data objects are in `package:foo` but not in `namespace:foo`. This means that data set `fig` **cannot** be accessed as `foo::fig`. The reason is again subtle: if the objects were in the namespace then functions in **foo** would find `fig` from the namespace rather than the object of that name first on the search path, and modifications to `fig` would be ignored by some functions but by not others.

Under the bonnet (or 'hood')

The implementation of lazy loading makes use of *promises*, which are user-visible through the use of the `delay` function. When R wants to find an object, it looks along a search path determined by the scope rules through a series of environments until it encounters one with a pointer to an object matching the name. So when the name `heart` is encountered in R code, R searches until it finds a matching variable, probably in `package:survival`. The pointer it would find there is to an object of type `PROMSXP` which contains instructions on how to get the real object, and evaluating it follows those instructions. The following shows the pattern

```
> library(survival)
> dump("heart", "", evaluate = FALSE)
heart <- delay(lazyLoadDBfetch(key, datafile,
       compressed, envhook), <environment>)
Warning message: deparse may be incomplete
```

The actual objects are stored in a simple database, in a format akin to the `.rda` objects produced by `save(compress = TRUE, ascii = FALSE)`. Function `lazyLoadDBfetch` fetches objects from such databases, which are stored as two files with extensions `.rdb` and `.rdx` (an index file). Readers nay be puzzled as to how `lazyLoadDBfetch` knows which object to fetch, as `key` seems to be unspecified. The answer lies (literally) in the environment shown as `<environment>` which was not dumped. The code in function `lazyLoad` contains essentially

```
wrap <- function(key) {
    key <- key
    mkpromise(expr, environment())
}
for (i in along(vars))
    set(vars[i], wrap(map$variables[[i]]), envir)
```

so key is found from the immediate environment and the remaining arguments from the enclosing environment of that environment, the body of lazyLoad.

This happens from normal R code completely transparently, perhaps with a very small delay when an object is first used. We can see how much by a rather unrealistic test:

```
> all.objects <-
    unlist(lapply(search(), ls, all.names=TRUE))
> system.time(sapply(all.objects,
                     function(x) get(x); TRUE),
            gcFirst = TRUE)
[1] 0.66 0.06 0.71 0.00 0.00
> system.time(sapply(all.objects,
                     function(x) get(x); TRUE),
            gcFirst = TRUE)
[1] 0.03 0.00 0.03 0.00 0.00
```

Note the use of the new gcFirst argument to system.time. This tells us that the time saved in start up would be lost if you were to load all 2176 objects on the search path (and there are still hidden objects in namespaces that have not been accessed).

People writing C code to manipulate R objects may need to be aware of this, although we have only encountered a handful of examples where promises need to be evaluated explicitly, all in R's graphical packages.

Note that when we said that there were many fewer objects to garbage collect, that does not mean fewer *named* objects, since each named object is still there, perhaps as a promise. It is rather that we do not have in memory the components of a list, the elements of a character vector and the components of the parse tree of a function, each of which are R objects. We can see this *via*

```
> gc()
         used (Mb) gc trigger (Mb)
Ncells 140236  3.8     350000  9.4
Vcells  52911  0.5     786432  6.0
> memory.profile()
    NILSXP     SYMSXP    LISTSXP     CLOSXP
         1       4565      70606        959
    ENVSXP    PROMSXP    LANGSXP SPECIALSXP
      2416       2724      27886        143
BUILTINSXP    CHARSXP     LGLSXP
       912      13788       1080          0
               INTSXP    REALSXP    CPLXSXP
         0       2303       2986          0
    STRSXP     DOTSXP     ANYSXP     VECSXP
      8759          0          0       1313
   EXPRSXP   BCODESXP  EXTPTRSXP WEAKREFSXP
         0          0         10          0
```

```
> sapply(all.objects,
         function(x) get(x); TRUE) -> junk
> gc()
         used (Mb) gc trigger (Mb)
Ncells 429189 11.5     531268 14.2
Vcells 245039  1.9     786432  6.0
> memory.profile()
    NILSXP     SYMSXP    LISTSXP     CLOSXP
         1       7405     222887       3640
    ENVSXP    PROMSXP    LANGSXP SPECIALSXP
       822       2906     101110        208
BUILTINSXP    CHARSXP     LGLSXP
      1176      44308       4403          0
               INTSXP    REALSXP    CPLXSXP
         0        824      11710          9
    STRSXP     DOTSXP     ANYSXP     VECSXP
     24877          0          0       2825
   EXPRSXP   BCODESXP  EXTPTRSXP WEAKREFSXP
         0          0        106          0
```

Notice the increase in the number of LISTSXP and LANGSXP (principally storing parsed functions) and STRSXP and CHARSXP (character vectors and their elements), and in the sum (the number of objects has trebled to over 400,000). Occasionally people say on R-help that they 'have no objects in memory', but R starts out with hundreds of thousands of objects.

Installing packages

We have taken the opportunity of starting the 2.x.y series of R to require all packages to be reinstalled, and to do more computation when they are installed. Some of this is related to the move to lazy loading.

- The 'DESCRIPTION' and 'NAMESPACE' files are read and parsed, and stored in a binary format in the installed package's 'Meta' subdirectory.

- If either lazy loading of R code or a saved image has been requested, we need to load the code into memory and dump the objects created to a database or a single file 'all.rda'. This means the code has to parse correctly (not normally checked during installation), and all the packages needed to load the code have to be already installed.

  This is simplified by accurate use of the 'Describe', 'Suggests' and 'Import' fields in the 'DESCRIPTION' file: see below.

- We find out just what data() can do. Previously there was no means of finding out what, say, data(sunspot) did without trying it (and in the base system it created objects sunspot.months and sunspot.years but not sunspot, but not after package **lattice** was loaded). So we do try loading all the possible datasets—this not only tests that they work but gives us an index of datasets which is stored in binary format and used by `data()` (with no argument).

We have always said any R code used to make datasets has to be self-sufficient, and now this is checked.

- If lazy loading of data is requested, the datasets found in the previous step are dumped into a database in the package directory `data`.

If we need to have one package installed to install another we have a dependency graph amongst packages. Fortuitously, installing CRAN packages in alphabetical order has worked (and still did at the time of writing), even though for example **RMySQL** required **DBI**. However, this is not true of BioConductor packages and may not remain true for CRAN, but `install.packages` is able to work out a feasible install order and use that. (It is also now capable of finding all the packages which need already to be installed and installing those first: just ask for its help!)

One problem with package **A** `require()`ing package **B** in `.First.lib`/`.onLoad` was that package **B** would get loaded after package **A** and so precede it on the search path. This was particularly problematic if **A** made a function in **B** into an S4 generic, and the file 'install.R' was used to circumvent this (but this only worked because it did not work as documented!).

We now have a much cleaner mechanism. All packages mentioned in the '`Depends`' field of the 'DESCRIPTION' file of a package are loaded in the order they are mentioned, both before the package is prepared for lazy-loading/save image and before it is loaded by `library`. Many packages currently have unneeded entries in their '`Depends`' field (often to packages that no longer exist) and will hopefully be revised soon. The current description from *'Writing R Extensions'* is

- Packages whose namespace only is needed to load the package using `library(`*pkgname*`)` must be listed in the '`Imports`' field.
- Packages that need to attached to successfully load the package using `library(`*pkgname*`)` must be listed in the '`Depends`' field.
- All packages that are needed to successfully run `R CMD check` on the package must be listed in one of '`Depends`' or '`Suggests`' or '`Imports`'.

For Package Writers

The previous section ended with a plea for accurate 'DESCRIPTION' files. The 'DESCRIPTION' file is where a package writer can specify if lazy loading of code is to be allowed or mandated or disallowed (*via* the '`LazyLoad`' field), and similarly for lazy loading of datasets (*via* the '`LazyData`' field). Please make use of these, as otherwise a package can be installed with options to `R CMD INSTALL` that may override your intentions and make your documentation inaccurate.

Large packages that make use of saved images would benefit from being converted to lazy loading. It is possible to first save an image then convert the saved image to lazy-loading, but this is almost never necessary. The normal conversion route is to get the right '`Depends`' and '`Imports`' fields, add '`LazyLoad: yes`' then remove the 'install.R' file.

For a few packages lazy loading will be of little benefit. One is John Fox's **Rcmdr**, which accesses virtually all its functions on startup to build its menus.

Acknowledgement

Lazy loading was (yet another) idea from Luke Tierney, who wrote the first implementation as a package **lazyload**.

Brian D. Ripley
University of Oxford, UK
`ripley@stats.ox.ac.uk`