The Saves Package

an approximate benchmark of performance issues while loading datasets

Gergely Daróczi
gergely@snowl.net

October 13, 2011

1 Introduction

The purpose of this package is to be able to save and load only the needed variables/columns of a dataframe. This is done with special binary files (tar archives), which seems to be a lot faster method than loading the whole binary object (Rdata files) via the basic load() function, or than loading columns from SQLite/MySQL databases via SQL commands. Performance gain on SSD drives is a lot more sensible compared to basic load() function.

The performance improvement gained by loading only the chosen variables in binary format can be useful in some special cases (e.g. where merging data tables is not an option and very different datasets are needed for reporting), but be sure if using this package that you really need this, as non-standard file formats are used!

2 Requirements

The package has no special requirements, it does not depend on any nonstandard package.

3 Usage

Basic usage requires the user to save R objects (lists and data frames) to a special binary format via the saves() function. For example saving the 'diamonds' data frame from ggplot2 package is done as follows:

```
> saves(diamonds)
```

This command will create a file named to "diamonds.Rdatas" in the current working directory. This file is a compressed tar archive which contains the binary objects of all variables from the diamonds data frame (by issuing a simple save() on each of them).

Custom filename can be specified via the "file" parameter, see:

```
> saves(diamonds, file="foo.bar")
```

Loading of the saved binary format can be done via the loads() function:

```
> df <- loads("diamonds", variables = c("color", "cut"))
```

Of course the vector of variables to be load have to be specified. If you need to load all variables, you should use load() instead of loads().

4 Advanced (crazy) usage

Calling the "ultra.fast" option in saves() and loads() will make the commands run a lot faster but without any check made on user input, so be careful and use only if you are really know what you are up to!

This option eliminates all inner control over given paramaters, checks are left out for greater performance, and also: no compression is done while saving the variables, all files are put in a new working directory.

To save a data frame for later "ultra.fast" loading:

```
> saves(diamonds, ultra.fast = TRUE)
```

Loading of the data frame (or list) is done via the loads() function just like above with an extra parameter:

```
> df <- loads("diamonds", variables = c("color", "cut"), ultra.fast = TRUE)
> str(df)
List of 2
 $ color: Factor w/ 7 levels "D","E","F","G",..: 2 2 2 6 7 7 6 5 2 5 ...
 $ cut  : Factor w/ 5 levels "Fair","Good",..: 5 4 2 4 2 3 3 3 1 3 ...
```

It is advised not to convert the R object to data frame from the list format for performance issues, so specifying to.data.frame = FALSE parameter might be a good choice (thought this is the default setting, it might change in the future):

```
> df <- loads("diamonds", variables = c("color", "cut"), to.data.frame = FALSE, ultra.fast = TRUE)
```

5 Benchmarks

The benchmark was run on a HP 6715b laptop (AMD64 X2 TL-60 2Ghz dual-core, 4 Gb RAM) with a standard SSD drive (). The most important part of the procedures were repeated in server environment also (2xIntel QuadCore 5410 at 2.3 Ghz, 8 Gb RAM, 2x500 Gb WD RE3 HDD in Raid1). The results shown in this paper are based on the experiments run on the laptop computer, the server's results are marked/annotated accordingly.

To be able to compare the different procedures of loading data, the following environment was set.

Loading the required packages:

```
> library(microbenchmark)       # for benchmarking (instead of system.time)

> library(foreign)                              # different loading algorithms and formats
> library(sqldf)                                #
> library(RMySQL)                               #
> con <- dbConnect(MySQL(), user='foo', dbname='bar', host='localhost', password='*****')
> library(saves)                                #

> library(ggplot2)              # for plotting and also for reading diamonds dataset
```

Getting required data:

1. diamonds data frame from ggplot2 via: dataset(diamonds) [53.940 obs. of 10 variables]

2. European Social Survey (Hungary, 4. wave) available at: http://www.esshu.hu/letoltheto-adatbazisok [1.544 obs. of 508 variables]

Transform each data frame to the required formats:

```
> save("diamonds", file="diamonds.Rdata")
> save("diamonds", file="diamonds-nocompress.Rdata", compress=FALSE, precheck=FALSE)
> write.table(diamonds, file="diamonds.csv", sep = ",", quote = FALSE, row.names=FALSE)
> names(data)[6] <- "tablee"                         # as the default "table" is internal in MySQL
> dbWriteTable(con, "diamonds", data, overwrite = TRUE)
> saves(diamonds)
> saves(diamonds, file="diamonds-uf.Rdatas", ultra.fast = TRUE)
> rm(data)
```

File transformation to SPSS sav format was done in external software from the above made csv files. All the same was done for the ESS dataset (ESS_HUN_4).

Functions to test:

```
> sav <- function() read.spss('diamonds.sav')
> csv <- function() read.csv('diamonds.csv')
> classes <- sapply(diamonds, class)
> csv2 <- function() read.table('diamonds.csv', header=TRUE, sep=",", comment.char="",
                 stringsAsFactors=FALSE, colClasses=classes,  nrows=53940)
> sqldf <- function() {
        f <- file("diamonds.csv")
```

```
        sqldf::sqldf("select carat, clarity from f", dbname = tempfile(),
                file.format = list(header = TRUE, row.names = FALSE, sep=","), drv="sqldf.driver")
}
> sql <- function() {
        data <- dbReadTable(con, 'diamonds')
        query <-('SELECT carat, clarity FROM diamonds')
        dbGetQuery(con, query)
}
> binary <- function() load("diamonds.Rdata")
> binary2 <- function() load("diamonds-nocompress.Rdata")
> loads <- function() saves::loads("diamonds.Rdatas", c('carat', 'clarity'))
> loads2<- function() saves::loads("diamonds", c('carat', 'clarity'), ultra.fast = TRUE)
```

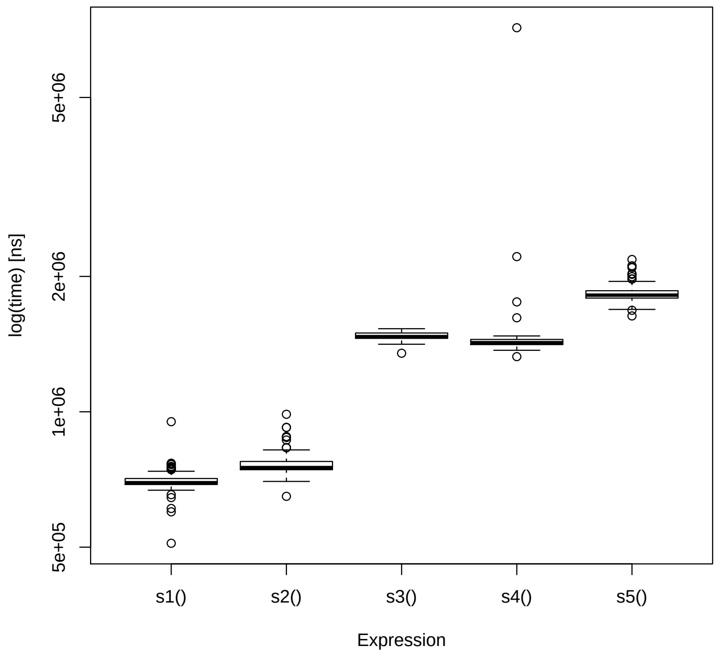
And the appropriate form of the above for the ESS_HUN_4 data frame also:

```
> sav <- function() read.spss('ESS_HUN_4.sav')
> csv <- function() read.csv('ESS_HUN_4.csv')
> classes <- sapply(diamonds, class)
> csv2 <- function() read.table('ESS_HUN_4.csv', header=TRUE, sep=",", comment.char="",
                stringsAsFactors=FALSE, colClasses=classes, nrows=53940)
> sqldf <- function() {
        f <- file("ESS_HUN_4.csv")
        sqldf::sqldf("select 'length', 'a1' from f", dbname = tempfile(),
                file.format = list(header = TRUE, row.names = FALSE, sep=","), drv="sqldf.driver")
}
> sql <- function() {
        data <- dbReadTable(con, 'data')
        query <-('SELECT length, a1 FROM data')
        dbGetQuery(con, query)
}
> binary <- function() load("ESS_HUN_4.Rdata")
> binary2 <- function() load("ESS_HUN_4-nocompress.Rdata")
> loads <- function() saves::loads("ESS_HUN_4.Rdatas", c('length', 'a1'))
> loads2 <- function() saves::loads("ESS_HUN_4", c('length', 'a1'), ultra.fast = TRUE)
```

Where the *sav()* stands for reading SPSS sav file, *csv()* for reading comma separated values, *csv2()* for the same but a lot optimized way, *sqldf()* and *sql()* are reading the data frames from a virtual and a real MySQL data frame. *binary()* reads the object from Rdata, where *binary2()* reads an uncompressed file. Reading data frame with *loads()* and *loads2()* are implemented in this package. The latter uses the *ultra.fast = TRUE* parameter.

The difference of the required time to save data frames in binary format with the different parameters of *save()* (compress, precheck, ascii) can be simulated easily as follows:

```
> x <- rnorm(1000)
> s1 <- function() save("x", file="temp", compress=FALSE, precheck=FALSE)
> s2 <- function() save("x", file="temp", compress=FALSE)
> s3 <- function() save("x", file="temp")
> s4 <- function() save("x", file="temp", precheck=FALSE)
> s5 <- function() save("x", file="temp", precheck=FALSE, ascii=TRUE)
> res_saves <- microbenchmark(s1(),s2(),s3(),s4(),s5(), times=100)
> boxplot(res_saves)
```



For the sake of the Solid State Drive's longer life-span, the benhmarking processes (writing and loading from disks) were run only a hundred times. Previous experiments showed that longer benchmarking process did not resulted in more suprising details, but I welcome any donation of hardware for these purposes! :)

Of course only reading the different types of data frames (without running estimates against write speeds) from disk would not lower the life-span of the drives, but running the processes 10.000 times will only be attached to the next version of this paper.

Benchmarking of the above functions was done via:

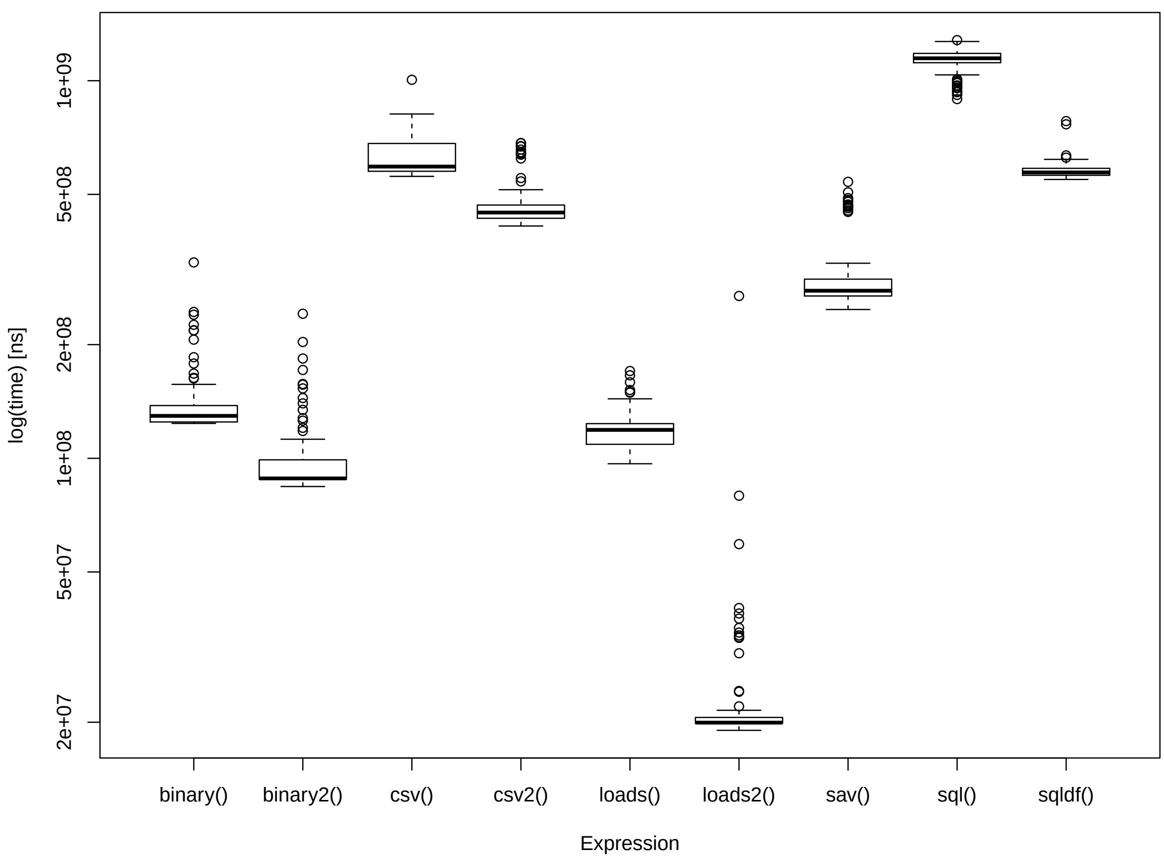
```
> microbenchmark(sav(), csv(), csv2(), sqldf(), sql(), binary(), binary2(), loads(), loads2(), times=100)
```

for both datasets (diamonds and ESS_HUN_4).

6 Results - diamonds

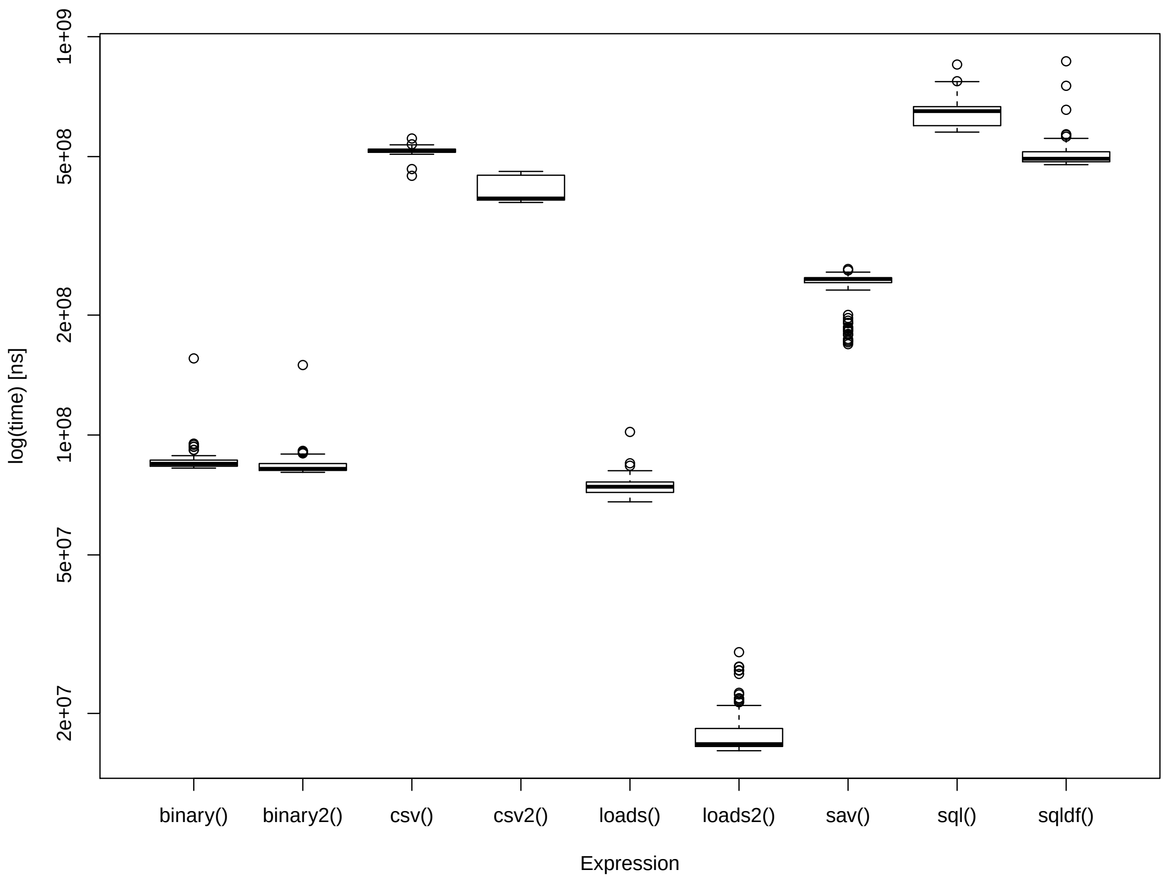
Run on the laptop with SSD attached (high IO capability):

```
Unit: nanoeconds
                 min         lq     median         uq        max
sav()      247768186  269080764  277842078  298155661  539805845
csv()      558015701  575679546  592369539  681940860 1005660411
csv2()     412340206  432280842  447627948  468533077  684155492
sqldf()    547616215  561715835  571669320  586092861  781810538
sql()      894087049 1116075540 1146487458 1181165774 1280771316
binary()   123704605  124799280  129533188  137940692  330225725
binary2()   84195442   88034138   88451503   99073221  241455191
loads()     96727569  108917898  118919434  123527491  170300594
loads2()    19032016   19889932   19952370   20595877  269095433
```



Run in server environment (high CPU, but low IO capability):

```
Unit: nanoeconds
                 min         lq     median         uq        max
sav()      169058869  241284026  246358267  248229742  261133064
csv()      447431996  512482638  517086442  522108758  555228387
csv2()     383554684  388878624  392469560  448931604  458952437
sqldf()    477170533  485206866  493820346  514099318  867471078
sql()      576023018  597863746  650239598  667263688  851414226
binary()    82564640   83481720   84699074   86497266  155718942
binary2()   80594466   81493388   82258268   84792060  149867083
loads()     67956000   71764023   74152600   76218841  101806185
loads2()    16121788   16525269   16748340   18332214   28500877
```

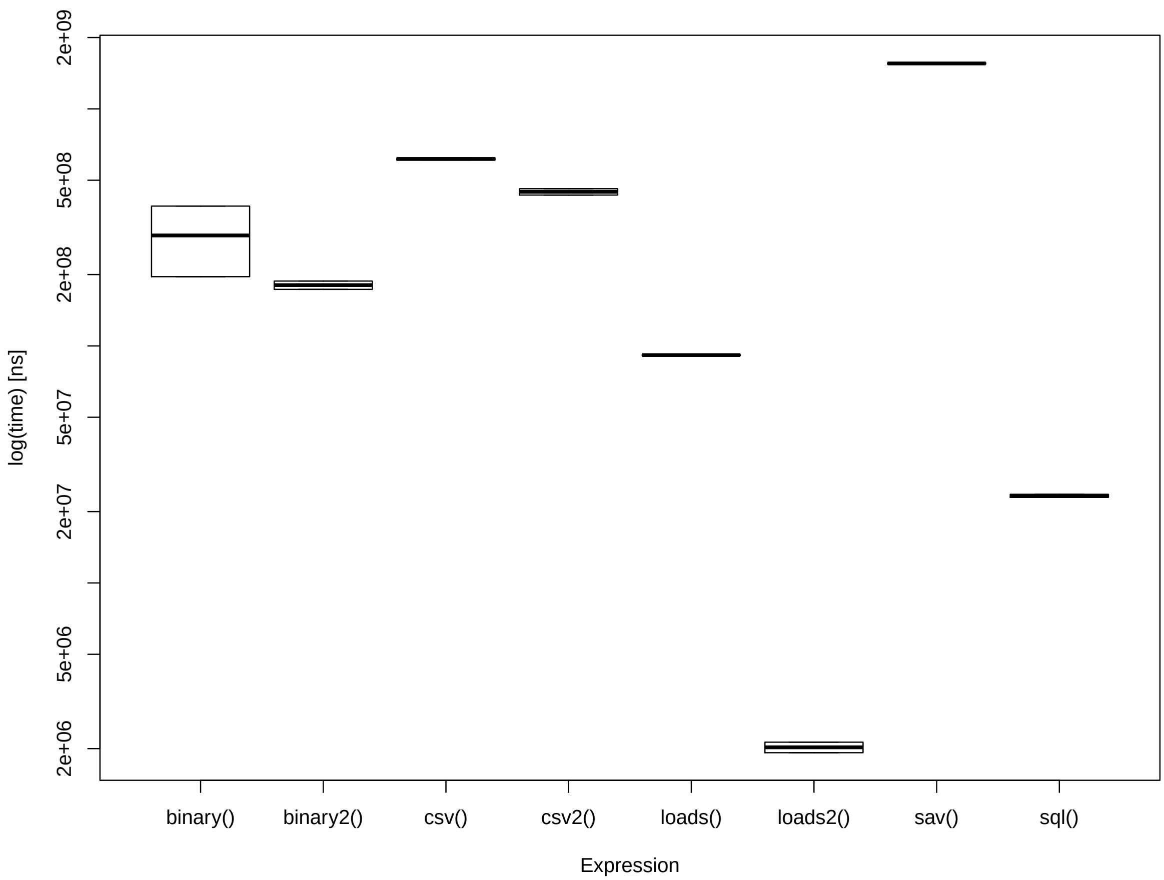


7 Results - ESS

Run on the laptop with SSD attached (high IO capability):

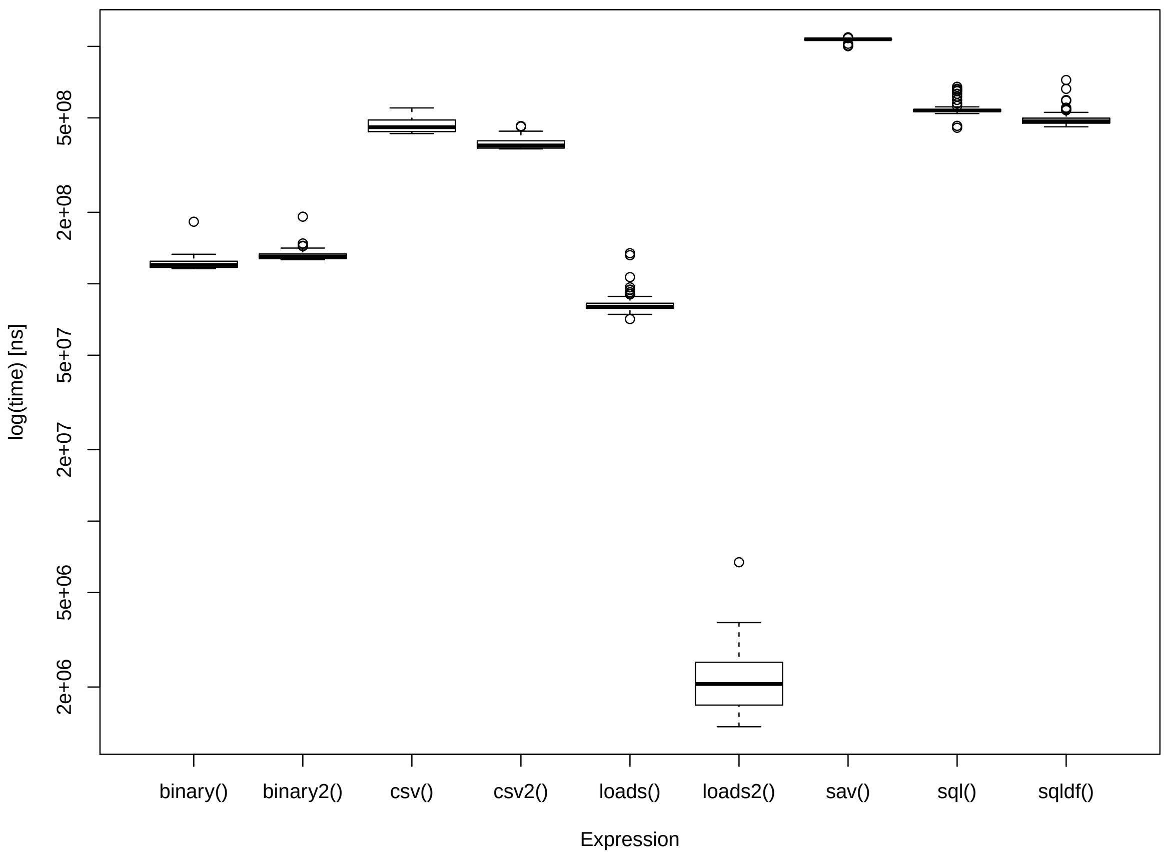
Unit: nanoeconds

	min	lq	median	uq	max
sav()	1546345401	1546345401	1555045257	1563745113	1563745113
csv()	608659349	608659349	614847051	621034753	621034753
csv2()	432886076	432886076	446772544	460659013	460659013
sql()	22967372	22967372	23307353	23647334	23647334
binary()	195957930	195957930	292467874	388977819	388977819
binary2()	173168231	173168231	180481340	187794448	187794448
loads()	91112136	91112136	91452398	91792659	91792659
loads2()	1922003	1922003	2025506	2129010	2129010



Run in server environment (high CPU, but low IO capability):

```
Unit: nanoeconds
                  min         lq     median         uq        max
sav()      1002706199 1069822150 1072309107 1074794738 1092789416
csv()       429448685  437668012  456707466  489854026  550664349
csv2()      370109696  372899655  382606256  400250634  461689820
sql()       454294835  532463950  537078038  542445448  676130528
sqldf()     458514868  475001036  483471416  498829774  722040292
binary()    115832235  117273083  120065867  124422444  182590704
binary2()   126332841  127640386  130243322  133516098  191742881
loads()      71019188   78846217   80092247   82805980  134582247
loads2()      1360472    1678182    2059142    2542211    6710979
```



Any feedback is welcome!