

A Tale of Two Packages

Tim Hesterberg¹

¹Google

UseR! 2012, June 15, 2012



Outline

- ▶ `dataframe` Package
- ▶ `aggregate` Package

Overview for dataframe

- ▶ Replace `src / library /base/R/dataframe.R`
- ▶ Same functionality (except for `round(x)`)
- ▶ Makes fewer copies of the inputs
- ▶ Faster (21% for creation and column subscripting, 14% for row subscripting)

I'll give benchmarks, and programming tips for your own code.

Make Fewer Copies

	R-2.14.2	+library(dataframe)
<code>as.data.frame(y)</code>	3	1
<code>data.frame(y)</code>	6	3
<code>data.frame(y, z)</code>	7 each	3 each
<code>as.data.frame(l)</code>	8	3
<code>data.frame(l)</code>	13	5
<code>d["y"]</code>	2	1
<code>d[, "y", drop=F]</code>	2	1

`y` and `z` are vectors, `l` and `L` are lists.

Not all copies are in memory at once.

Make Fewer Copies

	R-2.14.2	+library(dataframe)
<code>d\$z <- z</code>	3,2	1,1
<code>d[["z"]] <- z</code>	4,3	2,1
<code>d[, "z"] <- z</code>	6,4,2	2,2,1
<code>d["z"] <- z</code>	6,5,2	2,2,1
<code>d["z"] <- list(z=z)</code>	6,3,2	2,2,1
<code>d["z"] <- L</code>	6,2,2	2,1,1

copies of: (old object, new column, row names)

Programming Tricks

I'll show some programming tricks to save memory. These are from [dataframe/inst/doc/differences.txt](#).

For background, these are equivalent pairs:

```
x[i, j]  
"[.data.frame"](x, i, j)
```

```
x[i, j] <- value  
x <- "[<-.data.frame"](x, i, j, value)
```

```
attributes(x) <- value  
x <- "attributes<-"(x, value)
```

Add Attributes All At Once

Old:

```
value <- list(x)
if(!optional) names(value) <- nm
attr(value, "row.names") <- row.names
class(value) <- "data.frame"
value
```

New:

```
"attributes <-"(list(x),
                 c(if(!optional) list(names = nm),
                   list(row.names = row.names,
                        class = "data.frame")))
```

The old code makes 3 copies of `x`, the new code 1 copy.

Luke Tierney may fix this.



Strip Attributes Only If Needed

Old code:

```
if (is.atomic(value))  
  names(value) <- NULL
```

New code:

```
if (is.atomic(value)  
    && !is.null(names(value)))  
  names(value) <- NULL
```

The new code makes one fewer copy of `value`.

Call eval using names, not the object

Old code (`as.data.frame.list`):

```
x <- eval(as.call(c(expression(data.frame),
                        x,
                        check.names = etc.)))
```

New code:

```
cn <- names(x)
x <- eval(as.call(c(expression(data.frame),
                        lapply(cn, as.name),
                        check.names = etc.)),
          envir = x)
```

Reduce 8 copies to 3 (caveat: not all here). Extra copies:

- ▶ convert `x` to an expression
- ▶ in `c()`
- ▶ in `as.call()`

Avoid `list(...)`, instead use `..1`, `..2`, etc.

Old code (`data.frame`):

```
x <- list(...)  
n <- length(x)  
vnames <- names(x)  
...  
for(i in seq_len(n)) {  
  computations using x[[i]]  
}
```

New code:

```
n <- (function(...)(nargs(...)))(...)  
vnames <- names(match.call(expand.dots = F)$...)  
...  
for(i in seq_len(n)) {  
  x0 <- eval(parse(text = paste0("..", i)))[[1]])  
  computations using x0  
}
```



Use Small Objects instead of list

Old code uses these commands inside loops:

```
xi <- vlist [[ i ]]  
...  
vlist [[ i ] <- xi
```

New code:

```
xi <- get(vlistNames [ i ])  
...  
assign(vlistNames [ i ], xi)
```

Two Intentional Differences

- ▶ Bug fix: if `x` has 2 columns, `x[[4]] <- value` doesn't create an illegal data frame.
- ▶ `round(x)` rounds numerical columns, skips factor and character rather than fail. Similarly `trunc`, `abs`, `signif`, ...

Profiling tools

- ▶ `tracemem` to find which functions make copies.
- ▶ Use `Rprofmem` and stubs to find where inside functions copies are made.

```
force(nm)
z <- seq(10^3+2*1)
nrows <- length(x)
z <- seq(10^3+2*2)
```

- ▶ Benchmarks using `Rprofmem`. Must compile R with `--enable-memory-profiling`

Summary for dataframe

- ▶ Ugly programming tricks
- ▶ Make fewer copies of the data, hence faster.
- ▶ Impetus for improving *R*
- ▶ Plan: faster row subscripting
- ▶ Plan: restructure

Overview for aggregate

- ▶ `mean`, `sum`, `var`, `stdev` *by* one or more factors.
- ▶ Add weights to `mean`, `colMeans`, `table`,
- ▶ Simultaneous subscripting and aggregation.
- ▶ Fast `anyNA`
- ▶ `colMinus`, . . .
- ▶ `lapply2`, `sapply2`
- ▶ `ifelse1`

Aggregation by one or more factors

```
byMeans(x, by, na.rm = FALSE, weights = NULL)
groupMeans(x, group, na.rm=FALSE, weights=NULL)
```

- ▶ `by` = list of one or more factors or integers
- ▶ `group` = single factor or integer
- ▶ Sums, Means, Vars, Stdevs

Grouping:	One		Two	
	Vector	DF	Vector	DF
groupSums	.065	.23		
bySums	.18	.33	1.09	1.19
tabulate	.069			
aggregate	12.7	24.	14.	26.
sapply(split)	.23	4.2	1.3	240.
rowsum	.45	.49		
tapply	3.7	4.5		




```
colMeans(x, na.rm=FALSE, dims=1, weights=NULL)
```

Add `weights` to: `mean`, `colMeans`, `colSums`, `colVars`, `colStdevs`, `tabulate` and `table`

Might remove `mean`.

Plan to add other functions like `colMins`, `colMaxs`, `colRanges`.

Aggregation while subscripting

```
indexMeans(x, indices, na.rm = FALSE)
```

`indexSums` and `indexMeans` subscript vectors, matrices or data frames and calculate summaries, like `colMeans(x[indices[, i], , drop=TRUE])`

14–80× faster than combination of loop, subscripting and `colSums`, for 1000 samples. 1.5–2.0× slower for one sample.

```
anyNA(x)
```

Compare to `any(is.na(x))`.

Uses `.Call`, makes no copies of data, does not convert a data frame to a matrix.

3.6× faster for vectors, 17× faster for data frames; even more if early missing values.

colDivide and %col/%

```
colDivide(x, STATS)      (or)      x %col/% STATS
```

Also (col,row)×(Plus,Minus,Times,Divide)

Alternatives (some terrible):

```
x %col/% colSums(x)
```

```
x / rep(colSums(x), each = nrow(x))
```

```
sweep(x, 2, colSums(x), "/" )
```

```
scale(x, center = FALSE, scale = colSums(x))
```

```
t(t(x) / colSums(x))
```

```
x / matrix(colSums(x), nrow(x), ncol(x), byrow=TRUE)
```

```
x %*% diag(1/colSums(x))
```



ifelse1

```
ifelse1(test, x, y, ...)
```

Compare:

```
if(cond) {  
  result <- x  
} else {  
  result <- y  
}
```

```
result <- (if(cond) x else y)
```

```
result <- ifelse1(cond, x, y)
```

lapply2 and sapply2

```
lapply2(X1, X2, FUN, ...)}  
sapply2(X1, X2, FUN, ..., simplify = TRUE,  
        USE.NAMES = TRUE)
```

Compare:

```
lapply(seq(along=x), function(i) f(x[[i]], y[[i]]))  
lapply2(x, y, f)
```

1.5× slower than `mapply` for small inputs, 420× faster for large inputs.

df.aggregate2

```
df.aggregate2(data, by,
              aggregated.columns=setdiff(names(data), by),
              FUN = colSums, agg.names = NULL)
```

Here `by` = character vector of names in `data`.

This was an afterthought—I didn't realize the potential of this until investigating the `data.table` package.

Summary for aggregate

- ▶ Variety of aggregation functions
- ▶ Well-loved at Google
- ▶ Plan: use the `df.aggregate2` idea more widely.
- ▶ Plan: more code call `.C`, maybe `.Internal`.
- ▶ Plan: Other “might” items above

Another package: `data.table`

- ▶ Huge potential.
- ▶ Faster than data frames
- ▶ Fast aggregation
- ▶ Danger: make a copy, modify, the original is changed.
- ▶ Danger: different syntax, `x[, 1]` gives `1`.
- ▶ Annoy me: converts character to factor.

<http://timhesterberg.home.comcast.net/Rpackages>
[http://timhesterberg.home.comcast.net/articles/
Efficient.R](http://timhesterberg.home.comcast.net/articles/Efficient.R)