* [Motivation](https://r-some-blog.blogspot.com/2018/12/smartly-select-and-mutate-data-frame_9.html" \l "motivation" \t "_blank)
* [Column operations](https://r-some-blog.blogspot.com/2018/12/smartly-select-and-mutate-data-frame_9.html#column-operations)
  + [Add](https://r-some-blog.blogspot.com/2018/12/smartly-select-and-mutate-data-frame_9.html#add)
  + [Modify](https://r-some-blog.blogspot.com/2018/12/smartly-select-and-mutate-data-frame_9.html#modify)
  + [Remove](https://r-some-blog.blogspot.com/2018/12/smartly-select-and-mutate-data-frame_9.html#remove)
* [Benchmark](https://r-some-blog.blogspot.com/2018/12/smartly-select-and-mutate-data-frame_9.html#benchmark)
* [Summary](https://r-some-blog.blogspot.com/2018/12/smartly-select-and-mutate-data-frame_9.html#summary)

**Motivation**

The [dplyr](https://cran.r-project.org/package=dplyr) functions select and mutate nowadays are commonly applied to perform data.frame column operations, frequently combined with [magrittr](https://cran.r-project.org/package=magrittr)s forward %>% pipe. While working well interactively, however, these methods often would require additional checking if used in “serious” code, for example, to catch column name clashes.

In principle, the [container](https://cran.r-project.org/web/packages/container/index.html) package provides a dict-class (resembling [Python](https://www.python.org/)s dict type), which allows to cover these issues more easily. In its very recent update, the [container](https://cran.r-project.org/web/packages/container/index.html) package for this reason gained an S3 method interface plus functions to convert back and forth between dict and data.frame. This can be used to extend the set of data.frame column operations and in this post I will show how and when they can serve as a useful alternative to mutate and select.

To keep matters simple, we use a tiny data table.

data <- data.frame(x = c(0.2, 0.5), y = letters[1:2])  
data  
## x y  
## 1 0.2 a  
## 2 0.5 b

**Column operations**

**Add**

Let’s add a column using mutate.

library(dplyr)  
n <- nrow(data)  
  
data %>%   
 mutate(ID = 1:n)  
## x y ID  
## 1 0.2 a 1  
## 2 0.5 b 2

For someone not familar with the [tidyverse](https://www.tidyverse.org/), this code block might read somewhat odd as the column is added and *not* mutated. To add a column using dict simply use add.

library(container)  
data %>%   
 as.dict() %>%   
 add("ID", 1:n) %>%  
 as.data.frame()  
## x y ID  
## 1 0.2 a 1  
## 2 0.5 b 2

The intended add-operation is stated more clearly, but on the downside we also had to add some overhead. Of course, since this has to be done only at the beginning and at the end of the pipe, it will be less of an issue if multiple dict-operations are performed in between. Next, instead of ID, let’s add another numeric column y, which happens to “name-clash” with the already existing column.

data %>%   
 mutate(y = rnorm(n))  
## x y  
## 1 0.2 1.5295729  
## 2 0.5 0.4821652

Ooops – we have accidently overwritten the initial y-column. While this was easy to see here, it may not if the data.frame has a lot of columns or if column names are created automatically as part of some script. To catch this, usually some overhead is required, too.

if ("y" %in% colnames(data)) {  
 stop("column y already exists")  
} else {  
 data %>%   
 mutate(y = rnorm(n))  
}  
## Error in eval(expr, envir, enclos): column y already exists

Let’s see the dict-operation in comparison.

data %>%   
 as.dict() %>%   
 add("y", rnorm(n)) %>%  
 as.data.frame()  
## Error in x$add(key, value): key 'y' already in Dict

The name clash is catched by default and the overhead does not look so silly anymore. As a bonus, the error message still provides information about the originally *intended* add-operation.

**Modify**

If the intend was indeed to overwrite the value, the dict-function setval can be used.

data %>%   
 as.dict() %>%   
 setval("y", rnorm(n)) %>%  
 as.data.frame()  
## x y  
## 1 0.2 0.5315501  
## 2 0.5 -0.1573612

As we saw above, if a column does not exist, mutate silently creates it for you. If this is not what you want, which means, you want to make sure something is overwritten, again, a workaround is needed.

if ("ID" %in% colnames(data)) {  
 data %>%   
 mutate(ID = 1:n)   
} else {  
 stop("column ID not in data.frame")  
}  
## Error in eval(expr, envir, enclos): column ID not in data.frame

Once again, the workaround is already “built-in” in the dict-framework.

data %>%   
 as.dict() %>%   
 setval("ID", 1:n) %>%  
 as.data.frame()  
## Error in x$set(key, value, add): key 'ID' not in Dict

After all, the intend of the mutate function actually would be something like: *overwrite a column, or, create it if it does not exist*. If desired, this behaviour can be expressed within the dict-framework as well.

data %>%   
 as.dict() %>%   
 setval("ID", 1:n, add=TRUE) %>%  
 as.data.frame()  
## x y ID  
## 1 0.2 a 1  
## 2 0.5 b 2

**Remove**

A common [tidyverse](https://www.tidyverse.org/) approach to remove a column is based on the select function. One corresponding dict-function is remove.

data %>%   
 select(-"y")  
## x  
## 1 0.2  
## 2 0.5  
  
data %>%   
 as.dict() %>%   
 remove("y") %>%   
 as.data.frame()  
## x  
## 1 0.2  
## 2 0.5

Let’s see what happens if the column does not exist in the first place.

data %>%  
 select(-"ID")  
## Error: Unknown column `ID`  
  
data %>%   
 as.dict() %>%   
 remove("ID") %>%   
 as.data.frame()  
## Error in x$remove(key): key 'ID' not in Dict

Again, we obtain a slightly more informative error message with dict. Assume we want the column to be removed if it exist but otherwise silently ignore the command, for example:

if ("ID" %in% colnames(data)) {  
 data %>%  
 select(-"ID")   
}

You may have expected this by now – the dict-framework provides a straigh-forward solution, namely, the discard function:

data %>%   
 as.dict() %>%   
 discard("ID") %>%   
 discard("y") %>%  
 add("z", c("Hello", "World")) %>%  
 as.data.frame()  
## x z  
## 1 0.2 Hello  
## 2 0.5 World

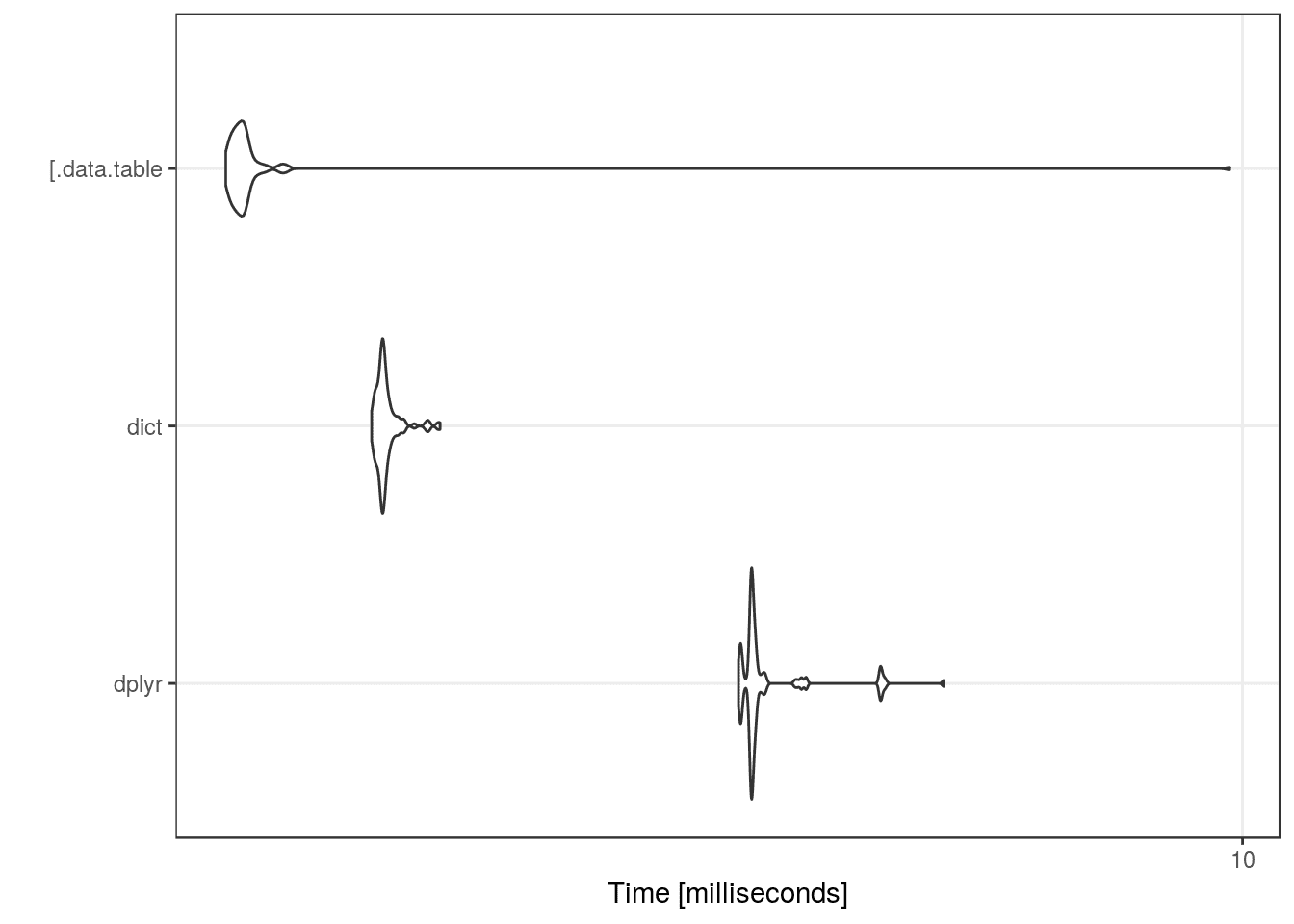
**Benchmark**

The required additional code lines are limited but what about the computational overhead? To examine this, we benchmark some column operations using the famous ‘iris’ data set. As a hallmark reference we will also bring the [data.table](https://cran.r-project.org/package=data.table) framework to the competition.

set.seed(123)  
library(microbenchmark)  
library(ggplot2)  
library(data.table)  
data <- iris  
n <- nrow(data)  
head(iris)  
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## 1 5.1 3.5 1.4 0.2 setosa  
## 2 4.9 3.0 1.4 0.2 setosa  
## 3 4.7 3.2 1.3 0.2 setosa  
## 4 4.6 3.1 1.5 0.2 setosa  
## 5 5.0 3.6 1.4 0.2 setosa  
## 6 5.4 3.9 1.7 0.4 setosa

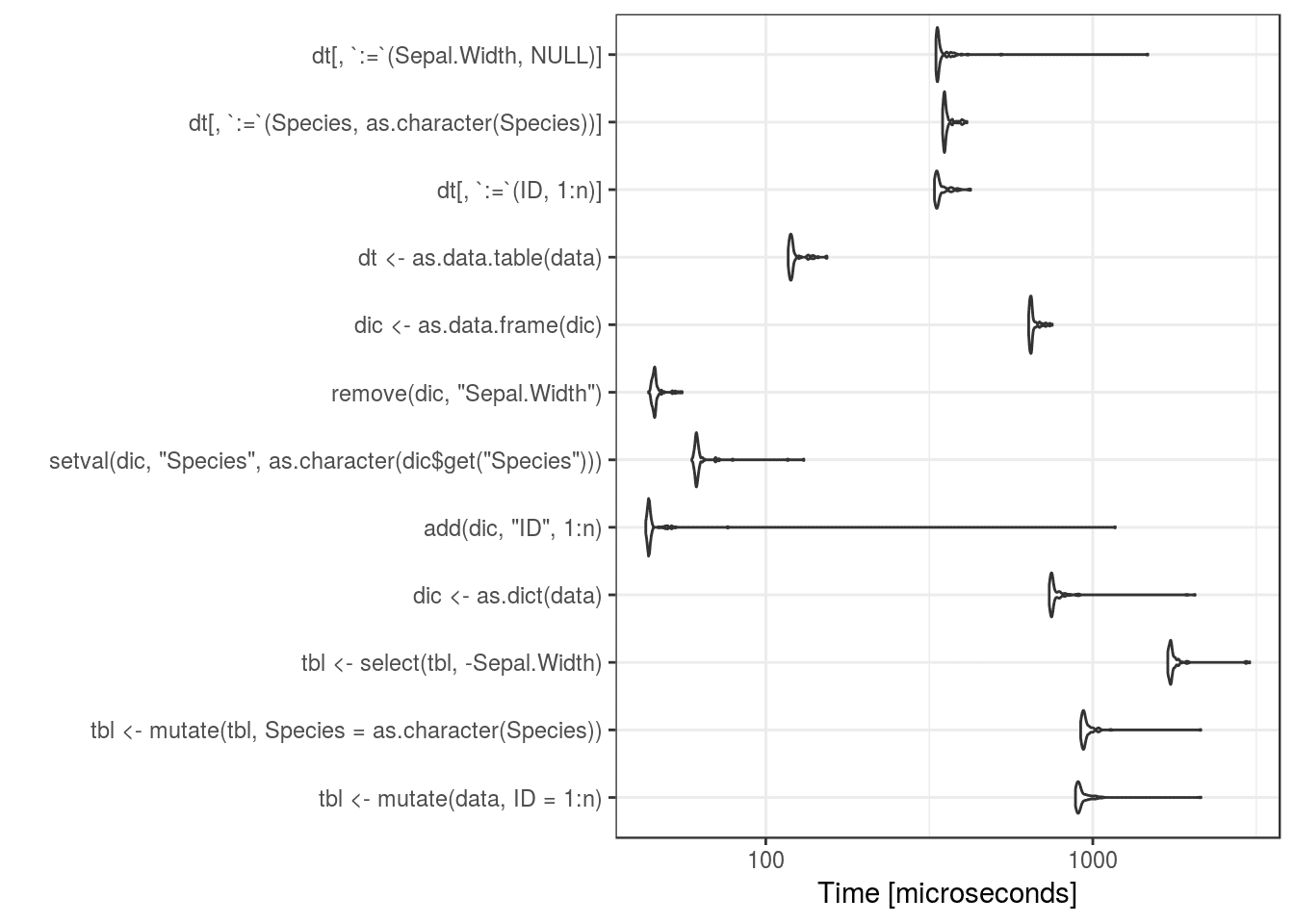
For the benchmark, we add one, transform one and finally delete one column.

bm <- microbenchmark(control = list(order="inorder"), times = 100,  
 dplyr = data %>%   
 mutate(ID = 1:n) %>%  
 mutate(Species = as.character(Species)) %>%  
 select(-Sepal.Width),  
   
 dict = data %>%  
 as.dict() %>%  
 add("ID", 1:n) %>%  
 setval(., "Species", as.character(.$get("Species"))) %>%  
 remove("Sepal.Width") %>%  
 as.data.frame(),  
   
 `[.data.table` = data %>%   
 as.data.table() %>%  
 .[, ID := 1:n] %>%  
 .[, Species := as.character(Species)] %>%  
 .[, Sepal.Width := NULL]  
)  
autoplot(bm) + theme\_bw()

[](https://i0.wp.com/1.bp.blogspot.com/-54WhxkOCNSg/XAlvgZwUcwI/AAAAAAAAAtI/QsqeLbzxut49sjuAljo_NKnnCwGm1oijgCPcBGAYYCw/s1600/benchmark1-1.png?ssl=1)

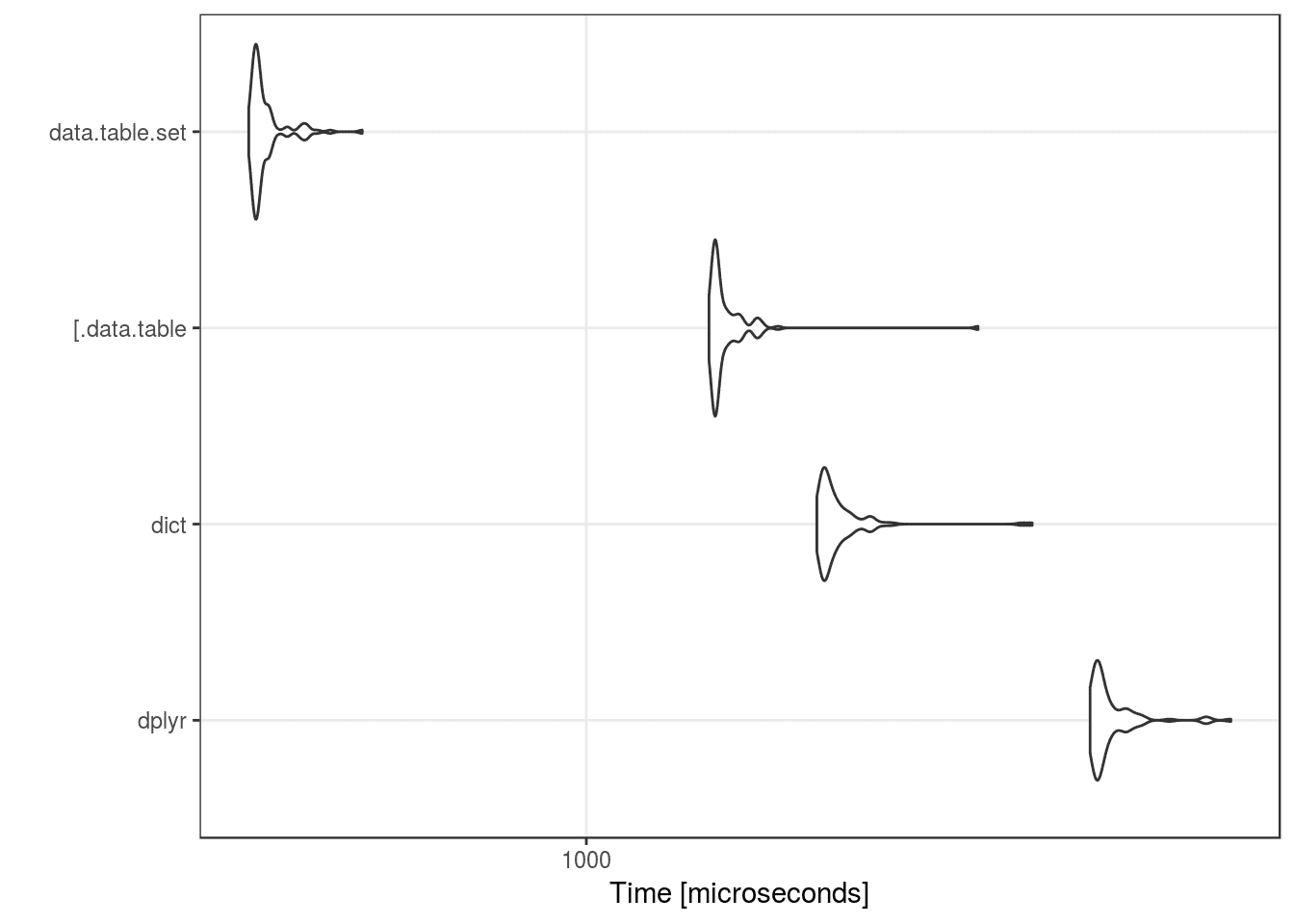
Somewhat surprisingly maybe, the dict-implementation is closer to the [data.table](https://cran.r-project.org/package=data.table) than to the [dplyr](https://cran.r-project.org/package=dplyr) performance. Let’s examine each operation in more detail.

bm <- microbenchmark(control = list(order="inorder"), times = 100,  
 tbl <- mutate(data, ID = 1:n),  
 tbl <- mutate(tbl, Species = as.character(Species)),  
 tbl <- select(tbl, -Sepal.Width),  
   
 dic <- as.dict(data),  
 add(dic, "ID", 1:n),  
 setval(dic, "Species", as.character(dic$get("Species"))),  
 remove(dic, "Sepal.Width"),  
 dic <- as.data.frame(dic),  
   
 dt <- as.data.table(data),  
 dt[, ID := 1:n],  
 dt[, Species := as.character(Species)],  
 dt[, Sepal.Width := NULL]  
)  
autoplot(bm) + theme\_bw()

[](https://i2.wp.com/2.bp.blogspot.com/-c2nnWFqsAlo/XAlusRLz8bI/AAAAAAAAAs4/tholDFE7pb0YyZzUknnT0d-a8tKmrpQGACPcBGAYYCw/s1600/benchmark2-1.png?ssl=1)

Apparently, the mutate and select operations are the slowest in comparison, I think, because both the dict and [data.table](https://cran.r-project.org/package=data.table) approach work by reference while probably some copying is done in the [dplyr](https://cran.r-project.org/package=dplyr) pipe. We also see that the dict-approach spends most of the computation time for the transformation back and forth between a dict and a data.frame while the actual column operations seem very efficient, even more efficient than that of [data.table](https://cran.r-project.org/package=data.table). This certainly came as a surprise to me, as the focus when developing the [container](https://cran.r-project.org/web/packages/container/index.html) package has never been on speed but rather on providing a concise data structure. Internally the dict simply consists of a named list, so I guess this speaks for the efficiency of base R list operations. Having said that, I found that the [data.table](https://cran.r-project.org/package=data.table) code can be further improved by avoiding the overhead of the [.data.table operator and instead use the built-in set function:

bm <- microbenchmark(control = list(order="inorder"), times = 100,  
 dplyr = data %>%   
 mutate(ID = 1:n) %>%  
 mutate(Species = as.character(Species)) %>%  
 select(-Sepal.Width),  
   
 dict = data %>%  
 as.dict() %>%  
 add("ID", 1:n) %>%  
 setval(., "Species", as.character(.$get("Species"))) %>%  
 remove("Sepal.Width") %>%  
 as.data.frame(),  
   
 `[.data.table` = data %>%   
 as.data.table() %>%  
 .[, ID := 1:n] %>%  
 .[, Species := as.character(Species)] %>%  
 .[, Sepal.Width := NULL],  
   
 data.table.set = data %>%   
 as.data.table() %>%  
 set(., j= "ID", value = 1:n) %>%  
 set(., j = "Species", value = as.character(.[["Species"]])) %>%  
 set(., j = "Sepal.Width", value = NULL)  
)  
autoplot(bm) + theme\_bw()

[](https://i2.wp.com/2.bp.blogspot.com/-0hHhodE858c/XAluc7ZoctI/AAAAAAAAAsw/tkYk09xyTNYYsr8bd6ey9H434sYpNPIswCPcBGAYYCw/s1600/benchmark3-1.png?ssl=1)

This puts things back into perspective, I guess It might also be interesting to know, how much of the computation time is spent on the non-standard evaluation part of the dplyr and [.data.table implementation, but that’s probably a topic on its own.

**Summary**

Accidently overwriting existing data columns leads to nasty bugs. The presented workflow allows to increase both reliability and precision of standard data frame column manipulation at very little cost. The intended column operations can be expressed more clearly and, in case of failures, informative error messages are provided by default. As a result, the dict-framework may serve as a useful supplement to “interactive piping”.