When should I use setDT() instead of data.table() to create a data.table?

Question Link

I am having difficulty grasping the essence of the setDT() function. As I read code on SO, I frequently come across the use of setDT() to create a data.table. Of course the use of data.table() is ubiquitous. I feel like I solidly comprehend the nature of data.table() yet the relevance of setDT() eludes me. ?setDT tells me this:

> "setDT converts lists (both named and unnamed) and data.frames to data.tables by reference."

as well as:

> "In data.table parlance, all set* functions change their input by reference. That is, no copy is made at all, other than temporary working memory, which is as large as one column."

So this makes me think I should only use setDT() to make a data.table, right? Is setDT() simply a list to data.table converter?

```
library(data.table)

a <- letters[c(19,20,1,3,11,15,22,5,18,6,12,15,23)]
b <- seq(1,41,pi)
ab <- data.frame(a,b)
d <- data.table(ab)
e <- setDT(ab)

str(d)
```

```
## Classes 'data.table' and 'data.frame':   13 obs. of  2 variables:
##  $ a: Factor w/ 12 levels "a","c","e","f",..: 9 10 1 2 5 7 11 3 8 4 ...
##  $ b: num  1 4.14 7.28 10.42 13.57 ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

```
str(e)
```

```
## Classes 'data.table' and 'data.frame':   13 obs. of  2 variables:
##  $ a: Factor w/ 12 levels "a","c","e","f",..: 9 10 1 2 5 7 11 3 8 4 ...
##  $ b: num  1 4.14 7.28 10.42 13.57 ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

Seemingly no difference in this instance. In another instance the difference is evident:

```
## Classes 'data.table' and 'data.frame':   2 obs. of  1 variable:
##  $ ba:List of 2
##   ..$ : chr  "s" "t" "a" "c" ...
##   ..$ : num  1 4.14 7.28 10.42 13.57 ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

```
## Classes 'data.table' and 'data.frame':   13 obs. of  2 variables:
##  $ V1: chr  "s" "t" "a" "c" ...
##  $ V2: num  1 4.14 7.28 10.42 13.57 ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

When should I use setDT()? What makes setDT() relevant? Why not just make the original data.table() function capable of doing what setDT() is able to do?

Answer

Update: > "@Roland makes some good points in the comments section, and the post is better for them. While I originally focused on memory overflow issues, he pointed out that even if this doesn't happen, memory management of various copies takes substantial time, which is a more common everyday concern. Examples of both issues have now been added as well.""

I like this question on stackoverflow because I think it is really about avoiding stack overflow in R when dealing with larger data sets. ðŸ˜Š Those who are unfamiliar with `data.table` family of `set` operations may benefit from this discussion!

One should use `setDT()` when working with larger data sets that take up a considerable amount of RAM because the operation will modify each object in place, conserving memory. For data that is a very small percentage of RAM, using data.tableâ€™s copy-and-modify is fine.

The creation of the `setDT` function was actually inspired by the following thread on stack overflow, which is about working with a large data set (several GB's). You will see Matt Dowle chime in an suggest the 'setDT' name.

https://stackoverflow.com/questions/20345022/convert-a-data-frame-to-a-data-table-without-copy

A bit more depth:

With R, data is stored in memory. This speeds things up considerably because RAM is much faster to access than storage devices. However, a problem can arise when oneâ€™s data set is a large portion of RAM. Why? Because base R has a tendency to make copies of each `data.frame` when some operations are applied to them. This has improved after version 3.1, but addressing that is beyond the scope of this post. If one is pulling multiple `data.frame`s or `list`s into one `data.frame` or `data.table`, your memory usage will expand rather quickly because at some point during the operation, multiple copies of your data exist in RAM. If the data set is big enough, you may run out of memory when all the copies are produced, and your stack will overflow. See example of this below. We get an error, and original memory address and class of object does not change.

```
> N <- 1e8
> P <- 1e2
> data <- as.data.frame(rep(data.frame(rnorm(N)), P))
>
> pryr::object_size(data)
800 MB
>
> tracemem(data)
[1] "<0000000006D2DF18>"
>
> data <- data.table(data)
Error: cannot allocate vector of size 762.9 Mb
>
> tracemem(data)
[1] "<0000000006D2DF18>"
> class(data)
[1] "data.frame"
>
```

The ability to just modify the object in place without copying is a big deal. That is what `setDT` does when it takes a `list` or `data.frame` and returns a `data.table`. The same example as above using `setDT`, now works fine and without error. Both class and memory address change, and no copies take place.

```
> tracemem(data)
[1] "<0000000006D2DF18>"
```

```
> class(data)
[1] "data.frame"
>
> setDT(data)
>
> tracemem(data)
[1] "<0000000006A8C758>"
> class(data)
[1] "data.table" "data.frame"
```

@Roland points out that for most people, the bigger concern is speed, which suffers as a side effect of such intensive use of memory management. Here is an example with smaller data that does not crash the cpu, and illustrates how much faster `setDT` is for this job. Notice the results of 'tracemem' in the wake of `data <- data.table(data)`, making copies of `data`. Contrast that with `setDT(data)` which doesn't print a single copy. We have to then call `tracemem(data)` to see the new memory address.

```
> N <- 1e5
> P <- 1e2
> data <- as.data.frame(rep(data.frame(rnorm(N)), P))
> pryr::object_size(data)
808 kB

> # data.table method
> tracemem(data)
[1] "<0000000019098438>"
> data <- data.table(data)
tracemem[0x0000000019098438 -> 0x0000000007aad7d8]: data.table
tracemem[0x0000000007aad7d8 -> 0x0000000007c518b8]: copy as.data.table.data.frame as.data.table data
tracemem[0x0000000007aad7d8 -> 0x0000000018e454c8]: as.list.data.frame as.list vapply copy as.data.
> class(data)
[1] "data.table" "data.frame"
>
> # setDT method
> # back to data.frame
> data <- as.data.frame(data)
> class(data)
[1] "data.frame"
> tracemem(data)
[1] "<000000000125BE1A0>"
> setDT(data)
> tracemem(data)
[1] "<000000000125C2840>"
> class(data)
[1] "data.table" "data.frame"
>
```

How does this impact timing? As we can see, `setDT` is much faster for it.

```
> # timing example
> data <- as.data.frame(rep(data.frame(rnorm(N)), P))
> microbenchmark(setDT(data), data <- data.table(data))
Unit: microseconds
                      expr       min         lq        mean    median            max neval        uq
               setDT(data)    49.948    55.7635    69.66017    73.553        100.238   100    79.198
 data <- data.table(data) 54594.289 61238.8830 81545.64432 64179.131     611632.427   100 68647.917
```

Set functions can be used in many areas, not just when converting objects to a data.tables. You can find more information on the reference semantics and how to apply them elsewhere by calling the vignette on the subject.

```
    library(data.table)
    vignette("datatable-reference-semantics")
```

This is a great question and those thinking of using R with larger data sets or who just want to speed up data manipulation actives, can benefit from being familiar with the significant performance improvements of `data.table` reference semantics.