1 Introduction

A lot of people may have heard about Data Table as being a package that allows users to “speed up” the run time of scripts. While true, what is more amazing than the speed up in processing time is the speed up in development time. The reason for this gain is because Data Table lets users write less code that is not only more elegant, but also easier to maintain.

The package itself has been out for several years, and has undergone significant development to make it easier to use and more powerful. The core development team is very well staffed and I believe the package will be maintained for quite some time, so it is definitely worth learning.

This presentation will cover everything from basic syntax to some of the more intricate aspects of data.table. A quick overview of topics covered:

<table>
<thead>
<tr>
<th>Topics Covered</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expressions in data.table</strong></td>
<td>Scoping</td>
</tr>
<tr>
<td></td>
<td>Assignment</td>
</tr>
<tr>
<td></td>
<td>Grouping</td>
</tr>
<tr>
<td><strong>Built in Constants</strong></td>
<td>Aggregation Constants</td>
</tr>
<tr>
<td></td>
<td>Apply Constants</td>
</tr>
<tr>
<td><strong>Quick Speedups</strong></td>
<td>fread</td>
</tr>
<tr>
<td></td>
<td>rbindlist</td>
</tr>
<tr>
<td></td>
<td>set</td>
</tr>
<tr>
<td></td>
<td>chmatch</td>
</tr>
<tr>
<td><strong>Reshaping data.table</strong></td>
<td>reshape2</td>
</tr>
<tr>
<td><strong>Viewing data.table’s in memory</strong></td>
<td>tables</td>
</tr>
<tr>
<td><strong>Gotchyas’s</strong></td>
<td>Unique and Duplicated</td>
</tr>
<tr>
<td></td>
<td>Copy</td>
</tr>
</tbody>
</table>

1 Another solid introduction to the package comes in the form of a F.A.Q.
2 Motivating Data.Table

2.1 Elegance

On even the most basic level, data.table has much more elegant syntax than data.frame. Assuming a large data set with an unknown column ordering:

```r
### Changing names on a data.frame
names(data)[which(names(data) == "old.var")] <- "new.var"

### Changing names on a data.table
require(data.table)
setnames(data, "old.var", "new.var")
```

Of course, the real beauty is under the hood. For those who are interested (ignore the trailing output):

```r
data <- data.frame(x = 1:10)

### 2 copies of "data" to change 1 name.
tracemem(data)

## [1] "<0x000000000646c1c0>"

names(data) <- 'y'

## tracemem[0x000000000646c1c0 -> 0x000000000d21dd10]: eval eval withVisible withCallingHandlers doTryCatch tryCatchOne ... evaluate in_dir block_exec call_block process_group.block process_group withCallingHandlers process_file knit

### No copies.
setattr(data, 'names', value = 'z')
data <- data.table(data)

## tracemem[0x000000000d21dc20 -> 0x000000000d1431c8]: data.table eval eval withVisible withCallingHandlers doTryCatch tryCatchOne ... evaluate in_dir block_exec call_block process_group.block process_group withCallingHandlers process_file knit

setnames(data, 'new.var')
```

This is not so much of a problem with data.frame as it is with the “<-” operator and R itself, which tends to create copies of objects. More on this in the [conclusion](#).
2.2 A Practical Example

It may take a huge stretch of the imagination, but figure that in some crazy universe, you are examining a *messy* data set. The data has come to you with *really* missing observations. What do I mean when I say *really* missing? Imagine a data set that looks like this:

```
raw

## id  date   val
## 1: a 2012-01-04 158
## 2: a 2012-01-10 153
## 3: a 2012-01-12 154
## 4: a 2012-01-13 153
## 5: a 2012-01-15 162
## ---
## 4754: z 2012-12-23 163
## 4755: z 2012-12-25 160
## 4756: z 2012-12-26 152
## 4757: z 2012-12-27 155
## 4758: z 2012-12-29 156
```

At the very least, you want to have a data set which has at least something filled in for every observation, even if it’s an “NA”. What you might consider doing is creating a template of data that has at least *dates* filled in for every observation, for every individual in your data set.

Suppose now that after discussing the situation with your project manager, you decide that the data is more or less complete, and you can take a simple approach to remedy the issue:

- Merge crummy data into the template to get a “balanced” panel
- Fill in missing data with the “most recent” values

This method is known as “Last Observation Carried Forward”. This entire process can also easily be accomplished with *data.table*, using what’s known as a “rolling join”.

3
### Set keys and create a template...

```r
setkey(raw, id, date)
ids <- letters[1:26]
dates <- seq.Date(from = as.Date("2012-01-01"),
                  to = as.Date("2012-12-31"),
                  by = "1 day")
```

### ...of all pairwise combinations.

```r
template <- CJ(ids, dates)
```

### Using a join.

```r
raw[template]
```

### After LOCF

```r
raw[template, roll = T]
```
data <- raw[template, roll = T]

What happened here is that since our data.table was keyed and we fed in a data.table for the “i” argument, the “[“ operator acted as a “join” operator, such that the data are merged. The “roll” argument is what allows us the roll the “last” observation forward. It’s possible to roll observations forward and backwards without limit, as well as for specified cutoffs.

Notice that each time we perform a data.table operation, there is a lot happening behind the hood. Let’s take a second to consider the way “keys” work in data.tables, and then we’ll revisit the roll functionality.

3 Keys

In the preceding example, we used the “set key” function, which was necessary before we could perform the rolling join. What exactly did this do?

**For the Nerd** When you set a key, “data table” goes out and stores the data such that the “groups” appear contiguously in the computer’s RAM. If ever past that point a function is called that can group operations by the first n columns of the data.table’s keys, then these are used such that the memory is copied in bulk for faster access. Anytime we use a “set” function, we are modifying an attribute by reference, in this case the “key” of a table is an attribute, and nothing is copied in memory.

**For the Rest** Basically, if you set a key, you are changing how your data is stored in memory, such that certain operations are much faster. Since our data is stored more efficiently in memory after setting a key, it can be helpful to set it before certain tasks, but it is also important to note that **setting a key is quite computationally expensive itself.**

Getting back to the preceding example, it is certainly the case that Joins can be very handy for this exact type of scenario. However, in general I tend to avoid them otherwise, as the syntax is nuanced. It may be appealing to think about a join like a merge, but this is not really the case. For example, when performing a join \( x[i] \), where \( x \) is a keyed data.table and \( i \) is another data.table that is being joined to \( x \), what really happens is that each of the columns of \( i \) are joined in order to the order of the keyed columns of \( x \). This is unlike a typical data.frame or even data.table merge. I will leave the rest of the details to the [documentation](#).
4 Expressions in Data.Table

So now that we know “data.table” can do cool stuff, let’s learn a little bit more about how it works and how it’s different from “data.frame”. The main things to understand when coming from data.frames, is that the rules for what goes inside the “[” operators is different.

4.1 “[” as an Operator

“[” is actually a function, and is specifically a “extract” operator. This is true for base R, as well as data.table. For example, to extract the second element from some vector, we could write:

```
```[runif(10), 2]
```

```
## [1] 0.0486
```

It happens to be that the number of arguments that the “[” function takes depends on what kind of object it is given. This is the idea behind a method.

```
df <- data.frame(data[, list(id, date)])
```

```
`[`(df, 1, 2)
```

```
## [1] "2012-01-01"
```

It also happens to be that using the above syntax, had we only fed in one integer as an argument, the operator would have pulled out the nth column. This makes sense if you think about data.table as being a list of columns. Whereas data.frame typically expects two parameters inside the subset operator, a condition to subset rows and a condition to subset columns, for a data.table the “[” operator is really the gateway into the package contents.

Revisiting Roll There are many different expressions, arguments, and parameters which can be fed into a data.table expression. In our initial example, we used the “[” operator in conjunction with a data.table as the first argument, “i”, in order to ultimately access the “roll = T” option. Another way to get across the same point: the “roll” argument can only be specified if the “i” argument is a data.table. If a user had input a condition to subset the rows of the data.table, the “roll” argument would not be relevant.
4.2 A Note on Scoping

As we move through this presentation, notice that there is not one instance in which I refer to a variable using the “$” extract operator. This is because for any expression placed within a call to data.table, data.table evaluates the expression within the scope of the object. Therefore, there is no need to tell R to go out and look for “data$var”, because it already will recognize “var” as being an object within the scope of “data”. It’s as though once you’re in a data.table, all the variables are readily accessible without obfuscated syntax.

**Syntax**  It’s worth noting that reverting to using the “$” operator with data.table will, in most cases, degenerate to code which is not only obfuscated, but also slower to execute.

4.3 Assigning variables

There are several ways to assign variables within a data.table. An assignment is an example of an expression.

The following expressions all assign variables exactly as you would expect. The only visible difference is that there is an “:=” assignment operator instead of “<-”.

```r
### Indicator for positivity.
data[, pos := val > 0]

### Assign multiple variables.
data[, `:=`(month = month(date),
            year = year(date))]

### Assign multiple variables a different way.
data[, c('day', 'rnd') := list(gsub('.*?([0-9]+)$', '\\1', date),
                                    runif(nrow(data)))]
```
### View newly assigned columns.

data

```r
# View newly assigned columns.
data

```n

```r
## id date val pos month year day rnd
## 1: a 2012-01-01 NA NA 1 2012 01 0.50964
## 2: a 2012-01-02 NA NA 1 2012 02 0.12964
## 3: a 2012-01-03 NA NA 1 2012 03 0.49467
## 4: a 2012-01-04 158 TRUE 1 2012 04 0.43110
## 5: a 2012-01-05 158 TRUE 1 2012 05 0.97193
## ---
## 9512: z 2012-12-27 155 TRUE 12 2012 27 0.06787
## 9513: z 2012-12-28 155 TRUE 12 2012 28 0.35365
## 9514: z 2012-12-29 156 TRUE 12 2012 29 0.50147
## 9515: z 2012-12-30 156 TRUE 12 2012 30 0.76311
## 9516: z 2012-12-31 156 TRUE 12 2012 31 0.26177
```

### Subset (remove) columns

```r
rmv <- grep("pos|year|day|rnd", names(data))
data[, `:=`(rmv, NULL), with = F]
```

```r
## id date val month
## 1: a 2012-01-01 NA 1
## 2: a 2012-01-02 NA 1
## 3: a 2012-01-03 NA 1
## 4: a 2012-01-04 158 1
## 5: a 2012-01-05 158 1
## ---
## 9512: z 2012-12-27 155 12
## 9513: z 2012-12-28 155 12
## 9514: z 2012-12-29 156 12
## 9515: z 2012-12-30 156 12
## 9516: z 2012-12-31 156 12
```

### Alternative way to subset (keep) columns

```r
data <- data[, list(id, date, val, month)]
```

I personally feel strongly that of the two methods to assign multiple variables, the first is much better, because it explicitly pairs the variable being assigned to it's assignment. With regard to subsetting data, I generally prefer to reassign the data to a “new” object, according to the latter method.
4.4 Grouping

One of my favorite aspects of data.table is how easy it is to perform operations by group. The syntax is self-explanatory.

```r
data[, mean(val, na.rm = T), by = month(date)]
```

<table>
<thead>
<tr>
<th>month</th>
<th>V1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>161.1</td>
</tr>
<tr>
<td>2:</td>
<td>160.3</td>
</tr>
<tr>
<td>3:</td>
<td>161.3</td>
</tr>
<tr>
<td>4:</td>
<td>161.0</td>
</tr>
<tr>
<td>5:</td>
<td>160.8</td>
</tr>
<tr>
<td>6:</td>
<td>160.6</td>
</tr>
<tr>
<td>7:</td>
<td>160.9</td>
</tr>
<tr>
<td>8:</td>
<td>161.2</td>
</tr>
<tr>
<td>9:</td>
<td>161.0</td>
</tr>
<tr>
<td>10:</td>
<td>161.0</td>
</tr>
<tr>
<td>11:</td>
<td>160.8</td>
</tr>
<tr>
<td>12:</td>
<td>160.6</td>
</tr>
</tbody>
</table>

The output from the above command is a new data.table, which contains the output from the expression alongside the "by" variables, and this can easily be assigned to another object for future use, or simply viewed as a summary table for prototyping. Take a moment to realize that this is another example of an expression being fed into data.table.

In data.table, anything is possible. Whether we create a new data.table or assign new variables to an existing data.table, it’s possible to create one or more variables and/or group by one or more variables.

```r
## Create one new variable, using by, existing data.
data[, output := mean(val, na.rm = T), by = date]
```

<table>
<thead>
<tr>
<th>id</th>
<th>date</th>
<th>val</th>
<th>month</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>a 2012-01-01</td>
<td>NA</td>
<td>1</td>
<td>159.4</td>
</tr>
<tr>
<td>2:</td>
<td>a 2012-01-02</td>
<td>NA</td>
<td>1</td>
<td>161.5</td>
</tr>
<tr>
<td>3:</td>
<td>a 2012-01-03</td>
<td>NA</td>
<td>1</td>
<td>163.6</td>
</tr>
<tr>
<td>4:</td>
<td>a 2012-01-04</td>
<td>158</td>
<td>1</td>
<td>161.5</td>
</tr>
<tr>
<td>5:</td>
<td>a 2012-01-05</td>
<td>158</td>
<td>1</td>
<td>159.9</td>
</tr>
<tr>
<td>---</td>
<td>---------</td>
<td>-----</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>9512:</td>
<td>z 2012-12-27</td>
<td>155</td>
<td>12</td>
<td>160.4</td>
</tr>
</tbody>
</table>
### Create two new variables, using by, existing data.

```r
data[, `:=`(sd = sd(val, na.rm = T),
        var = var(val, na.rm = T)), by = id]
```

### Creating a new data.table, with one named output column.

```r
summary <- data[, list(SUM = sum(val)), by = id]
```

### New data.table, mult. output columns, by mult. vars.

```r
summary <- data[, list(SUM = sum(val),
                    MEAN = mean(val)), by = list(id, month)]
```
These examples help to illustrate not only how flexible the syntax is, but also that the syntax changes from one example to another are relatively minor and simple to follow.

**If you get confused** Just remember that unless you explicitly see the “:=” operator being used, the existing data.table is not being assigned to, and another data.table object is likely being created.

**Keyby** Don’t be fooled by what the ”keyby” argument actually does, although it can be handy:

```r
DT <- summary[, mean(MEAN), keyby = id]
DT2 <- summary[, mean(MEAN), by = id]
setkey(DT2, id)
identical(DT, DT2)
## [1] TRUE
```

## Built in Constants

There are several useful built in constants as part of the data.table package. Note that these constants are to be fed into the *expression* part of the data.table syntax, that is, after the “comma”. I have labeled the following two subjects in a way that I hope illustrates that two types of constants are really useful for two different types of problems, but that both are also “constants” in the sense that they are read-only in data.table, and cannot be overwritten.

The “aggregation constants” are great for building summary tables or summary statistics, and the “apply constants” are great for applying functions to subsets of Data, such as a set of columns.

### 5.1 Aggregation Constants - .N, .I, .GRP

- .\textit{N} is a built in constant, and specifically it is a length one integer describing the number of observations for each group.
- Another useful constant is .\textit{I}, which is an integer vector of length .\textit{N} which holds the row numbers for group \( i \).
- Last but not least, the .\textit{GRP} constant provides a simple integer counter for each grouping.
data[, list(group = .GRP, rows = .I), by = id]

## id group rows
## 1: a 1 1
## 2: a 1 2
## 3: a 1 3
## 4: a 1 4
## 5: a 1 5
## ---
## 9512: z 26 9512
## 9513: z 26 9513
## 9514: z 26 9514
## 9515: z 26 9515
## 9516: z 26 9516

**Practical Example, Cont.** Going back to the motivation for our “practical example”, perhaps you were interested to know how bad the “really” missing data problem was. Imagine that you want to how many observations are in your data for each identity

raw[, .N, by = id][order(-N)][c(1:3, 26)]

## id N
## 1: d 195
## 2: q 195
## 3: l 192
## 4: w 165

Here we have displayed the top 3 id’s with the most observations and the id with the least observations.

**A Note on Syntax** Notice that to achieve this result, multiple calls to different data.table were performed. The first call is to our “raw” object, but the second call is to the summary table that was produced after using the “.N” constant. Note that because we don’t see a “:=” assignment operator, we can rest assured that the expression spits out a new data.table. On the resulting data.table, we ask for a row operation to be performed, where the rows of the summary table are reordered from greatest to least. Then, on the resulting data.table, we ask for a sequence of rows. This trick can be
very handy for quick one liners when prototyping, or for simply summary tables, but in general, notice that the expression is actually quite dense to read. Therefore, as a general rule, I believe the syntax is not good practice for clean code that you intend to actually live in your program.

One other point worth noting is that, similar to data.frame, any ordering arguments are placed inside the position you might expect row operations to occur, that is to say “before the comma”. You may have also noticed that although the position looks correct, there is no closing comma; notice how and where both the order function and the integer vector one through five are fed in. Data.table doesn’t require a closing comma if only an “i” argument is supplied and in general it is my preference to leave commas out as they create space and ambiguity whether the author intended to have an expression.

5.2 Apply Constants - .SD, .SDcols

There are several tools that allow users to manipulate subsets of the data.table at a time. If you ever need to change multiple column classes.

```r
nvars <- grep('SUM|MEAN', names(summary))
summary[, nvars := lapply(.SD, log), .SDcols = nvars, with = F][1:6]
```

<table>
<thead>
<tr>
<th>id</th>
<th>month</th>
<th>SUM</th>
<th>MEAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>a</td>
<td>1</td>
<td>NA</td>
</tr>
<tr>
<td>2:</td>
<td>a</td>
<td>2</td>
<td>8.432</td>
</tr>
<tr>
<td>3:</td>
<td>a</td>
<td>3</td>
<td>8.536</td>
</tr>
<tr>
<td>4:</td>
<td>a</td>
<td>4</td>
<td>8.480</td>
</tr>
<tr>
<td>5:</td>
<td>a</td>
<td>5</td>
<td>8.514</td>
</tr>
<tr>
<td>6:</td>
<td>a</td>
<td>6</td>
<td>8.484</td>
</tr>
</tbody>
</table>

I find that using .SD and .SDcols can be useful, but sometimes the syntax can be confusing. We will see in a moment how there is another alternative to assigning over groups of columns.

---

2Recall, this can be a data.table, or an argument to subset rows.
6 Quick Speed Ups

Data.table is filled with little tricks that can save you time when you next find yourself playing with big data.

fread  Read in spreadsheet data faster with “fread”\(^3\) Your results will vary based on how large your data set is, but on a data set with several columns and several million rows, I found it to be 33% faster than read.csv.

rbindlist  If you’ve ever applied a data cleaning function to a bunch of data files, you’ve probably ended up with a list of data.frame’s, and used do.call(rbind, data) to stack up your data into one data.frame. Data.table has authored an alternative which is not only significantly faster, but it also automatically transforms your output into a data.table.

```r
data <- lapply(files, cleanData)
data <- rbindlist(data)
```

For Loop and Set  If you’re ever in a tight pinch for computing resources or time, consider using for loop and set.

```r
for (var in varlist) {
  set(data, j = var, value = MyFunc(data[[var]])
}
```

The set function cuts down on some of the overhead which the “:=” operator runs into when it’s busy checking types of variables. Therefore, the “set” function is marginally faster to use. As far as I know, this function is the only other way apart from the “:=” operator to assign observations to an existing data.table. It is worth noting that the “set” function cannot assign new variables, this must be down with “:=”\(^4\)

\(^3\)It’s worth noting the the documentation for this command. The authors expect to make non-backwards compatible changes sometime in the future, and so if you use it, beware that it may break one day. The changes will be syntactic, and will not affect how “fread” reads in data. The function is otherwise safe to use. Therefore, if you choose to use it, please leave a comment letting future script readers know that it can be equivalently replaced in favor of “read.csv”

\(^4\)This can give you an idea of how using “set” can be faster than the “:=” operator. The limitation is that “set” can’t assign new variables or remove them, but because of this it doesn’t have to do as much type checking, and can cut down on some overhead.
Lastly, but not least, the authors have included fast ways to search within character vectors.

```r
data[id %chin% c("a", "e", "i", "o", "u"), unique(val)]
```

## [1] NA 158 153 154 162 161 163 170 169 167 165 164 156 166 152 157 155
## [18] 159 160 168

There are also sister functions; See `?chmatch`

### 7 Reshaping Data Tables

In the past, one had to come up with a clever function to reshape data tables. Today, the package authors have developed data.table methods for the "reshape" and "dcast" functions within "reshape2". This means that when you apply either function to a data.table, the package authors have written a specific method that is optimized to deal with data.table structures. On a practical level, you should notice that data.table reshapes are now faster.\(^5\)

### 8 Viewing Data Tables in Memory

One last trick that might be handy is to view all data.table’s existing in memory. To do so, you may use the “`tables()`” command, which lists all data.tables in memory, alongside such useful information as: how the data.table is keyed, how many rows it has, which columns it contains, and how many megabytes the object is taking up in memory. This can be handy for spotting redundancies in your workspace, such that you may see when nearly duplicated data.table’s are taking up too much memory, and should be reconciled.

---

\(^5\)It is inherently a memory intensive task, however.
9 Gotchya’s

Data.table is a powerful tool, but it’s worthy of respect. Here are several important things to watch out for.

9.1 Unique, Duplicated, and Keys

The data.table method for the functions “unique” and duplicated both default to checking for unique combinations only among keyed columns of a data.table. This actually makes a lot of intuitive sense, and there is an appeal for why somebody would want to check this with sensibly structured and well-keyed data.

The important takeaway is that whenever you use either “unique” or “duplicated”, be sure to include the argument “NULL”, because otherwise it defaults to by = key(x).

### Take unique combinations of a variable.

```r
unique(raw[1d %chin% c("a", "e"), list(val, date)], by = NULL)
```

The behavior of “by = NULL” will be identical to what you are used to with unique and data.frame.

9.2 Copy

Copy This section is worthy of another memo, but it is worth it to be extremely cautious when copying data.tables using the “<-” assignment operator. The author’s wrote data.table for the purpose of computing on big data, and so everything was designed to allow for minimal strain on physical computing resources. Instead of copying everything excessively in memory like base R, data.table leaves everything in one place in memory as much as possible.

To some extent, in a weird way, the same is actually also true in base R.

```r
## Create a matrix.
mat <- matrix(1:9, ncol = 3)
mat
```

```
[,1] [,2] [,3]
[1,]  1  4  7
[2,]  2  5  8
[3,]  3  6  9
```
### Create what we "think" are "copies"

```r
dT2 <- data
mT2 <- mat
```

### Same object, same pointer...

```r
tracemem(data)
## [1] "<0x000000000d16b980>"

tracemem(DT2)
## [1] "<0x000000000d16b980>"
```

### True in base R as well.

```r
tracemem(mat)
## [1] "<0x000000000617fe80>"

tracemem(MT2)
## [1] "<0x000000000617fe80>"
```

Notice that in both base R and in data.table, when the assignment operator is used to create what the user often thinks is a new object, R is actually creating nothing but a pointer to the old object.

This means that, yes, it’s possible to modify our “original” object even by modifying what we thought was a “copy”.

### Oh oh!

```r
setattr(mat, "dimnames", list(letters[1:3], letters[24:26]))

mT2
```

```r
## x y z
## a 1 4 7
## b 2 5 8
## c 3 6 9
```

The difference is that whereas in base R most operations other than "setattr" don’t modify objects by reference, in data.table everything is designed to modify the object by reference. Because that’s what you as a user should want out of a big data package.
### Creates new copies "safely" in cheek.

MT2[, colnames(MT2)] <- NA

## tracemem[0x000000000617fe80 -> 0x000000000d788568]: eval eval withVisible withCallingHandlers doTryCatch tryCatchOne ... evaluate in_dir block_exec call_block process_group.block process_group withCallingHandlers process_file knit

## tracemem[0x000000000d788568 -> 0x000000000da80868]: eval eval withVisible withCallingHandlers doTryCatch tryCatchOne ... evaluate in_dir block_exec call_block process_group.block process_group withCallingHandlers process_file knit

### If you do anything before assigning,

# data.table also copies "safely".

DT3 <- DT2[val < 160]

### DT2 is a pointer to "data", and data.table assigns

# by reference only. Bye bye data.

DT2[, names(DT2)[-1] := NULL]

### Creates new copies "safely" in cheek.

MT2[, colnames(MT2)] <- NA

## tracemem[0x000000000617fe80 -> 0x000000000d788568]: eval eval withVisible withCallingHandlers doTryCatch tryCatchOne ... evaluate in_dir block_exec call_block process_group.block process_group withCallingHandlers process_file knit

## tracemem[0x000000000d788568 -> 0x000000000da80868]: eval eval withVisible withCallingHandlers doTryCatch tryCatchOne ... evaluate in_dir block_exec call_block process_group.block process_group withCallingHandlers process_file knit

The reason why the second operation above, where “DT3” is created, is “safe”, can be reasoned as follows. If all that were done on “DT2” was to ask for the subset of rows, with no other operation performed, no new object would be created in memory and instead the resulting rows would simply be printed to the console. If we used the “:=” assignment operator within the same expression, the assignment would be by reference, and again of course no new object would be created.

However, when we use the “<-” operator to create “DT3”, we are explicitly asking R to either create a pointer to an existing object in memory
or we are asking R to create a new memory address to hold a new object in memory. Because it is the case that after performing a row operation on ‘DT2’ where we subset rows, this object is no longer identical to ‘DT2’, R therefore creates a new memory address instead of creating a pointer. As far as I know, it is only when the user asks R to create an identical copy of an object, without first manipulating it, that R creates a pointer.

To reiterate, the issue of copying objects on assignment is not a data.table issue, it is a base R issue.

```r
test <- character()
tracemem(test)
## [1] "<0x0000000005dd63a8>"

test.copy <- test
tracemem(test.copy)
## [1] "<0x0000000005dd63a8>"
```

If you want to avoid this behavior, and for some strange reason copy a humongous piece of data to a new memory address without altering it first, then you can abuse this power by using the “copy” command.

```r
newDT <- copy(data)
## tracemem[0x000000000d16b980 -> 0x0000000005f6e9f8]: copy eval eval withVisible with
```

In general, I advise against using the “copy” operator, as it usually is a sign that your process should be arranged differently. For example, if you want to use “copy” to rename a data.table, perhaps the data.table should be renamed appropriately when it is first created to avoid duplication. Or for example, if you wish to create a duplicated data.table, such that several operations may be applied and then the data.table may be compared with it’s “original” version, perhaps you should consider feeding the raw data.table into a function that performs the operations, such that a new environment is created to store the copied data.table. With the latter method, the user is no longer burdened by thinking about the potential for their raw data to be tampered with unknowingly.