R data.table Tutorial

(This tutorial is adapted from a technical presentation by Brian Silva, a data scientist at Uptake, LLC, a Chicago-based analytics firm focusing on heavy industry)

This tutorial presents **`data.table`**, an R package by Matt Dowle, which is an extension on R's default `data.frame` used for fast aggregation of large data (e.g. 100GB in RAM). We will compare its speed and syntax with that of data.frame.

Installation

First, make sure the `data.table` package is installed by running:

```
install.packages('data.table')
```

Data importing: `fread` vs. `read.csv`

The `fread` function, which has almost identical arguments as those of `read.csv`, works many times faster than `read.csv`. For example, a large (10GB, 100M rows x 10 cols) data set will take `read.csv` hours to read, while it will take `fread` only about 4-5 minutes.

By default `fread` will read a .csv file into a data.table, but you can also read it in as a data.frame.

```
dt <- fread('dataset.csv')                  # returns data.table
df <- fread('dataset.csv', data.table=F)    # returns data.frame
```

Syntax: `data.table` vs. `data.frame`

Let's start by making a large amount of data in both `data.frame` and `data.table` formats:

```
set.seed(35753)

numRows <- 10000000

# Create data with assets and dateTimes
assets <- paste('Asset', 1:1000, sep='_')

dateTimes <- seq.POSIXt(from=as.POSIXct('2014-01-01 00:00:00'),
                        to=as.POSIXct('2015-01-01 00:00:00'),
                        length.out=50000)

# Randomly select assets dateTimes and generate signals
DF <- data.frame(name=1:numRows,
                 assetId=sample(assets, numRows, replace=T),
                 dateTime=sample(dateTimes, numRows, replace=T),
                 signal1=runif(numRows),
                 signal2=rexp(numRows),
                 signal3=sample(c('ON', 'OFF'), numRows, replace=T),
                 stringsAsFactors=F)

DT <- as.data.table(DF)
```

Viewing the data

First of all, data.table inherits from `data.frame`, which means that it can be passed to any package that only accepts data.frame.

```
class(DT)
```

```
## [1] "data.table" "data.frame"
```

Let's take a look at the data.frame and data.table we created.

```
system.time(h <- head(DF))    # see how much time it takes to present head of data.frame
```

```
##    user  system elapsed
##    0.00    0.02    0.02
```

```
h
```

```
##   name   assetId            dateTime    signal1   signal2 signal3
## 1    1 Asset_821 2014-08-22 22:52:14 0.73100637 0.7845882     OFF
## 2    2  Asset_19 2014-09-18 04:22:36 0.08924429 0.2062187     OFF
## 3    3 Asset_743 2014-09-24 18:24:37 0.56470635 0.9394910      ON
## 4    4 Asset_969 2014-06-26 14:07:17 0.14074313 0.1588975      ON
## 5    5 Asset_431 2014-10-06 02:19:34 0.25558412 0.3531577      ON
## 6    6 Asset_693 2014-02-23 22:45:14 0.24515170 0.1918058      ON
```

```
system.time(h <- head(DT))    # see how much time it takes to present head of data.table
```

```
##    user  system elapsed
##       0       0       0
```

```
h
```

```
##    name   assetId            dateTime    signal1   signal2 signal3
## 1:    1 Asset_821 2014-08-22 22:52:14 0.73100637 0.7845882     OFF
## 2:    2  Asset_19 2014-09-18 04:22:36 0.08924429 0.2062187     OFF
## 3:    3 Asset_743 2014-09-24 18:24:37 0.56470635 0.9394910      ON
## 4:    4 Asset_969 2014-06-26 14:07:17 0.14074313 0.1588975      ON
## 5:    5 Asset_431 2014-10-06 02:19:34 0.25558412 0.3531577      ON
## 6:    6 Asset_693 2014-02-23 22:45:14 0.24515170 0.1918058      ON
```

The output looks pretty similar with the only exception being the colon after the row number. One thing to take away, though, is that it takes less time to read a `data.table` as it does a `data.frame`. This is because `data.table` only makes references to the underlying data, whereas `data.frame` copies data into a new object. This will be a recurring theme when comparing `data.table` and `data.frame`.

Another nice thing is that when you print out a large `data.table` object, you are only shown a summary and not the whole things. You wouldn't want to do the following with a huge data.frame:

```r
DT
```

```
##               name   assetId            dateTime    signal1   signal2 signal3
##        1:         1 Asset_821 2014-08-22 22:52:14 0.73100637 0.7845882     OFF
##        2:         2  Asset_19 2014-09-18 04:22:36 0.08924429 0.2062187     OFF
##        3:         3 Asset_743 2014-09-24 18:24:37 0.56470635 0.9394910      ON
##        4:         4 Asset_969 2014-06-26 14:07:17 0.14074313 0.1588975      ON
##        5:         5 Asset_431 2014-10-06 02:19:34 0.25558412 0.3531577      ON
##       ---
##  9999996:  9999996   Asset_5 2014-07-31 19:18:11 0.42248875 2.6816182      ON
##  9999997:  9999997 Asset_711 2014-07-15 10:19:18 0.39674976 0.2533132     OFF
##  9999998:  9999998  Asset_86 2014-08-17 00:46:50 0.15199186 1.7126276     OFF
##  9999999:  9999999 Asset_482 2014-06-09 13:32:49 0.88929144 0.1391500      ON
## 10000000: 10000000 Asset_660 2014-10-16 05:51:32 0.13119666 0.7132515     OFF
```

Referencing rows

Let's now look at an example of conditionally selecting certain rows in both `data.frame` and `data.table`.

Select observations of `Asset_100` where `signal3` is 'ON.':

```r
system.time(h <- head(DF[DF$assetId == 'Asset_100' & DF$signal3 == 'ON', ]))
```

```
##    user  system elapsed
##    1.24    0.09    1.37
```

```r
h
```

```
##        name   assetId            dateTime   signal1    signal2 signal3
## 2398   2398 Asset_100 2014-11-07 15:15:34 0.1371596 2.36482202      ON
## 4317   4317 Asset_100 2014-07-14 11:22:12 0.9412520 0.03864704      ON
## 5848   5848 Asset_100 2014-04-05 17:02:20 0.4315933 0.52792693      ON
## 7483   7483 Asset_100 2014-06-13 16:53:14 0.7889169 0.06761485      ON
## 8556   8556 Asset_100 2014-09-13 09:05:34 0.3837638 3.03475422      ON
## 11717 11717 Asset_100 2014-12-05 05:31:43 0.5391351 0.06165879      ON
```

```r
system.time(h <- head(DT[assetId == 'Asset_100' & signal3 == 'ON', ]))
```

```
##    user  system elapsed
##    0.96    0.05    1.00
```

```r
h
```

```
##     name   assetId            dateTime   signal1    signal2 signal3
## 1:  2398 Asset_100 2014-11-07 15:15:34 0.1371596 2.36482202      ON
## 2:  4317 Asset_100 2014-07-14 11:22:12 0.9412520 0.03864704      ON
## 3:  5848 Asset_100 2014-04-05 17:02:20 0.4315933 0.52792693      ON
## 4:  7483 Asset_100 2014-06-13 16:53:14 0.7889169 0.06761485      ON
## 5:  8556 Asset_100 2014-09-13 09:05:34 0.3837638 3.03475422      ON
## 6: 11717 Asset_100 2014-12-05 05:31:43 0.5391351 0.06165879      ON
```

Notice that within our `data.table`, we don't have to say `DT$assetId == 'Asset_100' & DT$signal3 == 'ON'`. This is because within `data.table`'s square brackets, we can reference column names directly as variables.

While in the above code `data.table` is faster than `data.frame`, it is still not the best way to subset a `data.table`. Another method involves setting a key for the `data.table` and then subsetting.

```
setkey(DT, assetId, signal3)  # set key to use binary search instead of linear scan

system.time(h <- DT[list('Asset_100', 'ON'), ])
```

```
##    user  system elapsed
##       0       0       0
```

```
h
```

```
##          name  assetId            dateTime  signal1    signal2 signal3
##    1:    2398 Asset_100 2014-11-07 15:15:34 0.1371596 2.36482202      ON
##    2:    4317 Asset_100 2014-07-14 11:22:12 0.9412520 0.03864704      ON
##    3:    5848 Asset_100 2014-04-05 17:02:20 0.4315933 0.52792693      ON
##    4:    7483 Asset_100 2014-06-13 16:53:14 0.7889169 0.06761485      ON
##    5:    8556 Asset_100 2014-09-13 09:05:34 0.3837638 3.03475422      ON
##   ---
## 5038: 9984143 Asset_100 2014-08-19 00:57:41 0.2933300 0.08590503      ON
## 5039: 9985489 Asset_100 2014-03-10 12:45:33 0.1134014 1.23107313      ON
## 5040: 9986196 Asset_100 2014-09-02 06:04:58 0.5648079 0.07894776      ON
## 5041: 9991227 Asset_100 2014-06-19 06:23:32 0.5629068 0.33177273      ON
## 5042: 9998978 Asset_100 2014-03-02 10:51:36 0.8321245 0.01032277      ON
```

While setting up the key initially can take some time, all of the later subsetting is much faster. This is because `data.table` rearranges itself to allow binary search instead of linear scan. This means that instead of checking every row for these conditions, `data.table` can immediately eliminate many rows. For comparison, the computational complexity of vector scan is $\mathbf{O}(n)$, while that of binary search is $\mathbf{O}(\log\, n)$. Additionally, `data.table` performs computations by reference instead of making a copy and performing calculations on these. This is much more performant and memory efficient.

Referencing columns

Referencing columns within `data.table` is something that can seem a little confusing at first – especially when one is used to data.frame syntax. Take the following as an example:

```
head(DF[, 'signal1'])
```

```
## [1] 0.73100637 0.08924429 0.56470635 0.14074313 0.25558412 0.24515170
```

```
head(DT[, 'signal1'])
```

```
## [1] "signal1"
```

When we apply the same syntax from `data.frame` to `data.table`, we get something quite different. To some this may seem like a bug, but it is actually made this way by design. The second argument within `data.table`, which in `data.frame` references columns, can be an **expression** and not simply column names or indexes. So when you want to return the data from `signal1`, you can do either of the following:

```r
head(DT[, signal1])
```

```
## [1] 0.3164516 0.5844961 0.3657417 0.1189339 0.6619537 0.7530080
```

```r
head(DT[['signal1']])
```

```
## [1] 0.3164516 0.5844961 0.3657417 0.1189339 0.6619537 0.7530080
```

```r
head(DT[, 'signal1', with=F])
```

```
##      signal1
## 1: 0.3164516
## 2: 0.5844961
## 3: 0.3657417
## 4: 0.1189339
## 5: 0.6619537
## 6: 0.7530080
```

In the first example we just referenced the column name directly since column names are treated as variables within `data.table`. In the second example, we are essentially treating `DT` as a list.In the last example, we had to say `with=F`. This is because we want to pass in a string directly to reference the column name. (*don't ask us why the argument is named "with="… this probably has come from a lengthy history*)

Why use data.table over data.frame?

Right now we have seen that data.table can do the same things that data.frame can do. And we have seen that it can do them a bit faster too. But the syntax seems weird and confusing at first. Is data.table really worth the extra effort?

Let's first look at the structure of data.table's arguments and then look at some examples of where this structure is incredibly useful.

data.table's arguments

data.table's inputs - often denoted as `DT[i, j, by]` - allow the following:

- `i` allows you to evaluate conditional arguments (i.e. `signal1 > 0.5`)
- `j` allows you to select or perform expressions on columns
- `by` allows you to perform evaluations by group

`data.table`'s syntax is (in many ways) analogous to SQL. For example, you can think of the inputs to data.table as the following:

```
DT[where, select|update, group by][order by][...] ... [...]
```

Cool stuff in data.table

So we know that we can pass expressions to data.table's `j` argument. Here are a couple examples of where this could be useful:

Calculate the mean of `signal1`:

```
DT[, mean(signal1)]
```

```
## [1] 0.4999985
```

Calculate the mean and standard deviation of `signal1`:

```
DT[, list(avg=mean(signal1), sd=sd(signal1))]
```

```
##          avg        sd
## 1: 0.4999985 0.2887016
```

Create a new column called `sigDif`, which is the difference between `signal2` and `signal1`:

```
head(DT[, sigDif := signal2 - signal1])
```

```
##    name assetId            dateTime   signal1      signal2 signal3     sigDif
## 1: 1652 Asset_1 2014-09-19 07:00:28 0.3164516 0.3503752812     OFF  0.0339237
## 2: 3498 Asset_1 2014-04-10 08:28:06 0.5844961 3.9385727549     OFF  3.3540767
## 3: 5981 Asset_1 2014-01-11 03:42:29 0.3657417 0.0024183812     OFF -0.3633233
## 4: 6692 Asset_1 2014-10-15 07:47:00 0.1189339 0.6390884416     OFF  0.5201546
## 5: 6832 Asset_1 2014-12-01 08:08:43 0.6619537 0.2571891844     OFF -0.4047645
## 6: 6982 Asset_1 2014-05-23 13:29:52 0.7530080 0.0008710201     OFF -0.7521370
```

Notice we use `:=` to assign calculations to this new column. We can also use `data.table`'s `by` argument to perform these calculations by group:

Calculate the mean and standard deviation of `signal1` by `assetId`:

```
DT[, list(avg=mean(signal1), sd=sd(signal1)), by=assetId]
```

```
##          assetId       avg        sd
##    1:    Asset_1 0.5022450 0.2882127
##    2:   Asset_10 0.4979278 0.2920786
##    3:  Asset_100 0.4992683 0.2886824
##    4: Asset_1000 0.5038555 0.2892187
##    5:  Asset_101 0.4990661 0.2904023
##   ---
##  996:  Asset_995 0.4956604 0.2892405
##  997:  Asset_996 0.4989271 0.2886280
##  998:  Asset_997 0.4967501 0.2887766
##  999:  Asset_998 0.4990324 0.2888313
## 1000:  Asset_999 0.4997502 0.2875915
```

Helpful links

Introduction to the data.table package in R

FAQs about the data.table package in R

Matt Dowle's "data.table" talk at useR 2014