Intro to data.table

Why are we here?

You want to know about `data.table`.

I know somethings about `data.table`.

Why do I use it?

Let's load `data.table` and find out!

```
library(data.table)
```

It's terse —>>> *It's fast* —>>> It's base-ic

`data.table` is a short jump from base R and you get a big boost in speed with less typing.

Reading in data and subsetting is really familiar.

```
# base R
df <- read.csv("./data/slides/mergtab.csv")
df[df$fname== "Jack",][1:2,]
```

```
##     uid fname lname ccode    type       date conn
## 3  du43  Jack  Hill    US desktop 2019-11-04    1
## 10 du43  Jack  Hill    US   phone 2019-11-04    1
```

```
# data.table
dt <- fread("./data/slides/mergtab.csv")
dt[fname == "Jack",][1:2,]
```

```
##     uid fname lname ccode    type       date conn
## 1: du43  Jack  Hill    US desktop 2019-11-04    1
## 2: du43  Jack  Hill    US   phone 2019-11-04    1
```

dt[i, j, by / with / on / SDCols]

Base R subsetting using [already had a SQL like interface.

This...

```
df[<subset rows expression> , <subset columns expression>]
```

...is like saying...

```
df[<WHERE> , <SELECT>]
```

dt[i, j, with / by / on / SDCols]

data.table's [function acts like base R but has some additional arguments.

[i, j] work the same, more or less, like SQL:

dt[<WHERE>, <SELECT>]

- with allows to call columns using strings
- by allows for performing operations in groups of unique values, like SQL's GROUP BY operation
- on is used when using [as a merge operator

SPOILER ALERT: j can evaluate whole expressions!

Using `with` and `data.table`'s variable scope

Unlike `data.frame`, `data.table` will look for variables using the `data.table` object's namespace.

What does this mean? Less typing!

```
# base R
df[df$fname == "Jack" &
   as.Date(df$date) > as.Date("2019-11-02") &
   df$type == "desktop",]
```

```
##     uid fname lname ccode    type       date conn
## 3  du43  Jack  Hill    US desktop 2019-11-04    1
## 38 du43  Jack  Hill    US desktop 2019-11-05    1
```

```
unique(df[, c("lname")])
```

```
## [1] Woods Hill
## Levels: Hill Woods
```

Using `with` and `data.table`'s variable scope

```
# data.table
dt[fname == "Jack" & date > "2019-11-02" & type == "desktop",]
```

```
##     uid fname lname ccode    type       date conn
## 1: du43  Jack  Hill    US desktop 2019-11-04    1
## 2: du43  Jack  Hill    US desktop 2019-11-05    1
```

```
unique(dt[, .(fname, lname)])
```

```
##     fname lname
## 1: Gretel Woods
## 2:   Jack  Hill
## 3: Hansel Woods
## 4:   Jill  Hill
```

Using `with` and `data.table`'s variable scope

In base R, you can use `with()` to search a data.frame's scope for variable names.

```
with(df, unique(uid))
```

```
## [1] jhje du43 37du b4ud
## Levels: 37du b4ud du43 jhje
```

```
with(df, df[fname == "Jill" & type == "phone",])
```

```
##      uid fname lname ccode  type       date conn
## 8   b4ud  Jill  Hill    US phone 2019-11-03    1
## 12  b4ud  Jill  Hill    US phone 2019-11-03    2
## 16  b4ud  Jill  Hill    US phone 2019-11-03    3
## 25  b4ud  Jill  Hill    US phone 2019-11-02    1
## 32  b4ud  Jill  Hill    US phone 2019-11-03    4
## 35  b4ud  Jill  Hill    US phone 2019-11-04    1
## 36  b4ud  Jill  Hill    US phone 2019-11-02    2
## 42  b4ud  Jill  Hill    US phone 2019-11-03    5
```

Using `with` and `data.table`'s variable scope

`data.table` does this by default.

```
unique(dt[,uid])
```

```
## [1] "jhje" "du43" "37du" "b4ud"
```

```
dt[fname == "Jill" & type == "phone",]
```

```
##      uid fname lname ccode  type       date conn
## 1: b4ud  Jill  Hill    US phone 2019-11-03    1
## 2: b4ud  Jill  Hill    US phone 2019-11-03    2
## 3: b4ud  Jill  Hill    US phone 2019-11-03    3
## 4: b4ud  Jill  Hill    US phone 2019-11-02    1
## 5: b4ud  Jill  Hill    US phone 2019-11-03    4
## 6: b4ud  Jill  Hill    US phone 2019-11-04    1
## 7: b4ud  Jill  Hill    US phone 2019-11-02    2
## 8: b4ud  Jill  Hill    US phone 2019-11-03    5
```

Using `with` and `data.table`'s variable scope

With data.frame you can pass a variable of column names to subset a table.

```
colNames <- c("fname", "type", "date")
df[df$fname == "Gretel", colNames][1,]
```

```
##    fname  type       date
## 1 Gretel phone 2019-11-05
```

Using `with = FALSE` brings back that behavior to data.table, that is, if you store the column names in a variable then use `with = FALSE` to call those names as you would with a `data.frame`.

```
colNames <- c("fname", "type", "date")
dt[fname == "Gretel", colNames, with = F][1,]
```

```
##     fname  type       date
## 1: Gretel phone 2019-11-05
```

Using `with` and `data.table`'s variable scope

This main message here is that a `data.table` object will always look in it's own namespace first for variables. This makes subsetting a `data.table` object easier.

Using by

by acts like SQL `GROUP BY`. It performs operations by unique values in a given column on an expression passed to `j`.

```
dt[, sum(type == "desktop"), by = .(fname)]
```

```
##     fname V1
## 1: Gretel  7
## 2:   Jack  3
## 3: Hansel  7
## 4:   Jill  5
```

You could also do:

```
dt[, sum(type == "desktop"), by = c("fname")]
```

Using by with .()

Now is also a good time to introduce .().

.() is data.table short hand for list() and it is used for concatenating variables in a data.table's namespace

```
unique(dt[, .(fname, type)])
```

```
##     fname    type
## 1: Gretel   phone
## 2:   Jack desktop
## 3: Hansel desktop
## 4: Gretel desktop
## 5:   Jill desktop
## 6:   Jill   phone
## 7:   Jack   phone
## 8: Hansel   phone
```

Using by with .()

```
dt[, .(session_cnt = sum(type == "phone")), by = .(fname)]
```

```
##     fname session_cnt
## 1: Gretel           8
## 2:   Jack           8
## 3: Hansel           4
## 4:   Jill           8
```

Chaining by

Chaining `data.table` is awesome.

```
dt[, .(con_cnt = sum(type %like% "desktop|phone")), by = .(fname)
   ][, .(fname, con_cnt, con_perc = con_cnt / sum(con_cnt) * 100)]
```

```
##     fname con_cnt con_perc
## 1: Gretel      15       30
## 2:   Jack      11       22
## 3: Hansel      11       22
## 4:   Jill      13       26
```

You can chain `data.table` all day long...

```
dt[, .(con_cnt = sum(type %like% "desktop|phone")), by = .(fname)
   ][, .(fname, con_cnt, con_perc = con_cnt / sum(con_cnt) * 100)
     ][con_perc == max(con_perc),]
```

```
##     fname con_cnt con_perc
## 1: Gretel      15       30
```

Adding columns by reference

`data.table` allows for adding columns by reference and uses an operator type syntax, `:=`.

This makes a big difference in performance when working with large datasets.

```
dt[, isGerman := ifelse(ccode == "DE", 1, 0)]
dt[, isGermanPhone := as.numeric(isGerman & type == "phone")]
head(dt[,-c("date")])
```

```
##     uid  fname lname ccode    type conn isGerman isGermanPhone
## 1: jhje Gretel Woods    DE   phone    1        1             1
## 2: jhje Gretel Woods    DE   phone    2        1             1
## 3: du43   Jack  Hill    US desktop    1        0             0
## 4: 37du Hansel Woods    DE desktop    1        1             0
## 5: jhje Gretel Woods    DE   phone    1        1             1
## 6: jhje Gretel Woods    DE desktop    1        1             0
```

Adding columns by reference

And, this operation can be done multiple times as a single call using two different methods.

```
## calling `:=` using function call syntax
dt[, `:=`(
    isGerman = ifelse(ccode == "DE", 1, 0),
    isGermanPhone = as.numeric(isGerman & type == "phone")
    )]

## calling `:=` through chaining
dt[, isGerman := ifelse(ccode == "DE", 1, 0)
   ][, isGermanPhone := as.numeric(isGerman & type == "phone")]
```

Using .N and .SD

data.table has special variables that can be used in j.

- .N counts the number of records in a given group
- .SD passes a subset of a group's data.table

.N adds a row count as a field. This is similar to table().

```
dt[, .N, by = fname]
```

```
##      fname  N
## 1: Gretel 15
## 2:   Jack 11
## 3: Hansel 11
## 4:   Jill 13
```

Using .N

```
dt[, .N]
```

```
## [1] 50
```

```
dt[, .N, by = .(fname, type)
   ][N > 3 & type == "phone"]
```

```
##     fname  type N
## 1: Gretel phone 8
## 2:   Jill phone 8
## 3:   Jack phone 8
## 4: Hansel phone 4
```

Using .SD - "*S*ub *D*ata table"

This will return subsets of the data.table object using the by argument. The use of .SD can best be shown using a print statement in j.

```
dt[,
   print(
        .SD[, .N, by = .(fname, lname)]
   ), by = .(uid)]
```

```
##     fname lname  N
## 1: Gretel Woods 15
##    fname lname  N
## 1:  Jack  Hill 11
##     fname lname  N
## 1: Hansel Woods 11
##    fname lname  N
## 1:  Jill  Hill 13

## Empty data.table (0 rows and 1 cols): uid
```

Using .SD

```
dt[,
   .SD[, .(
       maxDate = max(date),
       totalPhone = sum(type == "phone"),
       totalDesktop = sum(type == "desktop")
   )],
   by = .(uid)]
```

```
##      uid    maxDate totalPhone totalDesktop
## 1: jhje 2019-11-05          8            7
## 2: du43 2019-11-05          8            3
## 3: 37du 2019-11-05          4            7
## 4: b4ud 2019-11-04          8            5
```

Using `merge`

`merge` can be used in two ways:

- the base-ic way: `merge(x, y)`
- the data.table way: `y[x]`

That's right, `[` is also used for merges.

Using [for merging

I go back and forth using this syntax.

`[` feels a bit too implicit to me, but it's great in chains.

```
y[x]
```

This is equivalent to `merge(x, y, all.x = T)`

Using this syntax, you also need to use the `on` argument.

```
y[x, on = "<joincolumn>"]
```

Using dcast

Super Great Casting

dcast works like reshape::cast, but it can also do multi-variable casting.

```
dcast(dt, fname~type+ccode, value.var = "fname")
```

```
## Aggregate function missing, defaulting to 'length'
```

```
##     fname desktop_DE desktop_US phone_DE phone_US
## 1: Gretel          7          0        8        0
## 2: Hansel          7          0        4        0
## 3:   Jack          0          3        0        8
## 4:   Jill          0          5        0        8
```

data.table can also melt.

set functions

Mutability can make a gigantic difference when concerned with performance.

data.table offers some methods for setting column names and order that don't copy data.

- `setnames`
- `setorder`
- `setcolorder`

set functions

Setting column names with `setnames`:

```
names(dt)
```

```
## [1] "uid"           "fname"         "lname"         "ccode"
## [5] "type"          "date"          "conn"          "isGerman"
## [9] "isGermanPhone"
```

```
setnames(dt, c("fname", "lname"), c("firstname", "lastname"))
names(dt)
```

```
## [1] "uid"           "firstname"     "lastname"      "ccode"
## [5] "type"          "date"          "conn"          "isGerman"
## [9] "isGermanPhone"
```

set functions

Setting column order with `setcolorder`:

```
names(dt)
```

```
## [1] "uid"           "firstname"     "lastname"      "ccode"
## [5] "type"          "date"          "conn"          "isGerman"
## [9] "isGermanPhone"
```

```
setcolorder(dt, c("lastname", "firstname"))
names(dt)
```

```
## [1] "lastname"      "firstname"     "uid"           "ccode"
## [5] "type"          "date"          "conn"          "isGerman"
## [9] "isGermanPhone"
```

set functions

Setting row order with `setorder`:

```
setorder(dt, uid, date)
head(dt[,c(1:6)])
```

```
##    lastname firstname  uid ccode    type       date
## 1:    Woods    Hansel 37du    DE desktop 2019-11-01
## 2:    Woods    Hansel 37du    DE desktop 2019-11-01
## 3:    Woods    Hansel 37du    DE   phone 2019-11-01
## 4:    Woods    Hansel 37du    DE   phone 2019-11-01
## 5:    Woods    Hansel 37du    DE desktop 2019-11-03
## 6:    Woods    Hansel 37du    DE desktop 2019-11-03
```

Know that there are two versions of this, one that accepts strings and one that looks for variables in the `data.table` namespace.

See `?setorder` for more details.

`setKey` and indexing

On thing that can make `data.table` very fast is the use of indexes and keys.

- performance boost for large `data.table` join or subset operations
- no need to define `on` arguments when merging

setKey and indexing

```
uids <- fread("./data/slides/usertab.csv")
conn <- fread("./data/slides/conntab.csv")

setkey(uids, uid)
setkey(conn, uid)

head(uids[conn])
```

```
##     uid  fname lname ccode    type       date conn
## 1: 37du Hansel Woods    DE desktop 2019-11-04    1
## 2: 37du Hansel Woods    DE desktop 2019-11-01    1
## 3: 37du Hansel Woods    DE   phone 2019-11-04    1
## 4: 37du Hansel Woods    DE desktop 2019-11-03    1
## 5: 37du Hansel Woods    DE desktop 2019-11-03    2
## 6: 37du Hansel Woods    DE   phone 2019-11-05    1
```

Putting some of this stuff together...

What does this do?

```
dt[lname == "Woods", note:="Don't talk about witches"]
```

Assigning IDs by group...

```
dt[, conn:=(1:.N), by = .(uid, type, date)]
```

Grouping and chaining

```
dt[, firstCon := date == min(date) & con == 1, by = uid
   ][, firstConIsDesk := firstCon & device == "desktop"
     ][ firstConIsDesk == 1 ]
```

Lets work with some data!