FREE eBook

LEARNING data.table

Free unaffiliated eBook created from **Stack Overflow contributors.**

#data.table

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: data-table

It is an unofficial and free data.table ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official data.table.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with data.table

Remarks

Data.table is a package for the R statistical computing environment. It extends the functionality of data frames from base R, particularly improving on their performance and syntax. A number of related tasks, including rolling and non-equi joins, are handled in a consistent concise syntax like `DT[where, select|update|do, by]`.

A number of complementary functions are also included in the package:

- I/O: `fread`/`fwrite`
- Reshaping: `melt`/`dcast`/`rbindlist`/`split`
- Runs of values: `rleid`

Versions

Version	Notes	Release Date on CRAN
1.9.4		2014-10-02
1.9.6		2015-09-19
1.9.8		2016-11-24
1.10.0	"With hindsight, the last release v1.9.8 should have been named v1.10.0"	2016-12-03
1.10.1	In development	2016-12-03

Examples

Installation and setup

Install the stable release from CRAN:

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

Or the development version from github:

```
install.packages("data.table", type = "source",
  repos = "http://Rdatatable.github.io/data.table")
```

To revert from devel to CRAN, the current version must first be removed:

```
remove.packages("data.table")
install.packages("data.table")
```

Visit the website for full installation instructions and the latest version numbers.

Using the package

Usually you will want to load the package and all of its functions with a line like

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

If you only need one or two functions, you can refer to them like `data.table::fread` instead.

Getting started and finding help

The package's official wiki has some essential materials:

- As a new user, you will want to check out the vignettes, FAQ and cheat sheet.
- Before asking a question -- here on StackOverflow or anywhere else -- please read the support page.

For help on individual functions, the syntax is `help("fread")` or `?fread`. If the package has not been loaded, use the full name like `?data.table::fread`.

Syntax and features

Basic syntax

`DT[where, select|update|do, by]` syntax is used to work with columns of a data.table.

- The "where" part is the `i` argument
- The "select|update|do" part is the `j` argument

These two arguments are usually passed by position instead of by name.

A sequence of steps can be chained like `DT[...][...]`.

Shortcuts, special functions and special symbols inside `DT[...]`

Function or symbol	Notes
`.()`	in several arguments, replaces `list()`

Function or symbol	Notes
`J()`	in `i`, replaces `list()`
`:=`	in `j`, a function used to add or modify columns
`.N`	in `i`, the total number of rows in `j`, the number of rows in a group
`.I`	in `j`, the vector of row numbers in the table (filtered by `i`)
`.SD`	in `j`, the current subset of the data selected by the `.SDcols` argument
`.GRP`	in `j`, the current index of the subset of the data
`.BY`	in `j`, the list of by values for the current subset of data
`V1, V2, ...`	default names for unnamed columns created in `j`

Joins inside `DT[...]`

Notation	Notes
`DT1[DT2, on, j]`	join two tables
`i.*`	special prefix on DT2's columns after the join
`by=.EACHI`	special option available only with a join
`DT1[!DT2, on, j]`	anti-join two tables
`DT1[DT2, on, roll, j]`	join two tables, rolling on the last column in `on=`

Reshaping, stacking and splitting

Notation	Notes
`melt(DT, id.vars, measure.vars)`	transform to long format for multiple columns, use `measure.vars = patterns(...)`
`dcast(DT, formula)`	transform to wide format
`rbind(DT1, DT2, ...)`	stack enumerated data.tables
`rbindlist(DT_list, idcol)`	stack a list of data.tables

Notation	Notes
`split(DT, by)`	split a data.table into a list

Some other functions specialized for data.tables

Function(s)	Notes
`foverlaps`	overlap joins
`merge`	another way of joining two tables
`set`	another way of adding or modifying columns
`fintersect`, `fsetdiff`, `funion`, `fsetequal`, `unique`, `duplicated`, `anyDuplicated`	set-theory operations with rows as elements
`CJ`	the Cartesian product of vectors
`uniqueN`	the number of distinct rows
`rowidv(DT, cols)`	row ID (1 to .N) within each group determined by cols
`rleidv(DT, cols)`	group ID (1 to .GRP) within each group determined by runs of cols
`shift(DT, n)`	apply a shift operator to every column
`setorder`, `setcolorder`, `setnames`, `setkey`, `setindex`, `setattr`	modify attributes and order by reference

Other features of the package

Features	Notes
`IDate` and `ITime`	integer dates and times

Read Getting started with data.table online: https://riptutorial.com/data-table/topic/3389/getting-started-with-data-table

Chapter 2: Adding and modifying columns

Remarks

The official vignette, "Reference semantics", is the best introduction to this topic.

A reminder: `DT[where, select|update|do, by]` syntax is used to work with columns of a data.table.

- The "where" part is the `i` argument
- The "select|update|do" part is the `j` argument

These two arguments are usually passed by position instead of by name.

All modifications to columns can be done in `j`. Additionally, the `set` function is available for this use.

Examples

Editing values

```
# example data
DT = as.data.table(mtcars, keep.rownames = TRUE)
```

Editing a column

Use the `:=` operator inside `j` to create new columns or modify existing ones:

```
DT[, mpg_sq := mpg^2]
```

Editing on a subset of rows

Use the `i` argument to subset to rows "where" edits should be made:

```
DT[1:3, newvar := "Hello"]
```

As in a data.frame, we can subset using row numbers or logical tests. It is also possible to use [a "join" in `i` when modifying][need_a_link].

Removing a column

Remove columns by setting to `NULL`:

```
DT[, mpg_sq := NULL]
```

Note that we do not `<-` assign the result, since `DT` has been modified in-place.

Editing multiple columns

Add multiple columns by using the `:=` operator's multivariate format:

```
DT[, `:=`(mpg_sq = mpg^2, wt_sqrt = sqrt(wt))]
# or
DT[, c("mpg_sq", "wt_sqrt") := .(mpg^2, sqrt(wt))]
```

The `.()` syntax is used when the right-hand side of `LHS := RHS` is a list of columns.

Editing multiple sequentially-dependent columns

If the columns are dependent and must be defined in sequence, some ways to do that are:

```
DT[, c("mpg_sq", "mpg2_hp") := .(temp1 <- mpg^2, temp1/hp)]
# or
DT[, c("mpg_sq", "mpg2_hp") := {temp1 = mpg^2; .(temp1, temp1/hp)}]
```

Editing columns by dynamically-determined names

For dynamically-determined column names, use parentheses:

```
vn = "mpg_sq"
DT[, (vn) := mpg^2]
```

Using `set`

Columns can also be modified with `set` for a small reduction in overhead, though this is rarely necessary:

```
set(DT, j = "hp_over_wt", v = mtcars$hp/mtcars$wt)
```

Reordering columns

```
# example data
DT = as.data.table(mtcars, keep.rownames = TRUE)
```

To rearrange the order of columns, use `setcolorder`. For example, to reverse them

```
setcolorder(DT, rev(names(DT)))
```

This costs almost nothing in terms of performance, since it is just permuting the list of column pointers in the data.table.

Renaming columns

```
# example data
DT = as.data.table(mtcars, keep.rownames = TRUE)
```

To rename a column (while keeping its data the same), there is no need to copy the data to a column with a new name and delete the old one. Instead, we can use

```
setnames(DT, "mpg_sq", "mpq_squared")
```

to modify the original column by reference.

Modifying factor levels and other column attributes

```
# example data
DT = data.table(iris)
```

To modify factor levels by reference, use `setattr`:

```
setattr(DT$Species, "levels", c("set", "ver", "vir")
# or
DT[, setattr(Species, "levels", c("set", "ver", "vir"))]
```

The second option might print the result to the screen.

With `setattr`, we avoid the copy usually incurred when doing `levels(x) <- lvls`, but it will also skip some checks, so it is important to be careful to assign a valid vector of levels.

Read Adding and modifying columns online: https://riptutorial.com/data-table/topic/3781/adding-and-modifying-columns

Chapter 3: Cleaning data

Examples

Handling duplicates

```
# example data
DT = data.table(id = c(1,2,2,3,3,3))[, v := LETTERS[.I]][]
```

To deal with "duplicates," combine counting rows in a group and subsetting rows by group.

Keep one row per group

Aka "drop duplicates" aka "deduplicate" aka "uniquify."

```
unique(DT, by="id")
# or
DT[, .SD[1L], by=id]
#    id v
# 1:  1 A
# 2:  2 B
# 3:  3 D
```

This keeps the first row. To select a different row, one can fiddle with the `1L` part or use `order` in `i`.

Keep only unique rows

```
DT[, if (.N == 1L) .SD, by=id]
#    id v
# 1:  1 A
```

Keep only nonunique rows

```
DT[, if (.N > 1L) .SD, by=id]
#    id v
# 1:  2 B
# 2:  2 C
# 3:  3 D
# 4:  3 E
# 5:  3 F
```

Read Cleaning data online: https://riptutorial.com/data-table/topic/5206/cleaning-data

Chapter 4: Computing summary statistics

Remarks

A reminder: `DT[where, select|update|do, by]` syntax is used to work with columns of a data.table.

- The "where" part is the `i` argument
- The "select|update|do" part is the `j` argument

These two arguments are usually passed by position instead of by name.

Examples

Counting rows by group

```
# example data
DT = data.table(iris)
DT[, Bin := cut(Sepal.Length, c(4,6,8))]
```

Using `.N`

`.N` in `j` stores the number of rows in a subset. When exploring data, `.N` is handy to...

1. count rows in a group,

```
DT[Species == "setosa", .N]

# 50
```

2. or count rows in all groups,

```
DT[, .N, by=.(Species, Bin)]

#       Species   Bin  N
# 1:     setosa (4,6] 50
# 2: versicolor (6,8] 20
# 3: versicolor (4,6] 30
# 4:  virginica (6,8] 41
# 5:  virginica (4,6]  9
```

3. or find groups that have a certain number of rows.

```
DT[, .N, by=.(Species, Bin)][ N < 25 ]

#       Species   Bin  N
# 1: versicolor (6,8] 20
# 2:  virginica (4,6]  9
```

Handling missing groups

However, we are missing groups with a count of zero above. If they matter, we can use `table` from base:

```
DT[, data.table(table(Species, Bin))][ N < 25 ]

#       Species   Bin  N
# 1:  virginica (4,6]  9
# 2:     setosa (6,8]  0
# 3: versicolor (6,8] 20
```

Alternately, we can join on all groups:

```
DT[CJ(Species=Species, Bin=Bin, unique=TRUE), on=c("Species","Bin"), .N, by=.EACHI][N < 25]

#       Species   Bin  N
# 1:     setosa (6,8]  0
# 2: versicolor (6,8] 20
# 3:  virginica (4,6]  9
```

A note on `.N`:

- This example uses `.N` in `j`, where it refers to size of a subset.
- In `i`, it refers to the total number of rows.

Custom summaries

```
# example data
DT = data.table(iris)
DT[, Bin := cut(Sepal.Length, c(4,6,8))]
```

Suppose we want the `summary` function output for `Sepal.Length` along with the number of observations:

```
DT[, c(
    as.list(summary(Sepal.Length)),
    N = .N
), by=.(Species, Bin)]

#       Species   Bin Min. 1st Qu. Median  Mean 3rd Qu. Max.  N
# 1:     setosa (4,6]  4.3     4.8    5.0 5.006     5.2  5.8 50
# 2: versicolor (6,8]  6.1     6.2    6.4 6.450     6.7  7.0 20
# 3: versicolor (4,6]  4.9     5.5    5.6 5.593     5.8  6.0 30
# 4:  virginica (6,8]  6.1     6.4    6.7 6.778     7.2  7.9 41
# 5:  virginica (4,6]  4.9     5.7    5.8 5.722     5.9  6.0  9
```

We have to make `j` a list of columns. Usually, some playing around with `c`, `as.list` and `.` is enough to figure out the correct way to proceed.

Assigning summary statistics as new columns

Instead of making a summary table, we may want to store a summary statistic in a new column. We can use `:=` as usual. For example,

```
DT[, is_big := .N >= 25, by=.(Species, Bin)]
```

Pitfalls

Untidy data

If you find yourself wanting to parse column names, like

> Take the mean of `x.Length/x.Width` where `x` takes ten different values.

then you are probably looking at data embedded in column names, which is a bad idea. Read about tidy data and then reshape to long format.

Rowwise summaries

Data frames and data.tables are well-designed for tabular data, where rows correspond to observations and columns to variables. If you find yourself wanting to summarize over rows, like

> Find the standard deviation across columns for each row.

then you should probably be using a matrix or some other data format entirely.

The summary function

```
# example data
DT = data.table(iris)
DT[, Bin := cut(Sepal.Length, c(4,6,8))]
```

`summary` is handy for browsing summary statistics. Besides direct usage like `summary(DT)`, it can also be applied per-group conveniently with `split`:

```
lapply(split(DT, by=c("Species", "Bin"), drop=TRUE, keep.by=FALSE), summary)

# $`setosa.(4,6]`
#   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
#  Min.   :4.300   Min.   :2.300   Min.   :1.000   Min.   :0.100
#  1st Qu.:4.800   1st Qu.:3.200   1st Qu.:1.400   1st Qu.:0.200
#  Median :5.000   Median :3.400   Median :1.500   Median :0.200
#  Mean   :5.006   Mean   :3.428   Mean   :1.462   Mean   :0.246
```

```
#   3rd Qu.:5.200   3rd Qu.:3.675   3rd Qu.:1.575   3rd Qu.:0.300
#   Max.   :5.800   Max.   :4.400   Max.   :1.900   Max.   :0.600
#
# $`versicolor.(6,8]`
#    Sepal.Length    Sepal.Width     Petal.Length     Petal.Width
#   Min.   :6.10   Min.   :2.20   Min.   :4.000   Min.   :1.20
#   1st Qu.:6.20   1st Qu.:2.80   1st Qu.:4.400   1st Qu.:1.30
#   Median :6.40   Median :2.90   Median :4.600   Median :1.40
#   Mean   :6.45   Mean   :2.89   Mean   :4.585   Mean   :1.42
#   3rd Qu.:6.70   3rd Qu.:3.10   3rd Qu.:4.700   3rd Qu.:1.50
#   Max.   :7.00   Max.   :3.30   Max.   :5.000   Max.   :1.70
#
# [...results truncated...]
```

To include zero-count groups, set `drop=FALSE` in `split`.

Applying a summarizing function to multiple variables

```
# example data
DT = data.table(iris)
DT[, Bin := cut(Sepal.Length, c(4,6,8))]
```

To apply the same summarizing function to every column by group, we can use `lapply` and `.SD`

```
DT[, lapply(.SD, median), by=.(Species, Bin)]

#        Species   Bin Sepal.Length Sepal.Width Petal.Length Petal.Width
# 1:     setosa (4,6]          5.0         3.4         1.50         0.2
# 2: versicolor (6,8]          6.4         2.9         4.60         1.4
# 3: versicolor (4,6]          5.6         2.7         4.05         1.3
# 4:  virginica (6,8]          6.7         3.0         5.60         2.1
# 5:  virginica (4,6]          5.8         2.7         5.00         1.9
```

We can filter the columns in `.SD` with the `.SDcols` argument:

```
DT[, lapply(.SD, median), by=.(Species, Bin), .SDcols="Petal.Length"]

#        Species   Bin Petal.Length
# 1:     setosa (4,6]         1.50
# 2: versicolor (6,8]         4.60
# 3: versicolor (4,6]         4.05
# 4:  virginica (6,8]         5.60
# 5:  virginica (4,6]         5.00
```

Multiple summarizing functions

Currently, the simplest extension to multiple functions is perhaps:

```
DT[, unlist(recursive=FALSE, lapply(
    .(med = median, iqr = IQR),
    function(f) lapply(.SD, f)
)), by=.(Species, Bin), .SDcols=Petal.Length:Petal.Width]
```

```
#       Species   Bin med.Petal.Length med.Petal.Width iqr.Petal.Length iqr.Petal.Width
# 1:     setosa (4,6]             1.50             0.2            0.175           0.100
# 2: versicolor (6,8]             4.60             1.4            0.300           0.200
# 3: versicolor (4,6]             4.05             1.3            0.525           0.275
# 4:  virginica (6,8]             5.60             2.1            0.700           0.500
# 5:  virginica (4,6]             5.00             1.9            0.200           0.200
```

If you want the names to be like `Petal.Length.med` instead of `med.Petal.Length`, change the order:

```
DT[, unlist(recursive=FALSE, lapply(
    .SD,
    function(x) lapply(.(med = median, iqr = IQR), function(f) f(x))
)), by=.(Species, Bin), .SDcols=Petal.Length:Petal.Width]

#       Species   Bin Petal.Length.med Petal.Length.iqr Petal.Width.med Petal.Width.iqr
# 1:     setosa (4,6]             1.50            0.175             0.2           0.100
# 2: versicolor (6,8]             4.60            0.300             1.4           0.200
# 3: versicolor (4,6]             4.05            0.525             1.3           0.275
# 4:  virginica (6,8]             5.60            0.700             2.1           0.500
# 5:  virginica (4,6]             5.00            0.200             1.9           0.200
```

Read Computing summary statistics online: https://riptutorial.com/data-table/topic/3785/computing-summary-statistics

Chapter 5: Creating a data.table

Remarks

A data.table is an enhanced version of the data.frame class from base R. As such, its `class()` attribute is the vector `"data.table" "data.frame"` and functions that work on a data.frame will also work with a data.table. There are many ways to create, load or coerce to a data.table, as seen here.

Examples

Coerce a data.frame

To copy a data.frame as a data.table, use `as.data.table` or `data.table`:

```
DF = data.frame(x = letters[1:5], y = 1:5, z = (1:5) > 3)

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

This is rarely necessary. One exception is when using built-in datasets like `mtcars`, which must be copied since they cannot be modified in-place.

Build with data.table()

There is a constructor of the same name:

```
DT <- data.table(
  x = letters[1:5],
  y = 1:5,
  z = (1:5) > 3
)
#    x y     z
# 1: a 1 FALSE
# 2: b 2 FALSE
# 3: c 3 FALSE
# 4: d 4  TRUE
# 5: e 5  TRUE
```

Unlike `data.frame`, `data.table` will not coerce strings to factors by default:

```
sapply(DT, class)
#               x           y           z
#     "character"   "integer"   "logical"
```

Read in with fread()

We can read from a text file:

```
dt <- fread("my_file.csv")
```

Unlike `read.csv`, `fread` will read strings as strings, not as factors by default.

See the [topic on `fread`][need_a_link] for more examples.

Modify a data.frame with setDT()

For efficiency, data.table offers a way of altering a data.frame or list to make a data.table in-place:

```
# example data.frame
DF = data.frame(x = letters[1:5], y = 1:5, z = (1:5) > 3)

# modification
setDT(DF)
```

Note that we do not `<-` assign the result, since the object `DF` has been modified in-place.

The class attributes of the data.frame will be retained:

```
sapply(DF, class)
#         x         y         z
#  "factor" "integer" "logical"
```

Copy another data.table with copy()

```
# example data
DT1 = data.table(x = letters[1:2], y = 1:2, z = (1:2) > 3)
```

Due to the way data.tables are manipulated, `DT2 <- DT1` will *not* make a copy. That is, later modifications to the columns or other attributes of `DT2` will affect `DT1` as well. When you want a real copy, use

```
DT2 = copy(DT1)
```

To see the difference, here's what happens without a copy:

```
DT2 <- DT1
DT2[, w := 1:2]

DT1
#    x y     z w
# 1: a 1 FALSE 1
# 2: b 2 FALSE 2
DT2
#    x y     z w
# 1: a 1 FALSE 1
# 2: b 2 FALSE 2
```

And with a copy:

```
DT2 <- copy(DT1)
DT2[, w := 1:2]

DT1
#    x y     z
# 1: a 1 FALSE
# 2: b 2 FALSE
DT2
#    x y     z w
# 1: a 1 FALSE 1
# 2: b 2 FALSE 2
```

So the changes do not propagate in the latter case.

Read Creating a data.table online: https://riptutorial.com/data-table/topic/3782/creating-a-data-table

Chapter 6: Joins and merges

Introduction

A join combines two tables containing related columns. The term covers a wide range of operations, essentially everything except appending the two tables. "Merge" is a synonym. Type `` ?`[.data.table` `` for the official docs.

Syntax

- x[i, on, j]
  # join: data.table x & data.table or list i
- x[!i, on, j]
  # anti-join

Remarks

Working with keyed tables

If `x` & `i` have a key or `x` is keyed to match `i`'s first few columns, then the `on` can be skipped like `x[i]`.

Disambiguating column names in common

In `j` of `x[i, on, j]`, columns of `i` can be referred with `i.*` prefixes.

Grouping on subsets

In `j` of `x[i, on, j, by=.EACHI]`, `j` is computed for each row of `i`.

This is the only value of `by` worth using. For any other value, columns of `i` are not available.

Examples

Update values in a join

When data is "tidy," it is often organized into several tables. To combine the data for analysis, we need to "update" one table with values from another.

For example, we might have sales data for performances, where attributes of the performer (their budget) and of the location (its population) are stored in separate tables:

```
set.seed(1)
mainDT = data.table(
  p_id = rep(LETTERS[1:2], c(2,4)),
  geo_id = sample(rep(state.abb[c(1,25,50)], 3:1)),
  sales = sample(100, 6)
)
pDT   = data.table(id = LETTERS[1:2], budget = c(60, 75))
geoDT = data.table(id = state.abb[c(1,50)], pop = c(100, 200))

mainDT # sales data
#    p_id geo_id sales
# 1:    A     AL    95
# 2:    A     WY    66
# 3:    B     AL    62
# 4:    B     MO     6
# 5:    B     AL    20
# 6:    B     MO    17


pDT # performer attributes
#    id budget
# 1:  A     60
# 2:  B     75

geoDT # location attributes
#    id pop
# 1: AL 100
# 2: WY 200
```

When we are ready to do some analysis, we need to grab variables from these other tables:

```
DT = copy(mainDT)

DT[pDT, on=.(p_id = id), budget := i.budget]
DT[geoDT, on=.(geo_id = id), pop := i.pop]

#    p_id geo_id sales budget pop
# 1:    A     AL    95     60 100
# 2:    A     WY    66     60 200
# 3:    B     AL    62     75 100
# 4:    B     MO     6     75  NA
# 5:    B     AL    20     75 100
# 6:    B     MO    17     75  NA
```

A `copy` is taken to avoid contaminating the raw data, but we could work directly on `mainDT` instead.

Advantages to using separate tables

The advantages of this structure are covered in the paper on tidy data, but in this context:

1. *Tracing missing data.* Only rows that match up in the merge receive an assignment. We have no data for `geo_id == "MO"` above, so its variables are `NA` in our final table. If we see missing data like this unexpectedly, we can trace it back to the missing observation in the `geoDT` table and investigate from there whether we have a data problem that can be addressed.

2. *Comprehensibility.* In building our statistical model, it might be important to keep in mind that `budget` is constant for each performer. In general, understanding the structure of the data pays dividends.

3. *Memory size.* There might be a large number of performer and location attributes that don't end up in the statistical model. This way, we don't need to include them in the (possibly massive) table used for analysis.

Programmatically determining columns

If there are many columns in `pDT`, but we only want to select a few, we can use

```
p_cols = "budget"
DT[pDT, on=.(p_id = id), (p_cols) := mget(sprintf("i.%s", p_cols))]
```

The parentheses around `(p_cols) :=` are essential, as noted in the doc on creating columns.

Equi-join

```
# example data
a = data.table(id = c(1L, 1L, 2L, 3L, NA_integer_), x = 11:15)
#    id  x
# 1:  1 11
# 2:  1 12
# 3:  2 13
# 4:  3 14
# 5: NA 15

b = data.table(id = 1:2, y = -(1:2))
#    id  y
# 1:  1 -1
# 2:  2 -2
```

Intuition

Think of `x[i]` as selecting a subset of `x` for each row of `i`. This syntax mirrors matrix subsetting in base R and is consistent with the first argument meaning "where", in `DT[where, select|update|do, by]`.

One might wonder why this new syntax is worth learning, since `merge(x,i)` still works with data.tables. The short answer is that it we usually wants to merge and then do something further. The `x[i]` syntax concisely captures this pattern of use and also allows for more efficient computation. For a more detailed explanation, read FAQs 1.12 and 2.14.

Handling multiply-matched rows

By default, every row of `a` matching each row of `b` is returned:

```
a[b, on="id"]
#    id  x  y
# 1:  1 11 -1
# 2:  1 12 -1
# 3:  2 13 -2
```

This can be tweaked with `mult`:

```
a[b, on="id", mult="first"]
#    id  x  y
# 1:  1 11 -1
# 2:  2 13 -2
```

Handling unmatched rows

By default, unmatched rows of `a` still show up in the result:

```
b[a, on="id"]
#    id  y  x
# 1:  1 -1 11
# 2:  1 -1 12
# 3:  2 -2 13
# 4:  3 NA 14
# 5: NA NA 15
```

To hide these, use `nomatch`:

```
b[a, on="id", nomatch=0]
#    id  y  x
# 1:  1 -1 11
# 2:  1 -1 12
# 3:  2 -2 13
```

Note that `x[i]` will attempt to match NAs in `i`.

Counting matches returned

To count the number of matches for each row of `i`, use `.N` and `by=.EACHI`.

```
b[a, on="id", .N, by=.EACHI]
#    id N
# 1:  1 1
# 2:  1 1
# 3:  2 1
# 4:  3 0
# 5: NA 0
```

Read Joins and merges online: https://riptutorial.com/data-table/topic/4976/joins-and-merges

Chapter 7: Reshaping, stacking and splitting

Remarks

The official vignette, "Efficient reshaping using data.tables", is the best introduction to this topic.

Many reshaping tasks require moving between long and wide formats:

- Wide data is data with each column representing a seperate variable, and rows representing seperate observations
- Long data is data with the form ID | variable | value, where each row representing a observation-variable pair

Examples

melt and cast with data.table

`data.table` offers a wide range of possibilities to reshape your data both efficiently and easily

For instance, while reshaping from long to wide you can both pass several variables into the `value.var` and into the `fun.aggregate` parameters at the same time

```
library(data.table) #v>=1.9.6
DT <- data.table(mtcars)
```

Long to wide

```
dcast(DT, gear ~ cyl, value.var = c("disp", "hp"), fun = list(mean, sum))
   gear disp_mean_4 disp_mean_6 disp_mean_8 hp_mean_4 hp_mean_6 hp_mean_8 disp_sum_4
disp_sum_6 disp_sum_8 hp_sum_4 hp_sum_6 hp_sum_8
1:    3     120.100       241.5    357.6167        97     107.5  194.1667      120.1
483.0     4291.4        97       215     2330
2:    4     102.625       163.8         NaN        76     116.5       NaN      821.0
655.2        0.0       608       466        0
3:    5     107.700       145.0    326.0000       102     175.0  299.5000      215.4
145.0      652.0       204       175      599
```

This will set `gear` as the index column, while `mean` and `sum` will be calculated for `disp` and `hp` for every `gear` and `cyl` combination. In case some combinations don't exist you could specify additional parameters such as `na.rm = TRUE` (which will be passed to `mean` and `sum` functions) or specify the builtin `fill` argument. You can also add margins, drop missing combinations and subset the data. See more in `?data.table::dcast`

Wide to long

While reshaping from wide to long, you can pass columns to the `measure.vars` parameter using regular expressions, for instance

```
print(melt(DT, c("cyl", "gear"), measure = patterns("^d", "e")), n = 10)
    cyl gear variable value1 value2
 1:   6    4        1 160.00  16.46
 2:   6    4        1 160.00  17.02
 3:   4    4        1 108.00  18.61
 4:   6    3        1 258.00  19.44
 5:   8    3        1 360.00  17.02
---
60:   4    5        2   3.77   5.00
61:   8    5        2   4.22   5.00
62:   6    5        2   3.62   5.00
63:   8    5        2   3.54   5.00
64:   4    4        2   4.11   4.00
```

This will `melt` the data by `cyl` and `gear` as the index columns, while all the values for the variables that begin with `d` (`disp` & `drat`) will be present in `value1` and the values for the variables that contain the letter `e` in them (`qsec` and `gear`) will be present in the `value2` column.

You can also rename all the column names in the result while specifying `variable.name` and `value.name` arguments or decide if you want the `character` columns to be automatically converted to `factor`s or not while specifying `variable.factor` and `value.factor` arguments. See more in `?data.table::melt`

Reshape using `data.table`

`data.table` extends `reshape2`'s `melt` & `dcast` functions

(Reference: Efficient reshaping using data.tables)

```
library(data.table)

## generate some data
dt <- data.table(
  name = rep(c("firstName", "secondName"), each=4),
  numbers = rep(1:4, 2),
  value = rnorm(8)
)
dt
#          name numbers      value
# 1:  firstName       1 -0.8551881
# 2:  firstName       2 -1.0561946
# 3:  firstName       3  0.2671833
# 4:  firstName       4  1.0662379
# 5: secondName       1 -0.4771341
# 6: secondName       2  1.2830651
# 7: secondName       3 -0.6989682
# 8: secondName       4 -0.6592184
```

Long to Wide

```
dcast(data = dt,
      formula = name ~ numbers,
      value.var = "value")

#          name          1          2          3          4
```

```
# 1:  firstName  0.1836433 -0.8356286 1.5952808 0.3295078
# 2: secondName -0.8204684  0.4874291 0.7383247 0.5757814
```

On multiple columns (as of `data.table` 1.9.6)

```
## add an extra column
dt[, value2 := value * 2]

## cast multiple value columns
dcast(data = dt,
      formula = name ~ numbers,
      value.var = c("value", "value2"))

#          name    value_1    value_2   value_3   value_4   value2_1   value2_2 value2_3
value2_4
# 1:  firstName  0.1836433 -0.8356286 1.5952808 0.3295078  0.3672866 -1.6712572 3.190562
0.6590155
# 2: secondName -0.8204684  0.4874291 0.7383247 0.5757814 -1.6409368  0.9748581 1.476649
1.1515627
```

Wide to Long

```
## use a wide data.table
dt <- fread("name          1          2         3         4
firstName  0.1836433 -0.8356286 1.5952808 0.3295078
secondName -0.8204684  0.4874291 0.7383247 0.5757814", header = T)
dt
#          name          1          2         3         4
# 1:  firstName  0.1836433 -0.8356286 1.5952808 0.3295078
# 2: secondName -0.8204684  0.4874291 0.7383247 0.5757814

## melt to long, specifying the id column, and the name of the columns
## in the resulting long data.table
melt(dt,
     id.vars = "name",
     variable.name = "numbers",
     value.name = "myValue")
#          name  numbers    myValue
# 1:  firstName        1  0.1836433
# 2: secondName        1 -0.8204684
# 3:  firstName        2 -0.8356286
# 4: secondName        2  0.4874291
# 5:  firstName        3  1.5952808
# 6: secondName        3  0.7383247
# 7:  firstName        4  0.3295078
# 8: secondName        4  0.5757814
```

Going from wide to long format using melt

Melting: The basics

Melting is used to transform data from wide to long format.

Starting with a wide data set:

```
DT = data.table(ID = letters[1:3], Age = 20:22, OB_A = 1:3, OB_B = 4:6, OB_C = 7:9)
```

We can melt our data using the `melt` function in data.table. This returns another data.table in long format:

```
melt(DT, id.vars = c("ID","Age"))
1:  a  20     OB_A     1
2:  b  21     OB_A     2
3:  c  22     OB_A     3
4:  a  20     OB_B     4
5:  b  21     OB_B     5
6:  c  22     OB_B     6
7:  a  20     OB_C     7
8:  b  21     OB_C     8
9:  c  22     OB_C     9

class(melt(DT, id.vars = c("ID","Age")))
# "data.table" "data.frame"
```

Any columns not set in the `id.vars` parameter are assumed to be variables. Alternatively, we can set these explicitly using the `measure.vars` argument:

```
melt(DT, measure.vars = c("OB_A","OB_B","OB_C"))
   ID Age variable value
1:  a  20     OB_A     1
2:  b  21     OB_A     2
3:  c  22     OB_A     3
4:  a  20     OB_B     4
5:  b  21     OB_B     5
6:  c  22     OB_B     6
7:  a  20     OB_C     7
8:  b  21     OB_C     8
9:  c  22     OB_C     9
```

In this case, any columns not set in `measure.vars` are assumed to be IDs.

If we set both explicitly, it will only return the columns selected:

```
melt(DT, id.vars = "ID", measure.vars = c("OB_C"))
   ID variable value
1:  a     OB_C     7
2:  b     OB_C     8
3:  c     OB_C     9
```

Naming variables and values in the result

We can manipulate the column names of the returned table using `variable.name` and `value.name`

```
melt(DT,
     id.vars = c("ID"),
     measure.vars = c("OB_C"),
     variable.name = "Test",
     value.name = "Result"
```

```
    )
   ID Test Result
1:  a OB_C      7
2:  b OB_C      8
3:  c OB_C      9
```

Setting types for measure variables in the result

By default, melting a data.table converts all `measure.vars` to factors:

```
M_DT <- melt(DT,id.vars = c("ID"), measure.vars = c("OB_C"))
class(M_DT[, variable])
# "factor"
```

To set as character instead, use the `variable.factor` argument:

```
M_DT <- melt(DT,id.vars = c("ID"), measure.vars = c("OB_C"), variable.factor = FALSE)
class(M_DT[, variable])
# "character"
```

Values generally inherit from the data type of the originating column:

```
class(DT[, value])
# "integer"
class(M_DT[, value])
# "integer"
```

If there is a conflict, data types will be coerced. For example:

```
M_DT <- melt(DT,id.vars = c("Age"), measure.vars = c("ID","OB_C"))
class(M_DT[, value])
# "character"
```

When melting, any factor variables will be coerced to character type:

```
DT[, OB_C := factor(OB_C)]
M_DT <- melt(DT,id.vars = c("ID"), measure.vars = c("OB_C"))
class(M_DT)
# "character"
```

To avoid this and preserve the initial typing, use the `value.factor` argument:

```
M_DT <- melt(DT,id.vars = c("ID"), measure.vars = c("OB_C"), value.factor = TRUE)
class(M_DT)
# "factor"
```

Handling missing values

By default, any `NA` values are preserved in the molten data

```
DT = data.table(ID = letters[1:3], Age = 20:22, OB_A = 1:3, OB_B = 4:6, OB_C = c(7:8,NA))
melt(DT,id.vars = c("ID"), measure.vars = c("OB_C"))
   ID variable value
1:  a     OB_C     7
2:  b     OB_C     8
3:  c     OB_C    NA
```

If these should be removed from your data, set `na.rm = TRUE`

```
melt(DT,id.vars = c("ID"), measure.vars = c("OB_C"), na.rm = TRUE)
   ID variable value
1:  a     OB_C     7
2:  b     OB_C     8
```

Going from long to wide format using dcast

Casting: The Basics

Casting is used to transform data from long to wide format.

Starting with a long data set:

```
DT = data.table(ID = rep(letters[1:3],3), Age = rep(20:22,3), Test =
rep(c("OB_A","OB_B","OB_C"), each = 3), Result = 1:9)
```

We can cast our data using the `dcast` function in data.table. This returns another data.table in wide format:

```
dcast(DT, formula = ID ~ Test, value.var = "Result")
   ID OB_A OB_B OB_C
1:  a    1    4    7
2:  b    2    5    8
3:  c    3    6    9

class(dcast(DT, formula = ID ~ Test, value.var = "Result"))
[1] "data.table" "data.frame"
```

Casting a value

A `value.var` argument is necessary for a proper cast - if not provided dcast will make an assumption based on your data.

```
dcast(DT, formula = ID ~ Test, value.var = "Result")
```

```
   ID OB_A OB_B OB_C
1:  a    1    4    7
2:  b    2    5    8
3:  c    3    6    9

   ID OB_A OB_B OB_C
1:  a   20   20   20
2:  b   21   21   21
3:  c   22   22   22
```

Multiple `value.var`s can be provided in a list

```
dcast(DT, formula = ID ~ Test, value.var = list("Result","Age"))
   ID Result_OB_A Result_OB_B Result_OB_C Age_OB_A Age_OB_B Age_OB_C
1:  a           1           4           7       20       20       20
2:  b           2           5           8       21       21       21
3:  c           3           6           9       22       22       22
```

Formula

Casting is controlled using the formula argument in `dcast`. This is of the form ROWS ~ COLUMNS

```
dcast(DT, formula = ID ~ Test, value.var = "Result")
   ID OB_A OB_B OB_C
1:  a    1    4    7
2:  b    2    5    8
3:  c    3    6    9

dcast(DT, formula = Test ~ ID, value.var = "Result")
   Test a b c
1: OB_A 1 2 3
2: OB_B 4 5 6
3: OB_C 7 8 9
```

Both rows and columns can be expanded with further variables using `+`

```
dcast(DT, formula = ID + Age ~ Test, value.var = "Result")
   ID Age OB_A OB_B OB_C
1:  a  20    1    4    7
2:  b  21    2    5    8
3:  c  22    3    6    9

dcast(DT, formula = ID ~ Age + Test, value.var = "Result")
   ID 20_OB_A 20_OB_B 20_OB_C 21_OB_A 21_OB_B 21_OB_C 22_OB_A 22_OB_B 22_OB_C
1:  a       1       4       7      NA      NA      NA      NA      NA      NA
2:  b      NA      NA      NA       2       5       8      NA      NA      NA
3:  c      NA      NA      NA      NA      NA      NA       3       6       9

#order is important

dcast(DT, formula = ID ~ Test + Age, value.var = "Result")
   ID OB_A_20 OB_A_21 OB_A_22 OB_B_20 OB_B_21 OB_B_22 OB_C_20 OB_C_21 OB_C_22
1:  a       1      NA      NA       4      NA      NA       7      NA      NA
2:  b      NA       2      NA      NA       5      NA      NA       8      NA
3:  c      NA      NA       3      NA      NA       6      NA      NA       9
```

Casting can often create cells where no observation exists in the data. By default this is denoted by `NA`, as above. We can override this with the `fill=` argument.

```
dcast(DT, formula = ID ~ Test + Age, value.var = "Result", fill = 0)
   ID OB_A_20 OB_A_21 OB_A_22 OB_B_20 OB_B_21 OB_B_22 OB_C_20 OB_C_21 OB_C_22
1:  a       1       0       0       4       0       0       7       0       0
2:  b       0       2       0       0       5       0       0       8       0
3:  c       0       0       3       0       0       6       0       0       9
```

You can also use two special variables in the formula object

- . represents no other variables
- ... represents all other variables

```
dcast(DT, formula = Age ~ ., value.var = "Result")
   Age .
1:  20 3
2:  21 3
3:  22 3

dcast(DT, formula = ID + Age ~ ..., value.var = "Result")
   ID Age OB_A OB_B OB_C
1:  a  20    1    4    7
2:  b  21    2    5    8
3:  c  22    3    6    9
```

Aggregating our value.var

We can also cast and aggregate values in one step. In this case, we have three observations in each of the intersections of Age and ID. To set what aggregation we want, we use the `fun.aggregate` argument:

```
#length
dcast(DT, formula = ID ~ Age, value.var = "Result", fun.aggregate = length)
   ID 20 21 22
1:  a  3  0  0
2:  b  0  3  0
3:  c  0  0  3

#sum
dcast(DT, formula = ID ~ Age, value.var = "Result", fun.aggregate = sum)
   ID 20 21 22
1:  a 12  0  0
2:  b  0 15  0
3:  c  0  0 18

#concatenate
dcast(DT, formula = ID ~ Age, value.var = "Result", fun.aggregate =
function(x){paste(x,collapse = "_")})
ID    20    21    22
1:  a 1_4_7
2:  b       2_5_8
3:  c             3_6_9
```

We can also pass a list to `fun.aggregate` to use multiple functions

```
dcast(DT, formula = ID ~ Age, value.var = "Result", fun.aggregate = list(sum,length))
   ID Result_sum_20 Result_sum_21 Result_sum_22 Result_length_20 Result_length_21
Result_length_22
1:  a            12             0             0                3                0
0
2:  b             0            15             0                0                3
0
3:  c             0             0            18                0                0
3
```

If we pass more than one function and more than one value, we can calculate all combinations by passing a vector of value.vars

```
dcast(DT, formula = ID ~ Age, value.var = c("Result","Test"), fun.aggregate =
list(function(x){paste0(x,collapse = "_")},length))
   ID Result_function_20 Result_function_21 Result_function_22 Test_function_20
Test_function_21 Test_function_22 Result_length_20 Result_length_21
1:  a              1_4_7                                         OB_A_OB_B_OB_C
3                0
2:  b                                 2_5_8
OB_A_OB_B_OB_C                                  0                3
3:  c                                                    3_6_9
OB_A_OB_B_OB_C                0                 0
   Result_length_22 Test_length_20 Test_length_21 Test_length_22
1:                0              3              0              0
2:                0              0              3              0
3:                3              0              0              3
```

where each pair is calculated in the order `value1_formula1, value1_formula2, ... , valueN_formula(N-1), valueN_formulaN`.

Alternatively, we can evaluate our values and functions one-to-one by passing 'value.var' as a list:

```
dcast(DT, formula = ID ~ Age, value.var = list("Result","Test"), fun.aggregate =
list(function(x){paste0(x,collapse = "_")},length))
   ID Result_function_20 Result_function_21 Result_function_22 Test_length_20 Test_length_21
Test_length_22
1:  a              1_4_7                                                    3              0
0
2:  b                                 2_5_8                                 0              3
0
3:  c                                                    3_6_9              0              0
3
```

Naming columns in the result

By default, column name components are seperated by an underscore _. This can be manually overridden using the `sep=` argument:

```
dcast(DT, formula = Test ~ ID + Age, value.var = "Result")
Test a_20 b_21 c_22
```

```
1: OB_A    1    2    3
2: OB_B    4    5    6
3: OB_C    7    8    9

dcast(DT, formula = Test ~ ID + Age, value.var = "Result", sep = ",")
   Test a,20 b,21 c,22
1: OB_A    1    2    3
2: OB_B    4    5    6
3: OB_C    7    8    9
```

This will seperate any `fun.aggregate` or `value.var` we use:

```
dcast(DT, formula = Test ~ ID + Age, value.var = "Result", fun.aggregate = c(sum,length), sep
= ",")
   Test Result,sum,a,20 Result,sum,b,21 Result,sum,c,22 Result,length,a,20 Result,length,b,21
Result,length,c,22
1: OB_A               1               2               3                  1                  1
1
2: OB_B               4               5               6                  1                  1
1
3: OB_C               7               8               9                  1                  1
1
```

Stacking multiple tables using rbindlist

A common refrain in R goes along these lines:

> You should not have a bunch of related tables with names like `DT1`, `DT2`, ..., `DT11`. Iteratively reading and assigning to objects by name is messy. The solution is a list of tables of data!

Such a list looks like

```
set.seed(1)
DT_list = lapply(setNames(1:3, paste0("D", 1:3)), function(i)
  data.table(id = 1:2, v = sample(letters, 2)))

$D1
   id v
1:  1 g
2:  2 j

$D2
   id v
1:  1 o
2:  2 w

$D3
   id v
1:  1 f
2:  2 w
```

Another perspective is that you should store these tables together *as one table*, by stacking them. This is straightforward to do using `rbindlist`:

```
DT = rbindlist(DT_list, id="src")

   src id v
1:  D1  1 g
2:  D1  2 j
3:  D2  1 o
4:  D2  2 w
5:  D3  1 f
6:  D3  2 w
```

This format makes a lot more sense with data.table syntax, where "by group" operations are common and straightforward.

For a deeper look, Gregor's answer might be a good place to start. Also check out `?rbindlist`, of course. There's a separate example covering reading in a bunch of tables from CSV and then stacking them.

Read Reshaping, stacking and splitting online: https://riptutorial.com/data-table/topic/4117/reshaping--stacking-and-splitting

Chapter 8: Subsetting rows by group

Remarks

A reminder: `DT[where, select|update|do, by]` syntax is used to work with columns of a data.table.

- The "where" part is the `i` argument
- The "select|update|do" part is the `j` argument

These two arguments are usually passed by position instead of by name.

Examples

Selecting rows within each group

```
# example data
DT <- data.table(Titanic)
```

Suppose that, for each sex, we want the rows with the highest survival numbers:

```
DT[Survived == "Yes", .SD[ N == max(N) ], by=Sex]

#    Class    Sex   Age Survived   N
# 1:  Crew   Male Adult      Yes 192
# 2:   1st Female Adult      Yes 140
```

`.SD` is the subset of data associated with each `Sex`; and we are subsetting this further, to the rows that meet our condition. If speed is important, instead use an approach suggested by eddi on SO:

```
DT[ DT[Survived == "Yes", .I[ N == max(N) ], by=Sex]$V1 ]

#    Class    Sex   Age Survived   N
# 1:  Crew   Male Adult      Yes 192
# 2:   1st Female Adult      Yes 140
```

Pitfalls

In the last line of code, `.I` refers to the row numbers of the full data.table. However, this is not true when there is no `by`:

```
DT[ Survived == "Yes", .I]

# 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16

DT[ Survived == "Yes", .I, by=Sex]$I

# 17 18 19 20 25 26 27 28 21 22 23 24 29 30 31 32
```

Selecting groups

```
# example data
DT = data.table(Titanic)
```

Suppose we only want to see second class:

```
DT[ Class == "2nd" ]

#    Class    Sex   Age Survived   N
# 1:   2nd   Male Child       No   0
# 2:   2nd Female Child       No   0
# 3:   2nd   Male Adult       No 154
# 4:   2nd Female Adult       No  13
# 5:   2nd   Male Child      Yes  11
# 6:   2nd Female Child      Yes  13
# 7:   2nd   Male Adult      Yes  14
# 8:   2nd Female Adult      Yes  80
```

Here, we simply subset the data using `i`, the "where" clause.

Selecting groups by condition

```
# example data
DT = data.table(Titanic)
```

Suppose we want to see each class only if a majority survived:

```
DT[, if (sum(N[Survived=="Yes"]) > sum(N[Survived=="No"]) ) .SD, by=Class]

#    Class    Sex   Age Survived   N
# 1:   1st   Male Child       No   0
# 2:   1st Female Child       No   0
# 3:   1st   Male Adult       No 118
# 4:   1st Female Adult       No   4
# 5:   1st   Male Child      Yes   5
# 6:   1st Female Child      Yes   1
# 7:   1st   Male Adult      Yes  57
# 8:   1st Female Adult      Yes 140
```

Here, we return the subset of data `.SD` only if our condition is met. An alternative is

```
DT[, .SD[ sum(N[Survived=="Yes"]) > sum(N[Survived=="No"]) ) ], by=Class]
```

but this has sometimes proven slower.

Read Subsetting rows by group online: https://riptutorial.com/data-table/topic/3784/subsetting-rows-by-group

Chapter 9: Using .SD and .SDcols for the subset of data

Introduction

The special symbol `.SD` is available in `j` of `DT[i,j,by]`, capturing the **S**ubset of **D**ata for each `by` group surviving the filter, `i`. `.SDcols` is a helper. Type ``?`special-symbols` `` for the official docs.

Remarks

A reminder: `DT[where, select|update|do, by]` syntax is used to work with columns of a data.table.

- The "where" part is the `i` argument
- The "select|update|do" part is the `j` argument

These two arguments are usually passed by position instead of by name.

Examples

Using .SD and .SDcols

.SD

`.SD` refers to the subset of the `data.table` for each group, excluding all columns used in `by`.

`.SD` along with `lapply` can be used to apply any function to multiple columns by group in a `data.table`

We will continue using the same built-in dataset, `mtcars`:

```
mtcars = data.table(mtcars) # Let's not include rownames to keep things simpler
```

Mean of all columns in the dataset by *number of cylinders*, `cyl`:

```
mtcars[ , lapply(.SD, mean), by = cyl]

#   cyl      mpg     disp        hp     drat       wt     qsec        vs        am     gear
carb
#1:   6 19.74286 183.3143 122.28571 3.585714 3.117143 17.97714 0.5714286 0.4285714 3.857143
3.428571
#2:   4 26.66364 105.1364  82.63636 4.070909 2.285727 19.13727 0.9090909 0.7272727 4.090909
1.545455
#3:   8 15.10000 353.1000 209.21429 3.229286 3.999214 16.77214 0.0000000 0.1428571 3.285714
3.500000
```

Apart from `cyl`, there are other categorical columns in the dataset such as `vs`, `am`, `gear` and `carb`. It doesn't really make sense to take the `mean` of these columns. So let's exclude these columns. This is where `.SDcols` comes into the picture.

.SDcols

`.SDcols` specifies the columns of the `data.table` that are included in `.SD`.

Mean of all columns (continuous columns) in the dataset by *number of gears* `gear`, and *number of cylinders*, `cyl`, arranged by `gear` and `cyl`:

```
# All the continuous variables in the dataset
cols_chosen <- c("mpg", "disp", "hp", "drat", "wt", "qsec")

mtcars[order(gear, cyl), lapply(.SD, mean), by = .(gear, cyl), .SDcols = cols_chosen]

#   gear cyl    mpg     disp       hp     drat       wt    qsec
#1:    3   4 21.500 120.1000  97.0000 3.700000 2.465000 20.0100
#2:    3   6 19.750 241.5000 107.5000 2.920000 3.337500 19.8300
#3:    3   8 15.050 357.6167 194.1667 3.120833 4.104083 17.1425
#4:    4   4 26.925 102.6250  76.0000 4.110000 2.378125 19.6125
#5:    4   6 19.750 163.8000 116.5000 3.910000 3.093750 17.6700
#6:    5   4 28.200 107.7000 102.0000 4.100000 1.826500 16.8000
#7:    5   6 19.700 145.0000 175.0000 3.620000 2.770000 15.5000
#8:    5   8 15.400 326.0000 299.5000 3.880000 3.370000 14.5500
```

Maybe we don't want to calculate the `mean` by groups. To calculate the mean for all the cars in the dataset, we don't specify the `by` variable.

```
mtcars[ , lapply(.SD, mean), .SDcols = cols_chosen]

#        mpg     disp       hp     drat      wt     qsec
#1: 20.09062 230.7219 146.6875 3.596563 3.21725 17.84875
```

Note: It is not necessary to define `cols_chosen` beforehand. `.SDcols` can directly take column names

Read Using .SD and .SDcols for the subset of data online: https://riptutorial.com/data-table/topic/3787/using--sd-and--sdcols-for-the-subset-of-data

Chapter 10: Using keys and indices

Introduction

The key and indices of a data.table allow certain computations to run faster, mostly related to joins and subsetting. The key describes the table's current sort order; while each index stores information about the order of the table with respect a sequence of columns. See the "Remarks" section below for links to the official vignettes on the topic.

Remarks

The official vignettes are the best introduction to this topic:

- "Keys and fast binary search based subset"
- "Secondary indices and auto indexing"

Keys vs indices

A data.table can be "keyed" by a sequence of columns, telling interested functions that the data is sorted by those columns. To get or set the key, use the functions documented at `?key`.

Similarly, functions can take advantage of a data.table's "indices." Each index -- and a table can have more than one -- stores information about the order of the data with respect a sequence of columns. Like a key, an index can speed up certain tasks. To get or set indices, use the functions documented at `?indices`.

Indices may also be set automatically (currently only for a single column at a time). See `?datatable.optimize` for details on how this works and how to disable it if necessary.

Verification and updating

Missing values are allowed in a key column.

Keys and indices are stored as attributes and may, by accident, not correspond to the actual order of data in the table. Many functions check the validity of the key or index before using it, but it's worth keeping in mind.

Keys and indices are removed after updates where it's not obvious that sort order is preserved. For example, starting from `DT = data.table(a=c(1,2,4), key="a")`, if we update like `DT[2, a := 3]`, the key is broken.

Examples

Improving performance for selecting subsets

```
# example data
set.seed(1)
n  = 1e7
ng = 1e4
DT = data.table(
    g1  = sample(ng, n, replace=TRUE),
    g2  = sample(ng, n, replace=TRUE),
    v  = rnorm(n)
)
```

Matching on one column

After the first run of a subsetting operation with `==` or `%in%`...

```
system.time(
    DT[ g1 %in% 1:100]
)
#    user  system elapsed
#    0.12    0.03    0.16
```

An index has been created automatically for `g1`. Subsequent subsetting operations run almost instantly:

```
system.time(
    DT[ g1 %in% 1:100]
)
#    user  system elapsed
#       0       0       0
```

To monitor when an index is created or used, add the `verbose=TRUE` option or change the global setting `options(datatable.verbose=TRUE)`.

Matching on multiple columns

Currently, matching on two columns does not automatically create an index:

```
system.time(
    DT[ g1 %in% 1:100 & g2 %in% 1:100]
)
#    user  system elapsed
#    0.57    0.00    0.57
```

Re-run this and it will remain slow. Even if we manually add the index with `setindex(DT, g1, g2)`, it will remain slow because this query is not yet optimized by the package.

Fortunately, if we can enumerate the combinations of values we want to search for and an index is available, we can quickly equi-join:

```
system.time(
    DT[ CJ(g1 = 1:100, g2 = 1:100, unique=TRUE), on=.(g1, g2), nomatch=0]
)
#    user  system elapsed
#    0.53    0.00    0.54
setindex(DT, g1, g2)
system.time(
    DT[ CJ(g1 = 1:100, g2 = 1:100, unique=TRUE), on=.(g1, g2), nomatch=0]
)
#    user  system elapsed
#       0       0       0
```

With `CJ`, it's important to watch out for the number of combinations becoming too large.

Read Using keys and indices online: https://riptutorial.com/data-table/topic/4977/using-keys-and-indices

Chapter 11: Using list columns to store data

Introduction

Data.table supports column vectors belonging to R's `list` class.

Remarks

In case it looks weird that we're talking about lists without using that word in the code, note that `.()` is an alias for `list()` when used inside a `DT[...]` call.

Examples

Reading in many related files

Suppose we want to read and stack a bunch of similarly-formatted files. The quick solution is:

```
rbindlist(lapply(list.files(patt="csv$"), fread), id=TRUE)
```

We might not be satisfied with this for a couple reasons:

- It might run into errors when reading with `fread` or when stacking with `rbindlist` due to inconsistent or buggy data formatting.
- We may want to keep track of metadata for each file, grabbed from the file name or perhaps from some header rows within the (not quite tabular) files.

One way to handle this is to make a "files table" and store the contents of each file as a list-column entry on the row associated with it.

Example data

Before making the example data below, make sure you're in an empty folder you can write to. Run `getwd()` and read `?setwd` if you need to change folders.

```
# example data
set.seed(1)
for (i in 1:3)
  fwrite(data.table(id = 1:2, v = sample(letters, 2)), file = sprintf("file201%s.csv", i))
```

Identify files and file metadata

This part is fairly straightforward:

```
# First, identify the files you want:
fileDT = data.table(fn = list.files(pattern="csv$"))

# Next, optionally parse the names for metadata using regex:
fileDT[, year := type.convert(sub(".*([0-9]{4}).*", "\\1", fn))]

# Finally construct a string file-ID column:
fileDT[, id := as.character(.I)]

#              fn year id
# 1: file2011.csv 2011  1
# 2: file2012.csv 2012  2
# 3: file2013.csv 2013  3
```

Read in files

Read in the files as a list column:

```
fileDT[, contents := .(lapply(fn, fread))]

#              fn year id     contents
# 1: file2011.csv 2011  1 <data.table>
# 2: file2012.csv 2012  2 <data.table>
# 3: file2013.csv 2013  3 <data.table>
```

If there's a snag in reading one of the files or you need to change the arguments to `fread` depending on the file's attributes, this step can easily be extended, looking like:

```
fileDT[, contents := {
  cat(fn, "\n")

  dat = if (year %in% 2011:2012){
    fread(fn, some_args)
  } else {
    fread(fn)
  }

  .(.(dat))
}, by=fn]
```

For details on options for reading in CSVs and similar files, see `?fread`.

Stack data

From here, we want to stack the data:

```
fileDT[, rbindlist(setNames(contents, id), idcol="file_id")]

#    file_id id v
# 1:       1  1 g
# 2:       1  2 j
# 3:       2  1 o
```

```
# 4:        2  2 w
# 5:        3  1 f
# 6:        3  2 w
```

If some problem occurs in stacking (like column names or classes not matching), we can go back to the individual tables in `fileDT` to inspect where the problem originated. For example,

```
fileDT[id == "2", contents[[1]]]
#    id v
# 1:  1 o
# 2:  2 w
```

Extensions

If the files are not in your current working dir, use

```
my_dir = "whatever"
fileDT = data.table(fn = list.files(my_dir, pattern="*.csv"))

# and when reading
fileDT[, contents := .(lapply(fn, function(n) fread(file.path(my_dir, n))))]
```

Read Using list columns to store data online: https://riptutorial.com/data-table/topic/4456/using-list-columns-to-store-data

Chapter 12: Why is my old code not working?

Introduction

The data.table package has undergone a number of changes and innovations over time. Here are some potential pitfalls that can help users looking at legacy code or reviewing old blog posts.

Examples

unique and duplicated no longer works on keyed data.table

This is for those moving to data.table >= 1.9.8

You have a data set of pet owners and names, but you suspect some repeated data has been captured.

```
library(data.table)
DT <- data.table(pet = c("dog","dog","cat","dog"),
                 owner = c("Alice","Bob","Charlie","Alice"),
                 entry.date = c("31/12/2015","31/12/2015","14/2/2016","14/2/2016"),
                 key = "owner")


> tables()
     NAME NROW NCOL MB COLS                 KEY
[1,] DT      4    3  1 pet,owner,entry.date owner
Total: 1MB
```

Recall keying a table will sort it. Alice has been entered twice.

```
> DT
   pet   owner entry.date
1: dog   Alice 31/12/2015
2: dog   Alice  14/2/2016
3: dog     Bob 31/12/2015
4: cat Charlie  14/2/2016
```

Say you used `unique` to get rid of duplicates in your data based on the key, using the most recent data capture date by setting fromLast to TRUE.

1.9.8

```
clean.DT <- unique(DT, fromLast = TRUE)

> tables()
    NAME      NROW NCOL MB COLS                 KEY
[1,] clean.DT    3    3  1 pet,owner,entry.date owner
[2,] DT          4    3  1 pet,owner,entry.date owner
Total: 2MB
```

Alice duplicate been removed.

1.9.8

```
clean.DT <- unique(DT, fromLast = TRUE)

> tables()
     NAME      NROW NCOL MB COLS                  KEY
[1,] clean.DT     4    3  1 pet,owner,entry.date owner
[2,] DT           4    3  1 pet,owner,entry.date owner
```

This does not work. Still 4 rows!

Fix

Use the `by=` parameter **which no longer defaults to your key** but to all columns.

```
clean.DT <- unique(DT, by = key(DT), fromLast = TRUE)
```

Now all is well.

```
> clean.DT
   pet   owner entry.date
1: dog   Alice  14/2/2016
2: dog     Bob 31/12/2015
3: cat Charlie  14/2/2016
```

Details and stopgap fix

See item 1 in the NEWS release notes for details:

> Changes in v1.9.8 (on CRAN 25 Nov 2016)
>
> POTENTIALLY BREAKING CHANGES
>
> 1. By default all columns are now used by `unique()`, `duplicated()` and `uniqueN()` data.table methods, #1284 and #1841. To restore old behaviour: `options(datatable.old.unique.by.key=TRUE)`. In 1 year this option to restore the old default will be deprecated with warning. In 2 years the option will be removed. Please explicitly pass `by=key(DT)` for clarity. Only code that relies on the default is affected. 266 CRAN and Bioconductor packages using data.table were checked before release. 9 needed to change and were notified. Any lines of code without test coverage will have been missed by these checks. Any packages not on CRAN or Bioconductor were not checked.

So you can use the options as a temporary workaround until your code is fixed.

```
options(datatable.old.unique.by.key=TRUE)
```

Read Why is my old code not working? online: https://riptutorial.com/data-table/topic/8196/why-is-my-old-code-not-working-

Credits

S. No	Chapters	Contributors
1	Getting started with data.table	Community, Frank, micstr
2	Adding and modifying columns	eddi, Frank, jangorecki, micstr
3	Cleaning data	Frank
4	Computing summary statistics	Frank
5	Creating a data.table	Chris, Frank
6	Joins and merges	Chris, Frank
7	Reshaping, stacking and splitting	Chris, David Arenburg, Frank, SymbolixAU
8	Subsetting rows by group	Frank, micstr
9	Using .SD and .SDcols for the subset of data	Frank
10	Using keys and indices	Frank
11	Using list columns to store data	Frank
12	Why is my old code not working?	Frank, micstr