EBook Gratuito

APPRENDIMENTO
data.table

Free unaffiliated eBook created from
Stack Overflow contributors.

#data.table

Sommario

Di

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

Capitolo 1: Iniziare con data.table

Osservazioni

Data.table è un pacchetto per l'ambiente di calcolo statistico R. Estende le funzionalità dei frame di dati dalla base R, migliorando in particolare le loro prestazioni e sintassi. Un certo numero di attività correlate, compresi i join rotanti e non equi, vengono gestiti in una sintassi concisa coerente come `DT[where, select|update|do, by]`.

Un numero di funzioni complementari sono anche incluse nel pacchetto:

- I / O: `fread` / `fwrite`
- Rimodellamento: `melt` / `dcast` / `rbindlist` / `split`
- Corse di valori: `rleid`

Versioni

Versione	Gli appunti	Data di rilascio su CRAN
1.9.4		2014/10/02
1.9.6		2015/09/19
1.9.8		2016/11/24
1.10.0	"Con il senno di poi, l'ultima release v1.9.8 avrebbe dovuto essere denominata v1.10.0"	2016/12/03
1.10.1	In sviluppo	2016/12/03

Examples

Installazione e configurazione

Installa la versione stabile da CRAN:

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

Oppure la versione di sviluppo di github:

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

Per ripristinare da devel a CRAN, è necessario rimuovere prima la versione corrente:

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

Visita il sito Web per le istruzioni complete di installazione e gli ultimi numeri di versione.

Usando il pacchetto

Solitamente vorrai caricare il pacchetto e tutte le sue funzioni con una linea simile

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

Se hai solo bisogno di una o due funzioni, puoi fare riferimento ad esse come `data.table::fread` .

Iniziare e trovare aiuto

La wiki ufficiale del pacchetto ha alcuni materiali essenziali:

- Come nuovo utente, vorrai controllare le vignette, le FAQ e il cheat sheet .
- Prima di fare una domanda - qui su StackOverflow o altrove - leggi la pagina di supporto .

Per informazioni sulle singole funzioni, la sintassi è `help("fread")` o `?fread` . Se il pacchetto non è stato caricato, utilizzare il nome completo come `?data.table::fread` .

Sintassi e caratteristiche

Sintassi di base

`DT[where, select|update|do, by]` è usato per lavorare con le colonne di un data.table.

- La parte "dove" è l'argomento `i`
- La parte "select | update | do" è l'argomento `j`

Questi due argomenti vengono generalmente passati per posizione anziché per nome.

Una sequenza di passaggi può essere concatenata come `DT[...][...]` .

Scorciatoie, funzioni speciali e simboli speciali all'interno di `DT[...]`

Funzione o simbolo	Gli appunti
`.()`	in diversi argomenti, sostituisce `list()`

Funzione o simbolo	Gli appunti
`J()`	in `i` , sostituisce `list()`
`:=`	in `j` , una funzione utilizzata per aggiungere o modificare colonne
`.N`	in `i` , il numero totale di righe in `j` , il numero di righe in un gruppo
`.I`	in `j` , il vettore dei numeri di riga nella tabella (filtrato da `i`)
`.SD`	in `j` , il sottoinsieme corrente dei dati selezionato dall'argomento `.SDcols`
`.GRP`	in `j` , l'indice corrente del sottoinsieme dei dati
`.BY`	in `j` , l'elenco di valori per il sottoinsieme di dati corrente
`V1, V2, ...`	nomi predefiniti per colonne senza nome create in `j`

Entra a far parte di `DT[...]`

Notazione	Gli appunti
`DT1[DT2, on, j]`	unisciti a due tavoli
`i.*`	prefisso speciale sulle colonne DT2 dopo l'unione
`by=.EACHI`	opzione speciale disponibile solo con un join
`DT1[!DT2, on, j]`	anti-join due tabelle
`DT1[DT2, on, roll, j]`	unisciti a due tabelle, rotolando sull'ultima colonna in `on=`

Rimodellamento, accatastamento e divisione

Notazione	Gli appunti
`melt(DT, id.vars, measure.vars)`	trasformare in formato lungo per più colonne, usa `measure.vars = patterns(...)`
`dcast(DT, formula)`	trasformare in formato ampio
`rbind(DT1, DT2, ...)`	stack elencato data.tables
`rbindlist(DT_list, idcol)`	impila un elenco di dati.tables

Notazione	Gli appunti
`split(DT, by)`	dividere un data.table in una lista

Alcune altre funzioni sono specializzate per data.tables

Funzione (s)	Gli appunti
`foverlaps`	sovrapposizione di join
`merge`	un altro modo di unire due tavoli
`set`	un altro modo per aggiungere o modificare colonne
`fintersect` , `fsetdiff` , `funion` , `fsetequal` , `unique` , `duplicated` , `anyDuplicated`	operazioni di teoria delle serie con righe come elementi
`CJ`	il prodotto cartesiano di vettori
`uniqueN`	il numero di righe distinte
`rowidv(DT, cols)`	riga ID (da 1 a .N) all'interno di ciascun gruppo determinata da colonne
`rleidv(DT, cols)`	ID gruppo (da 1 a .GRP) all'interno di ciascun gruppo determinato da serie di colonne
`shift(DT, n)`	applica un operatore di turno a ogni colonna
`setorder` , `setcolorder` , `setnames` , `setkey` , `setindex` , `setattr`	modificare gli attributi e ordinare per riferimento

Altre caratteristiche del pacchetto

Caratteristiche	Gli appunti
`IDate` e `ITime`	numero intero di date e ore

Leggi Iniziare con data.table online: https://riptutorial.com/it/data-table/topic/3389/iniziare-con-data-table

Capitolo 2: Aggiunta e modifica di colonne

Osservazioni

La vignetta ufficiale, "Semantica di riferimento" , è la migliore introduzione a questo argomento.

Un promemoria: `DT[where, select|update|do, by]` è usato per lavorare con le colonne di un data.table.

- La parte "dove" è l'argomento `i`
- La parte "select | update | do" è l'argomento `j`

Questi due argomenti vengono generalmente passati per posizione anziché per nome.

Tutte le modifiche alle colonne possono essere fatte in `j` . Inoltre, la funzione `set` è disponibile per questo uso.

Examples

Modifica dei valori

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

Modifica di una colonna

Utilizza l'operatore `:=` all'interno di `j` per creare nuove colonne o modificare quelle esistenti:

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

Modifica su un sottoinsieme di righe

Utilizza l'argomento `i` per sottoporre a sotto le righe "dove" devono essere apportate modifiche:

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

Come in un data.frame, possiamo impostare sottoinsiemi usando numeri di riga o test logici. È anche possibile usare [un "join" in `i` quando si modifica] [need_a_link].

Rimozione di una colonna

Rimuovi colonne impostando su `NULL` :

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

Notare che non `<-` assegnare il risultato, poiché `DT` è stato modificato sul posto.

Modifica di più colonne

Aggiungere più colonne tramite `:=` formato multivariata dell'operatore:

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

La sintassi `.()` Viene utilizzata quando il lato destro di `LHS := RHS` è un elenco di colonne.

Modifica di più colonne dipendenti sequenzialmente

Se le colonne dipendono e devono essere definite in sequenza, alcuni modi per farlo sono:

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

Modifica di colonne con nomi determinati dinamicamente

Per i nomi di colonna determinati dinamicamente, utilizzare le parentesi:

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

Utilizzando `set`

Le colonne possono anche essere modificate con `set` per una piccola riduzione del sovraccarico, anche se questo è raramente necessario:

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

Riordinare le colonne

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

Per riorganizzare l'ordine delle colonne, usa `setcolorder` . Ad esempio, per invertirli

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

Questo non costa quasi nulla in termini di prestazioni, dal momento che sta solo permettendo l'elenco dei puntatori di colonna nel data.table.

Rinominare le colonne

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

Per rinominare una colonna (mantenendo i dati uguali), non è necessario copiare i dati in una colonna con un nuovo nome ed eliminare quello precedente. Invece, possiamo usare

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

per modificare la colonna originale per riferimento.

Modifica dei livelli dei fattori e altri attributi delle colonne

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

Per modificare i livelli dei fattori per riferimento, usa `setattr` :

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

La seconda opzione potrebbe stampare il risultato sullo schermo.

Con `setattr` , evitiamo la copia di solito incontrata durante i `levels(x) <- lvls` , ma salterà anche alcuni controlli, quindi è importante fare attenzione ad assegnare un vettore valido di livelli.

Leggi Aggiunta e modifica di colonne online: https://riptutorial.com/it/data-table/topic/3781/aggiunta-e-modifica-di-colonne

Capitolo 3: Calcolo delle statistiche di riepilogo

Osservazioni

Un promemoria: `DT[where, select|update|do, by]` è usato per lavorare con le colonne di un data.table.

- La parte "dove" è l'argomento `i`
- La parte "select | update | do" è l'argomento `j`

Questi due argomenti vengono generalmente passati per posizione anziché per nome.

Examples

Conteggio delle righe per gruppo

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

Utilizzando `.N`

`.N` in `j` memorizza il numero di righe in un sottoinsieme. Durante l'esplorazione dei dati, `.N` è utile per ...

1. contare le righe in un gruppo,

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

# 50
```

2. o contare le righe in tutti i gruppi,

```
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. o trova gruppi che hanno un certo numero di righe.

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

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

Gestione dei gruppi mancanti

Tuttavia, ci mancano gruppi con un conteggio pari a zero sopra. Se contano, possiamo usare la `table` dalla 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
```

In alternativa, possiamo partecipare a tutti i gruppi:

```
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
```

Una nota su `.N` :

- Questo esempio usa `.N` in `j` , dove si riferisce alla dimensione di un sottoinsieme.
- In `i` , si riferisce al numero totale di righe.

Riepiloghi personalizzati

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

Supponiamo di volere l'output della funzione di `summary` per `Sepal.Length` insieme al numero di osservazioni:

```
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
```

Dobbiamo fare `j` un elenco di colonne. Di solito, alcuni giocano con `c` , `as.list` e `.` è abbastanza per capire il modo corretto di procedere.

Assegnazione di statistiche di riepilogo come nuove colonne

Invece di creare una tabella di riepilogo, potremmo voler memorizzare una statistica riassuntiva in una nuova colonna. Possiamo usare `:=` come al solito. Per esempio,

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

insidie

Dati disordinati

Se ti trovi a voler analizzare i nomi delle colonne, ad esempio

> Prendi la media di `x.Length/x.Width` dove `x` prende dieci valori diversi.

quindi probabilmente stai guardando i dati incorporati nei nomi delle colonne, che è una cattiva idea. Leggi informazioni in ordine e poi rimodella in formato lungo.

Riepiloghi a riga

I frame di dati e i dati.tables sono ben progettati per i dati tabulari, dove le righe corrispondono a osservazioni e colonne a variabili. Se ti ritrovi a voler riassumere le righe, ad esempio

> Trova la deviazione standard tra le colonne per ogni riga.

allora dovresti probabilmente usare una matrice o qualche altro formato di dati interamente.

La funzione di riepilogo

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

`summary` è utile per consultare le statistiche di riepilogo. Oltre all'utilizzo diretto come `summary(DT)` , può anche essere applicato per gruppo comodamente con lo `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...]
```

Per includere i gruppi con conteggio zero, impostare `drop=FALSE` in `split` .

Applicazione di una funzione di riepilogo a più variabili

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

Per applicare la stessa funzione di riepilogo a ogni colonna per gruppo, possiamo usare `lapply` e `.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
```

Possiamo filtrare le colonne in `.SD` con l'argomento `.SDcols` :

```
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
```

Molteplici funzioni di riepilogo

Attualmente, l'estensione più semplice a più funzioni è forse:

```
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
```

Se desideri che i nomi siano come `Petal.Length.med` anziché `med.Petal.Length` , modifica l'ordine:

```
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
```

Leggi Calcolo delle statistiche di riepilogo online: https://riptutorial.com/it/data-table/topic/3785/calcolo-delle-statistiche-di-riepilogo

Capitolo 4: Creare un data.table

Osservazioni

Un data.table è una versione avanzata della classe data.frame dalla base R. Come tale, l'attributo `class()` è il vettore `"data.table" "data.frame"` e le funzioni che funzionano su un data.frame saranno anche funziona con un data.table. Ci sono molti modi per creare, caricare o forzare su un data.table, come visto qui.

Examples

Costruire un data.frame

Per copiare un data.frame come data.table, utilizzare `as.data.table` o `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)
```

Questo è raramente necessario. Un'eccezione è quando si utilizzano set di dati `mtcars` come `mtcars` , che devono essere copiati poiché non possono essere modificati sul posto.

Costruisci con data.table ()

C'è un costruttore con lo stesso nome:

```
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
```

A differenza di `data.frame` , `data.table` non costringerà le stringhe ai fattori per impostazione predefinita:

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

Leggi con fread ()

Possiamo leggere da un file di testo:

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

A differenza di `read.csv` , `fread` leggerà le stringhe come stringhe, non come fattori di default.

Vedi [argomento su `fread`] [need_a_link] per altri esempi.

Modifica un data.frame con setDT ()

Per efficienza, data.table offre un modo per modificare un data.frame o un elenco per creare un data.table sul posto:

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

# modification
setDT(DF)
```

Nota che non `<-` assegna il risultato, poiché l'oggetto `DF` è stato modificato sul posto.

Gli attributi di classe di data.frame verranno mantenuti:

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

Copia un altro data.table con copy ()

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

A causa del modo in cui i dati vengono manipolati, `DT2 <- DT1` *non* farà una copia. Cioè, le successive modifiche alle colonne o altri attributi di `DT2` influenzeranno anche `DT1` . Quando vuoi una copia reale, usa

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

Per vedere la differenza, ecco cosa succede senza una copia:

```
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
```

E con una copia:

```
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
```

Quindi le modifiche non si propagano in quest'ultimo caso.

Leggi Creare un data.table online: https://riptutorial.com/it/data-table/topic/3782/creare-un-data-table

Capitolo 5: Dati di pulizia

Examples

Gestione dei duplicati

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

Per gestire i "duplicati", combinare le righe di conteggio in un gruppo e inserire le righe per gruppo .

Mantieni una riga per gruppo

Aka "drop duplicates" alias "deduplicate" alias "uniquify".

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

Questo mantiene la prima riga. Per selezionare una riga diversa, si può giocare con la parte `1L` o usare l' `order` in `i` .

Mantieni solo righe univoche

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

Mantieni solo righe non univoche

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

Leggi Dati di pulizia online: https://riptutorial.com/it/data-table/topic/5206/dati-di-pulizia

Capitolo 6: Perché il mio vecchio codice non funziona?

introduzione

Il pacchetto data.table ha subito una serie di modifiche e innovazioni nel tempo. Ecco alcune potenziali insidie che possono aiutare gli utenti a consultare il codice legacy o a rivedere i vecchi post del blog.

Examples

unico e duplicato non funziona più su data.table con chiave

Questo è per chi si sposta su data.table> = 1.9.8

Hai un set di dati di proprietari e nomi di animali domestici, ma sospetti che siano stati catturati alcuni dati ripetuti.

```
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
```

Ricorda di digitare una tabella per ordinarla. Alice è stata inserita due volte.

```
> 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
```

Supponiamo che tu abbia usato `unique` per sbarazzarti dei duplicati nei tuoi dati in base alla chiave, utilizzando la data di acquisizione dati più recente impostando da Ultimo a VERO.

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
```

Il duplicato di Alice è stato rimosso.

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
```

Questo non funziona. Ancora 4 righe!

fissare

Utilizza il parametro `by=` **che non è più impostato di default sulla tua chiave** ma su tutte le colonne.

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

Ora va tutto bene.

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

Dettagli e correzione di stopgap

Vedere l' articolo 1 nelle note sulla versione di NEWS per i dettagli:

> Cambiamenti in v1.9.8 (su CRAN 25 Nov 2016)
>
> CAMBIAMENTI POTENZIALMENTE DI BREAKING
>
> 1. Per impostazione predefinita, tutte le colonne vengono ora utilizzate dai metodi `unique()` , `duplicated()` e `uniqueN()` data.table, # 1284 e # 1841. Per ripristinare il vecchio comportamento: `options(datatable.old.unique.by.key=TRUE)` . In 1 anno questa opzione per ripristinare il vecchio valore predefinito sarà deprecata con avviso. In 2 anni l'opzione verrà rimossa. Si prega di passare esplicitamente `by=key(DT)` per chiarezza. Solo il codice che si basa sull'impostazione predefinita è interessato. 266 pacchetti CRAN e Bioconductor che utilizza data.table sono stati controllati prima del rilascio. 9 necessari per cambiare e sono stati notificati.

Eventuali linee di codice senza copertura di prova saranno state perse da questi controlli. I pacchetti non su CRAN o Bioconductor non sono stati controllati.

Quindi puoi utilizzare le opzioni come soluzione temporanea finché il tuo codice non viene corretto.

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

Leggi Perché il mio vecchio codice non funziona? online: https://riptutorial.com/it/data-table/topic/8196/perche-il-mio-vecchio-codice-non-funziona-

Capitolo 7: Rimodellamento, accatastamento e divisione

Osservazioni

La bozza ufficiale, "Rimodellamento efficiente con data.tables" , è la migliore introduzione a questo argomento.

Molti compiti di rimodellamento richiedono lo spostamento tra formati lunghi e ampi:

- I dati ampi sono dati con ogni colonna che rappresenta una variabile separata e le righe che rappresentano le osservazioni separate
- I dati lunghi sono dati con l'ID del modulo | variabile | valore, dove ogni riga rappresenta una coppia di osservazioni-variabili

Examples

fondere e lanciare con data.table

`data.table` offre una vasta gamma di possibilità per rimodellare i tuoi dati in modo efficiente e semplice

Ad esempio, mentre si rimodella da lungo a largo è possibile passare diverse variabili nel `value.var` e nei parametri `fun.aggregate` allo stesso tempo

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

Da lungo a largo

```
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
```

Questo imposterà la `gear` come colonna dell'indice, mentre la `mean` e la `sum` saranno calcolate per `disp` e `hp` per ogni combinazione di `gear` e `cyl` . Nel caso in cui alcune combinazioni non esistano, è possibile specificare parametri aggiuntivi come `na.rm = TRUE` (che verrà passato per `mean` e `sum` funzioni) o specificare l'argomento di `fill` incorporato. È anche possibile aggiungere margini, eliminare combinazioni mancanti e sottoporre a subset i dati. Vedi di più in `?data.table::dcast`

Largo a lungo

Per quanto riguarda il rimodellamento da ampio a lungo, è possibile passare colonne al parametro `measure.vars` utilizzando le espressioni regolari, ad esempio

```
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
```

Questo `melt` i dati per `cyl` e `gear` come le colonne indice, mentre tutti i valori per le variabili che iniziano con `d` (`disp` & `drat`) saranno presenti in `value1` e i valori per le variabili che contengono la lettera `e` in essi (`qsec` e `gear`) saranno presenti nella colonna `value2` .

È inoltre possibile rinominare tutti i nomi di colonna nel risultato specificando `value.name` argomenti `variable.name` e `value.name` o decidere se si desidera che le colonne di `character` vengano automaticamente convertite in `factor` o senza specificare `value.factor` argomenti `variable.factor` e `value.factor` . Vedi di più in `?data.table::melt`

Risagoma usando `data.table`

`data.table` estende le `reshape2` di `melt` e `dcast`

(Riferimento: rimodellamento efficiente con 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
```

Da lungo a largo

```
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
```

Su più colonne (come `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
```

Da largo a lungo

```
## 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
```

Passando dal formato wide a long usando il fuso

Fusione: le basi

La fusione viene utilizzata per trasformare i dati da un formato ampio a uno lungo.

A partire da un ampio set di dati:

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

Possiamo fondere i nostri dati usando la funzione `melt` in data.table. Questo restituisce un altro data.table in formato lungo:

```
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"
```

Si presume che le colonne non impostate nel parametro `id.vars` siano variabili. In alternativa, possiamo impostarli esplicitamente usando l'argomento `measure.vars` :

```
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 questo caso, si presume che tutte le colonne non impostate in `measure.vars` siano ID.

Se impostiamo entrambi in modo esplicito, restituirà solo le colonne selezionate:

```
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
```

Denominazione di variabili e valori nel risultato

Possiamo manipolare i nomi delle colonne della tabella restituita usando `variable.name` e `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
```

Impostazione dei tipi per misurare le variabili nel risultato

Per impostazione predefinita, la fusione di un data.table converte tutti i valori da `measure.vars` a fattori:

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

Per impostare come carattere, utilizzare l'argomento `variable.factor` :

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

I valori generalmente ereditano dal tipo di dati della colonna di origine:

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

Se c'è un conflitto, i tipi di dati saranno forzati. Per esempio:

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

Quando si scioglie, qualsiasi variabile fattore sarà forzata al tipo di carattere:

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

Per evitare ciò e preservare la digitazione iniziale, utilizzare l'argomento `value.factor` :

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

Gestione dei valori mancanti

Per impostazione predefinita, tutti i valori `NA` vengono conservati nei dati fusi

```
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
```

Se questi devono essere rimossi dai tuoi dati, imposta `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
```

Passando dal formato lungo al formato wide usando dcast

Casting: The Basics

La trasmissione viene utilizzata per trasformare i dati dal formato lungo al formato wide.

A partire da un lungo set di dati:

```
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)
```

Possiamo `dcast` i nostri dati utilizzando la funzione dcast in data.table. Questo restituisce un altro data.table in formato ampio:

```
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 un valore

Un argomento `value.var` è necessario per un cast corretto - se non viene fornito dcast farà un'ipotesi basata sui tuoi dati.

```
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
```

Più `value.var` possono essere forniti in un elenco

```
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

La trasmissione è controllata utilizzando l'argomento formula in `dcast` . Questo è nella forma 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
```

Sia le righe che le colonne possono essere espanse con ulteriori variabili usando `+`

```
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
```

La fusione può spesso creare celle in cui non esiste alcuna osservazione nei dati. Di default questo è denotato da `NA` , come sopra. Possiamo sovrascriverlo con l'argomento `fill=` .

```
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
```

È inoltre possibile utilizzare due variabili speciali nell'oggetto formula

- . non rappresenta altre variabili
- . . . rappresenta tutte le altre variabili

```
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
```

Aggregazione del nostro valore.var

Possiamo anche eseguire il cast e aggregare i valori in un unico passaggio. In questo caso, abbiamo tre osservazioni in ciascuna delle intersezioni di Età e ID. Per impostare quale aggregazione vogliamo, usiamo l'argomento `fun.aggregate` :

```
#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
```

Possiamo anche passare un elenco a `fun.aggregate` per utilizzare più funzioni

```
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
```

Se passiamo più di una funzione e più di un valore, possiamo calcolare tutte le combinazioni passando un vettore di 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
```

dove ogni coppia è calcolata nell'ordine `value1_formula1, value1_formula2, ... , valueN_formula(N-1), valueN_formulaN .`

In alternativa, possiamo valutare i nostri valori e le funzioni uno a uno passando 'value.var' come lista:

```
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
```

Denominazione delle colonne nel risultato

Per impostazione predefinita, i componenti del nome della colonna sono separati da un carattere di sottolineatura _ . Questo può essere sovrascritto manualmente usando l'argomento `sep=` :

```
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
```

Questo `fun.aggregate` qualsiasi `fun.aggregate` o `value.var` che usiamo:

```
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
```

Impilare più tabelle usando rbindlist

Un ritornello comune in R segue queste linee:

> Non dovresti avere un gruppo di tabelle correlate con nomi come `DT1` , `DT2` , ..., `DT11` . Leggere e assegnare in modo iterativo agli oggetti per nome è disordinato. La soluzione è un elenco di tabelle di dati!

Un simile elenco sembra

```
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
```

Un'altra prospettiva è che dovresti memorizzare queste tabelle insieme *come una tabella* , impilandole. Questo è semplice da fare usando `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
```

Questo formato ha molto più senso con la sintassi data.table, dove le operazioni "per gruppo" sono comuni e semplici.

Per uno sguardo più approfondito, la risposta di Gregor potrebbe essere un buon punto di partenza. Controlla anche `?rbindlist` , ovviamente. C'è un esempio separato che riguarda la lettura in un gruppo di tabelle da CSV e quindi impilarle .

Leggi Rimodellamento, accatastamento e divisione online: https://riptutorial.com/it/data-table/topic/4117/rimodellamento--accatastamento-e-divisione

Capitolo 8: Si unisce e si fonde

introduzione

Un join combina due tabelle contenenti colonne correlate. Il termine copre una vasta gamma di operazioni, essenzialmente tutto tranne l' aggiunta delle due tabelle . "Unisci" è un sinonimo. Digitare `?`[.data.table`` per i documenti ufficiali.

Sintassi

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

Osservazioni

Lavorare con tabelle con chiave

Se `x` & `i` hanno una chiave o `x` è calettati per abbinare `i` 's prime colonne, allora la `on` possono essere ignorati come `x[i]` .

Nomi di colonne disambiguanti in comune

In `j` di `x[i, on, j]` , le colonne di `i` possono essere riferite con prefissi `i.*` .

Raggruppamento su sottoinsiemi

In `j` di `x[i, on, j, by=.EACHI]` , `j` viene calcolato per ogni riga di `i` .

Questo è l'unico valore `by` vale la pena utilizzare. Per qualsiasi altro valore, le colonne di `i` non sono disponibili.

Examples

Aggiorna valori in un join

Quando i dati sono "ordinati" , è spesso organizzato in più tabelle. Per combinare i dati per l'analisi, dobbiamo "aggiornare" una tabella con i valori di un'altra.

Ad esempio, potremmo avere dati di vendita per le prestazioni, in cui gli attributi dell'esecutore (il

loro budget) e della posizione (la sua popolazione) sono memorizzati in tabelle separate:

```
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
```

Quando siamo pronti per fare qualche analisi, dobbiamo prendere le variabili da queste altre tabelle:

```
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
```

Viene `mainDT` una `copy` per evitare di contaminare i dati grezzi, ma potremmo invece lavorare direttamente su `mainDT` .

Vantaggi nell'utilizzo di tabelle separate

I vantaggi di questa struttura sono trattati nel documento sui dati ordinati, ma in questo contesto:

1. *Tracciamento dei dati mancanti.* Solo le righe che corrispondono nell'unione ricevono un

compito. Non abbiamo dati per `geo_id == "MO"` sopra, quindi le sue variabili sono `NA` nella nostra tabella finale. Se vediamo inaspettatamente dati mancanti di questo tipo, possiamo risalire all'osservazione mancante nella tabella `geoDT` e indagare da lì se abbiamo un problema di dati che può essere risolto.

2. *Comprensibilità.* Nel costruire il nostro modello statistico, potrebbe essere importante tenere a mente che il `budget` è costante per ogni attore. In generale, capire la struttura dei dati paga i dividendi.

3. *Dimensione della memoria.* Potrebbe esserci un gran numero di attributi esecutore e posizione che non finiscono nel modello statistico. In questo modo, non è necessario includerli nella (possibilmente massiccia) tabella utilizzata per l'analisi.

Determinazione a livello di codice delle colonne

Se ci sono molte colonne in `pDT` , ma vogliamo solo selezionarne alcune, possiamo usarle

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

Le parentesi intorno `(p_cols) :=` sono essenziali, come indicato nel documento sulla creazione di colonne .

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
```

Intuizione

Pensa a `x[i]` come selezionando un sottoinsieme di `x` per ogni riga di `i` . Questa sintassi rispecchia il subset matrice in base R ed è coerente con il primo argomento che significa "dove", in `DT[where, select|update|do, by]` .

Ci si potrebbe chiedere perché valga la pena imparare questa nuova sintassi, dato che `merge(x,i)`

funziona ancora con data.tables. La risposta breve è che di solito vogliamo unire e quindi fare qualcosa di più. La sintassi `x[i]` cattura concisamente questo schema di utilizzo e consente anche un calcolo più efficiente. Per una spiegazione più dettagliata, leggere le domande frequenti 1.12 e 2.14 .

Gestione di righe con corrispondenza multipla

Per impostazione predefinita, viene restituita ogni riga di `a` corrispondenza di ogni riga di `b` :

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

Questo può essere ottimizzato con `mult` :

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

Gestione di righe non corrispondenti

Per impostazione predefinita, le righe senza precedenti di `a` ancora la loro comparsa nel risultato:

```
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
```

Per nasconderli, usa `nomatch` :

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

Si noti che `x[i]` tenterà di far corrispondere le NA in `i` .

Le partite di conteggio sono tornate

Per contare il numero di corrispondenze per ogni riga di `i` , utilizzare `.N` e `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
```

Leggi Si unisce e si fonde online: https://riptutorial.com/it/data-table/topic/4976/si-unisce-e-si-fonde

Capitolo 9: Subsetting di righe per gruppo

Osservazioni

Un promemoria: `DT[where, select|update|do, by]` è usato per lavorare con le colonne di un data.table.

- La parte "dove" è l'argomento `i`
- La parte "select | update | do" è l'argomento `j`

Questi due argomenti vengono generalmente passati per posizione anziché per nome.

Examples

Selezione di righe all'interno di ciascun gruppo

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

Supponiamo che, per ciascun sesso, desideriamo le righe con i numeri di sopravvivenza più alti:

```
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` è il sottoinsieme di dati associati a ciascun `Sex` ; e lo stiamo sottolineando ulteriormente, alle file che soddisfano la nostra condizione. Se la velocità è importante, usa invece un approccio suggerito da eddi su 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
```

insidie

Nell'ultima riga di codice, `.I` riferisce ai numeri di riga dell'intero data.table. Tuttavia, questo non è vero quando non c'è nessuno `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
```

Selezione di gruppi

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

Supponiamo di voler vedere solo la seconda classe:

```
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
```

Qui, semplicemente sommiamo i dati usando `i` , la clausola "where".

Selezione di gruppi in base alla condizione

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

Supponiamo di voler vedere ogni classe solo se sopravvive una maggioranza:

```
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
```

Qui, restituiamo il sottoinsieme di dati `.SD` solo se la nostra condizione è soddisfatta. Un'alternativa è

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

ma questo a volte si è dimostrato più lento.

Leggi Subsetting di righe per gruppo online: https://riptutorial.com/it/data-

table/topic/3784/subsetting-di-righe-per-gruppo

Capitolo 10: Utilizzo delle colonne elenco per memorizzare i dati

introduzione

Data.table supporta i vettori di colonna appartenenti alla classe di `list` di R.

Osservazioni

Nel caso in cui sembri strano che stiamo parlando di liste senza usare quella parola nel codice, nota che `.()` È un alias per `list()` se usato all'interno di una chiamata `DT[...]` .

Examples

Lettura in molti file correlati

Supponiamo di voler leggere e impilare un gruppo di file formattati in modo simile. La soluzione rapida è:

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

Potremmo non essere soddisfatti di questo per un paio di motivi:

- Potrebbe verificarsi errori durante la lettura con `fread` o quando si impila con `rbindlist` causa della formattazione dei dati incoerente o `rbindlist` .
- Potremmo voler tenere traccia dei metadati per ogni file, afferrati dal nome del file o forse da alcune righe di intestazione all'interno dei file (non del tutto tabulari).

Un modo per gestire ciò è creare una "tabella di file" e memorizzare il contenuto di ciascun file come una voce della colonna di elenco sulla riga associata.

Dati di esempio

Prima di inserire i dati di esempio qui sotto, assicurati di essere in una cartella vuota su cui scrivere. Esegui `getwd()` e leggi `?setwd` se hai bisogno di cambiare cartella.

```
# 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))
```

Identifica i file e i metadati dei file

Questa parte è abbastanza semplice:

```
# 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
```

Leggi nei file

Leggi nei file come una colonna di lista:

```
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>
```

Se c'è un intoppo nel leggere uno dei file o è necessario modificare gli argomenti `fread` seconda attributi del file, questo passaggio può essere facilmente esteso, guardando come:

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

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

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

Per i dettagli sulle opzioni di lettura in file CSV e file simili, vedere `?fread`

Impila i dati

Da qui, vogliamo impilare i dati:

```
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
```

Se si verificano alcuni problemi nello stacking (come i nomi delle colonne o le classi che non corrispondono), possiamo tornare alle singole tabelle in `fileDT` per verificare dove ha avuto origine il problema. Per esempio,

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

estensioni

Se i file non si trovano nella directory di lavoro corrente, utilizzare

```
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))))]
```

Leggi Utilizzo delle colonne elenco per memorizzare i dati online: https://riptutorial.com/it/data-table/topic/4456/utilizzo-delle-colonne-elenco-per-memorizzare-i-dati

Capitolo 11: Utilizzo di .SD e .SDcols per il sottoinsieme di dati

introduzione

Il simbolo speciale `.SD` è disponibile in `j` di `DT[i,j,by]` , catturando l'ubset **S** di **D** ata per ciascun `by` gruppo di superstiti del filtro, `i` . `.SDcols` è un aiuto. Digita `?`special-symbols`` per i documenti ufficiali.

Osservazioni

Un promemoria: `DT[where, select|update|do, by]` è usato per lavorare con le colonne di un data.table.

- La parte "dove" è l'argomento `i`
- La parte "select | update | do" è l'argomento `j`

Questi due argomenti vengono generalmente passati per posizione anziché per nome.

Examples

Utilizzando .SD e .SDcols

.SD

`.SD` fa riferimento al sottoinsieme di `data.table` per ciascun gruppo, escludendo tutte le colonne utilizzate `by` .

`.SD` insieme a `lapply` può essere utilizzato per applicare qualsiasi funzione a più colonne per gruppo in un `data.table`

Continueremo a utilizzare lo stesso set di dati `mtcars` , `mtcars` :

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

Media di tutte le colonne nel set di dati per *numero di cilindri* , `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
```

Oltre a `cyl` , ci sono altre colonne categoriali nel set di dati come `vs` , `am` , `gear` e `carb` . Non ha senso prendere la `mean` di queste colonne. Quindi escludiamo queste colonne. Questo è dove `.SDcols` entra nella foto.

.SDcols

`.SDcols` specifica le colonne di `data.table` che sono incluse in `.SD` .

Media di tutte le colonne (colonne continue) dell'insieme di dati di *numero di marce* `gear` , e *il numero di cilindri,* `cyl` , disposti da `gear` e `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
```

Forse non vogliamo calcolare la `mean` per gruppi. Per calcolare la media per tutte le auto nel set di dati, non specifichiamo la variabile `by` .

```
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
```

Nota: non è necessario definire prima `cols_chosen` . `.SDcols` può prendere direttamente i nomi delle colonne

Leggi Utilizzo di .SD e .SDcols per il sottoinsieme di dati online: https://riptutorial.com/it/data-table/topic/3787/utilizzo-di--sd-e--sdcols-per-il-sottoinsieme-di-dati

Capitolo 12: Utilizzo di chiavi e indici

introduzione

La chiave e gli indici di un data.table consentono a determinati calcoli di funzionare più velocemente, principalmente in relazione a join e sottoinsiemi. La chiave descrive l'ordinamento corrente della tabella; mentre ogni indice memorizza le informazioni sull'ordine della tabella rispetto a una sequenza di colonne. Vedere la sezione "Osservazioni" sotto per i collegamenti alle vignette ufficiali sull'argomento.

Osservazioni

Le vignette ufficiali sono la migliore introduzione a questo argomento:

- "Sottoinsieme di chiavi e ricerca binaria veloce"
- "Indici secondari e indicizzazione automatica"

Chiavi contro indici

Un data.table può essere "digitato" da una sequenza di colonne, indicando le funzioni interessate che i dati sono ordinati da tali colonne. Per ottenere o impostare la chiave, utilizzare le funzioni documentate al `?key`

Allo stesso modo, le funzioni possono trarre vantaggio dagli "indici" di data.table. Ogni indice e una tabella possono avere più di uno: memorizza le informazioni sull'ordine dei dati rispetto a una sequenza di colonne. Come una chiave, un indice può accelerare determinati compiti. Per ottenere o impostare indici, utilizzare le funzioni documentate in `?indices` .

Gli indici possono anche essere impostati automaticamente (attualmente solo per una singola colonna alla volta). Vedi `?datatable.optimize` per i dettagli su come funziona e su come disabilitarlo se necessario.

Verifica e aggiornamento

I valori mancanti sono consentiti in una colonna chiave.

Le chiavi e gli indici sono memorizzati come attributi e potrebbero, per errore, non corrispondere all'ordine effettivo dei dati nella tabella. Molte funzioni verificano la validità della chiave o dell'indice prima di usarlo, ma vale la pena tenerlo a mente.

Le chiavi e gli indici vengono rimossi dopo gli aggiornamenti, laddove non è ovvio che l'ordinamento sia preservato. Ad esempio, a partire da `DT = data.table(a=c(1,2,4), key="a")` , se aggiorniamo come `DT[2, a := 3]` , la chiave è rotta.

Examples

Miglioramento delle prestazioni per la selezione di sottoinsiemi

```
# 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)
)
```

Corrispondenza su una colonna

Dopo la prima esecuzione di un'operazione di subsetting con `==` o `%in%` ...

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

Un indice è stato creato automaticamente per `g1` . Le successive operazioni di subsetting avvengono quasi istantaneamente:

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

Per monitorare quando viene creato o utilizzato un indice, aggiungere l'opzione `verbose=TRUE` o modificare le `options(datatable.verbose=TRUE)` impostazione globale `options(datatable.verbose=TRUE)` .

Corrispondenza su più colonne

Attualmente, la corrispondenza su due colonne non crea automaticamente un indice:

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

Rieseguire questo e rimarrà lento. Anche se aggiungiamo manualmente l'indice con `setindex(DT,`

`g1, g2)` , rimarrà lento perché questa query non è ancora ottimizzata dal pacchetto.

Fortunatamente, se possiamo enumerare le combinazioni di valori che vogliamo cercare e un indice è disponibile, possiamo equi-join rapidamente:

```
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
```

Con `CJ` , è importante prestare attenzione al numero di combinazioni che diventano troppo grandi.

Leggi Utilizzo di chiavi e indici online: https://riptutorial.com/it/data-table/topic/4977/utilizzo-di-chiavi-e-indici

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con data.table	Community, Frank, micstr
2	Aggiunta e modifica di colonne	eddi, Frank, jangorecki, micstr
3	Calcolo delle statistiche di riepilogo	Frank
4	Creare un data.table	Chris, Frank
5	Dati di pulizia	Frank
6	Perché il mio vecchio codice non funziona?	Frank, micstr
7	Rimodellamento, accatastamento e divisione	Chris, David Arenburg, Frank, SymbolixAU
8	Si unisce e si fonde	Chris, Frank
9	Subsetting di righe per gruppo	Frank, micstr
10	Utilizzo delle colonne elenco per memorizzare i dati	Frank
11	Utilizzo di .SD e .SDcols per il sottoinsieme di dati	Frank
12	Utilizzo di chiavi e indici	Frank