eBook Gratuit

APPRENEZ data.table

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#data.table

Table des matières

À propos

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.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

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

Chapitre 1: Démarrer avec data.table

Remarques

Data.table est un package pour l'environnement informatique statistique R. Il étend la fonctionnalité des trames de données à partir de la base R, améliorant particulièrement leurs performances et leur syntaxe. Un certain nombre de tâches connexes, y compris les jointures roulantes et non-égales, sont gérées dans une syntaxe concise et cohérente, telle que `DT[where, select|update|do, by]` .

Un certain nombre de fonctions complémentaires sont également incluses dans le package:

- I / O: `fread` / `fwrite`
- Remodelage: `melt` / `dcast` / `rbindlist` / `split`
- `rleid` de valeurs: `rleid`

Versions

Version	Remarques	Date de sortie sur CRAN
1.9.4		2014-10-02
1.9.6		2015-09-19
1.9.8		2016-11-24
1.10.0	"Avec le recul, la dernière version v1.9.8 aurait dû être nommée v1.10.0"	2016-12-03
1.10.1	En développement	2016-12-03

Examples

Installation et configuration

Installez la version stable de CRAN:

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

Ou la version de développement de github:

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

Pour revenir de devel à CRAN, la version actuelle doit d'abord être supprimée:

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

Visitez le site Web pour obtenir les instructions d'installation complètes et les derniers numéros de version.

Utiliser le paquet

Habituellement, vous voudrez charger le paquet et toutes ses fonctions avec une ligne comme

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

Si vous n'avez besoin que d'une ou deux fonctions, vous pouvez vous y référer comme `data.table::fread` .

Démarrer et trouver de l'aide

Le wiki officiel du paquet contient des matériaux essentiels:

- En tant que nouvel utilisateur, vous souhaiterez consulter les vignettes, les FAQ et l'intercalaire .
- Avant de poser une question - ici sur StackOverflow ou ailleurs - veuillez lire la page de support .

Pour obtenir de l'aide sur des fonctions individuelles, la syntaxe est `help("fread")` ou `?fread` . Si le paquet n'a pas été chargé, utilisez le nom complet comme `?data.table::fread` .

Syntaxe et fonctionnalités

Syntaxe de base

`DT[where, select|update|do, by]` syntaxe `DT[where, select|update|do, by]` est utilisée pour travailler avec des colonnes d'un data.table.

- La partie "where" est l'argument `i`
- La partie "select | update | do" est l'argument `j`

Ces deux arguments sont généralement passés par position plutôt que par nom.

Une séquence d'étapes peut être chaînée comme `DT[...][...]` .

Raccourcis, fonctions spéciales et symboles

spéciaux dans `DT[...]`

Fonction ou symbole	Remarques
`.()`	dans plusieurs arguments, remplace la `list()`
`J()`	dans `i` , remplace la `list()`
`:=`	en `j` , une fonction utilisée pour ajouter ou modifier des colonnes
`.N`	dans `i` , le nombre total de lignes en `j` , le nombre de lignes d'un groupe
`.I`	dans `j` , le vecteur des numéros de ligne dans le tableau (filtré par `i`)
`.SD`	en `j` , le sous-ensemble actuel des données sélectionné par l'argument `.SDcols`
`.GRP`	dans `j` , l'index actuel du sous-ensemble des données
`.BY`	dans `j` , la liste des sous-valeurs pour le sous-ensemble de données actuel
`V1, V2, ...`	noms par défaut pour les colonnes sans nom créées dans `j`

Se joint à `DT[...]`

Notation	Remarques
`DT1[DT2, on, j]`	joindre deux tables
`i.*`	préfixe spécial sur les colonnes de DT2 après la jointure
`by=.EACHI`	option spéciale disponible uniquement avec une jointure
`DT1[!DT2, on, j]`	anti-joint deux tables
`DT1[DT2, on, roll, j]`	joindre deux tables, en roulant sur la dernière colonne dans `on=`

Remodelage, empilement et fractionnement

Notation	Remarques
`melt(DT, id.vars, measure.vars)`	transformer en format long pour plusieurs colonnes, utilisez `measure.vars = patterns(...)`
`dcast(DT, formula)`	transformer en format large
`rbind(DT1, DT2, ...)`	Data.tables énumérées par pile
`rbindlist(DT_list, idcol)`	empiler une liste de data.tables
`split(DT, by)`	diviser un data.table en une liste

Quelques autres fonctions spécialisées pour data.tables

Les fonctions)	Remarques
`foverlaps`	chevauchement des jointures
`merge`	une autre façon de joindre deux tables
`set`	une autre façon d'ajouter ou de modifier des colonnes
`fintersect` , `fsetdiff` , `funion` , `fsetequal` , `unique` , `duplicated` , `anyDuplicated`	opérations de théorie des ensembles avec des lignes en tant qu'éléments
`CJ`	le produit cartésien de vecteurs
`uniqueN`	le nombre de lignes distinctes
`rowidv(DT, cols)`	ID de ligne (1 à .N) dans chaque groupe déterminé par les cols
`rleidv(DT, cols)`	ID de groupe (1 à .GRP) dans chaque groupe déterminé par des exécutions de cols
`shift(DT, n)`	appliquer un opérateur de décalage à chaque colonne
`setorder` , `setcolorder` , `setnames` , `setkey` , `setindex` , `setattr`	modifier les attributs et ordonner par référence

Autres caractéristiques du package

Caractéristiques	Remarques
`IDate` et `ITime`	dates et heures entières

Lire Démarrer avec data.table en ligne: https://riptutorial.com/fr/data-table/topic/3389/demarrer-avec-data-table

Chapitre 2: Ajout et modification de colonnes

Remarques

La vignette officielle, "Sémantique de référence" , est la meilleure introduction à ce sujet.

Un rappel: la syntaxe `DT[where, select|update|do, by]` est utilisée pour travailler avec des colonnes d'un data.table.

- La partie "where" est l'argument `i`
- La partie "select | update | do" est l'argument `j`

Ces deux arguments sont généralement passés par position plutôt que par nom.

Toutes les modifications apportées aux colonnes peuvent être effectuées en `j` . De plus, la fonction `set` est disponible pour cette utilisation.

Examples

Modification des valeurs

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

Modifier une colonne

Utilisez l'opérateur `:=` dans `j` pour créer de nouvelles colonnes ou modifier celles existantes:

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

Modification sur un sous-ensemble de lignes

Utilisez l'argument `i` pour créer un sous-ensemble des lignes "où" des modifications doivent être apportées:

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

Comme dans un data.frame, nous pouvons sous-utiliser des numéros de lignes ou des tests logiques. Il est également possible d'utiliser [une "jointure" dans `i` lors de la modification] [need_a_link].

Supprimer une colonne

Supprimer les colonnes en définissant la `NULL` sur `NULL` :

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

Notez que nous ne `<-` affectons pas le résultat, car `DT` a été modifié sur place.

Modification de plusieurs colonnes

Ajoutez plusieurs colonnes en utilisant le format multivarié de l'opérateur `:=`

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

La syntaxe `.()` Est utilisée lorsque le côté droit de `LHS := RHS` est une liste de colonnes.

Modification de plusieurs colonnes dépendantes de manière séquentielle

Si les colonnes sont dépendantes et doivent être définies en séquence, vous pouvez procéder de la manière suivante:

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

Modification de colonnes par noms déterminés dynamiquement

Pour les noms de colonnes déterminés dynamiquement, utilisez des parenthèses:

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

En utilisant `set`

Les colonnes peuvent également être modifiées en `set` une petite réduction du temps système,

bien que cela soit rarement nécessaire:

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

Réorganisation des colonnes

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

Pour réorganiser l'ordre des colonnes, utilisez `setcolorder` . Par exemple, pour les inverser

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

Cela ne coûte presque rien en termes de performances, car il ne fait que permuter la liste des pointeurs de colonnes dans le fichier data.table.

Renommer des colonnes

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

Pour renommer une colonne (tout en conservant les mêmes données), il est inutile de copier les données dans une colonne sous un nouveau nom et de supprimer l'ancienne. Au lieu de cela, nous pouvons utiliser

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

modifier la colonne d'origine par référence.

Modification des niveaux de facteur et des autres attributs de colonne

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

Pour modifier les niveaux de facteur par référence, utilisez `setattr` :

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

La deuxième option peut imprimer le résultat à l'écran.

Avec `setattr` , nous évitons la copie généralement encourue lors de l' `levels(x) <- lvls` , mais nous `levels(x) <- lvls` également certaines vérifications, il est donc important de faire attention à assigner un vecteur de niveaux valide.

Lire Ajout et modification de colonnes en ligne: https://riptutorial.com/fr/data-table/topic/3781/ajout-

et-modification-de-colonnes

Chapitre 3: Calculer des statistiques sommaires

Remarques

Un rappel: la syntaxe `DT[where, select|update|do, by]` est utilisée pour travailler avec des colonnes d'un data.table.

- La partie "where" est l'argument `i`
- La partie "select | update | do" est l'argument `j`

Ces deux arguments sont généralement passés par position plutôt que par nom.

Examples

Compter les lignes par groupe

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

En utilisant `.N`

`.N` dans `j` stocke le nombre de lignes dans un sous-ensemble. Lors de l'exploration des données, `.N` est pratique pour ...

1. compter les lignes dans un groupe,

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

# 50
```

2. ou compter les lignes dans tous les groupes,

```
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. ou trouvez des groupes qui ont un certain nombre de lignes.

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

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

Manipulation des groupes manquants

Cependant, nous manquons de groupes avec un compte de zéro ci-dessus. S'ils sont importants, nous pouvons utiliser la `table` de 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
```

Alternativement, nous pouvons rejoindre tous les groupes:

```
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
```

Une note sur `.N` :

- Cet exemple utilise `.N` dans `j` , où il fait référence à la taille d'un sous-ensemble.
- Dans `i` , il s'agit du nombre total de lignes.

Résumés personnalisés

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

Supposons que nous voulons la sortie de la fonction de `summary` pour `Sepal.Length` avec le nombre d'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
```

Nous devons faire `j` une liste de colonnes. Habituellement, certains jouent avec `c` , `as.list` et `.` est suffisant pour trouver la bonne façon de procéder.

Affectation de statistiques récapitulatives sous forme de nouvelles colonnes

Au lieu de créer un tableau récapitulatif, nous pouvons souhaiter stocker une statistique récapitulative dans une nouvelle colonne. Nous pouvons utiliser `:=` comme d'habitude. Par exemple,

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

Pièges

Données désordonnées

Si vous vous trouvez à vouloir analyser les noms de colonnes, comme

> Prenez la moyenne de `x.Length/x.Width` où `x` prend dix valeurs différentes.

alors vous regardez probablement des données incorporées dans des noms de colonne, ce qui est une mauvaise idée. Lisez à propos des données bien rangées , puis modifiez-les au format long.

Résumés en lignes

Les blocs de données et les data.tables sont bien conçus pour les données tabulaires, où les lignes correspondent aux observations et aux colonnes aux variables. Si vous vous trouvez à vouloir résumer sur des lignes, comme

> Recherchez l'écart type entre les colonnes pour chaque ligne.

alors vous devriez probablement utiliser une matrice ou un autre format de données entièrement.

La fonction récapitulative

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

`summary` est pratique pour parcourir les statistiques sommaires. Outre l'utilisation directe comme `summary(DT)`

, il est également possible de l'appliquer par groupe de manière pratique avec la `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...]
```

Pour inclure des groupes à nombre de zéro, définissez `drop=FALSE` dans `split` .

Application d'une fonction de résumé à plusieurs variables

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

Pour appliquer la même fonction récapitulant à chaque colonne par groupe, nous pouvons utiliser `lapply` et `.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
```

Nous pouvons filtrer les colonnes dans `.SD` avec l'argument `.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
```

Fonctions de synthèse multiples

Actuellement, l'extension la plus simple à plusieurs fonctions est peut-être:

```
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
```

Si vous voulez que les noms soient comme `Petal.Length.med` au lieu de `med.Petal.Length` , changez l'ordre:

```
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
```

Lire Calculer des statistiques sommaires en ligne: https://riptutorial.com/fr/data-table/topic/3785/calculer-des-statistiques-sommaires

Chapitre 4: Créer un data.table

Remarques

Un data.table est une version améliorée de la classe data.frame de la base R. En tant que tel, son attribut `class()` est le vecteur `"data.table" "data.frame"` et les fonctions qui fonctionnent sur un data.frame seront également travailler avec un data.table. Il existe de nombreuses manières de créer, charger ou contraindre à une data.table, comme on le voit ici.

Examples

Contraindre un data.frame

Pour copier un data.frame en tant que data.table, utilisez `as.data.table` ou `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)
```

Ceci est rarement nécessaire. Une exception est l'utilisation de jeux de données `mtcars` tels que `mtcars` , qui doivent être copiés car ils ne peuvent pas être modifiés sur place.

Construire avec data.table ()

Il y a un constructeur du même nom:

```
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
```

Contrairement à `data.frame` , `data.table` ne contraindra pas les chaînes aux facteurs par défaut:

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

Lire avec fread ()

Nous pouvons lire un fichier texte:

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

Contrairement à `read.csv` , `fread` lira les chaînes comme des chaînes, et non comme des facteurs par défaut.

Voir le [topic on `fread`] [need_a_link] pour plus d'exemples.

Modifier un data.frame avec setDT ()

Pour plus d'efficacité, data.table offre un moyen de modifier un data.frame ou une liste pour créer un data.table sur place:

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

# modification
setDT(DF)
```

Notez que nous ne `<-` assignons pas le résultat, puisque l'objet `DF` a été modifié sur place.

Les attributs de classe du data.frame seront conservés:

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

Copier un autre data.table avec copy ()

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

En raison de la manière dont les data.tables sont manipulées, `DT2 <- DT1` *ne* fera *pas de* copie. Autrement dit, les modifications ultérieures apportées aux colonnes ou à d'autres attributs de `DT2` affecteront également `DT1` . Lorsque vous voulez une copie réelle, utilisez

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

Pour voir la différence, voici ce qui se passe sans copie:

```
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
```

Et avec une copie:

```
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
```

Les modifications ne se propagent donc pas dans ce dernier cas.

Lire Créer un data.table en ligne: https://riptutorial.com/fr/data-table/topic/3782/creer-un-data-table

Chapitre 5: Données de nettoyage

Examples

Gestion des doublons

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

Pour traiter les "doublons", combinez les lignes de comptage dans un groupe et les sous-ensembles de lignes par groupe .

Garder une ligne par groupe

Aka "drop duplicates" aka "dédupliquer" aka "uniquify".

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

Cela conserve la première rangée. Pour sélectionner une autre ligne, on peut manipuler la partie `1L` ou utiliser l' `order` dans `i` .

Ne conservez que des lignes uniques

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

Ne conservez que des rangées non uniques

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

Lire Données de nettoyage en ligne: https://riptutorial.com/fr/data-table/topic/5206/donnees-de-

nettoyage

Chapitre 6: Joint et fusionne

Introduction

Une jointure combine deux tables contenant des colonnes associées. Le terme couvre un large éventail d'opérations, essentiellement tout sauf les deux tableaux . "Fusionner" est un synonyme. Tapez `?`[.data.table`` pour les documents officiels.

Syntaxe

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

Remarques

Travailler avec des tables à clés

Si `x` et `i` ont une clé ou que `x` est associé aux premières colonnes de `i` , alors `on` peut ignorer la fonction `x[i]` .

Désactiver les noms de colonnes en commun

En `j` de `x[i, on, j]` , les colonnes de `i` peuvent être référencées avec les préfixes `i.*` .

Regroupement sur des sous-ensembles

En `j` de `x[i, on, j, by=.EACHI]` , `j` est calculé pour chaque ligne de `i` .

C'est la seule valeur de `by` vaut la peine. Pour toute autre valeur, les colonnes de `i` ne sont pas disponibles.

Examples

Mettre à jour les valeurs dans une jointure

Lorsque les données sont "rangées", elles sont souvent organisées en plusieurs tables. Pour combiner les données à analyser, nous devons "mettre à jour" une table avec les valeurs d'une autre.

Par exemple, nous pouvons avoir des données de vente pour les performances, où les attributs de l'exécutant (leur budget) et de l'emplacement (sa population) sont stockés dans des tables séparées:

```
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
```

Lorsque nous sommes prêts à faire des analyses, nous devons récupérer les variables de ces autres 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
```

Une `copy` est prise pour éviter de contaminer les données brutes, mais nous pourrions travailler directement sur `mainDT` place.

Avantages de l'utilisation de tables séparées

Les avantages de cette structure sont traités dans le document sur les données bien rangées,

mais dans ce contexte:

1. *Traçage des données manquantes* Seules les lignes correspondant à la fusion reçoivent une affectation. Nous n'avons pas de données pour `geo_id == "MO"` ci-dessus, donc ses variables sont `NA` dans notre table finale. Si nous voyons des données manquantes comme celle-ci de manière inattendue, nous pouvons remonter à l'observation manquante dans le tableau `geoDT` et rechercher à partir de là si nous avons un problème de données qui peut être résolu.

2. *Compréhensibilité.* En construisant notre modèle statistique, il peut être important de garder à l'esprit que le `budget` est constant pour chaque interprète. En général, la compréhension de la structure des données porte ses fruits.

3. *Taille mémoire.* Il peut y avoir un grand nombre d'attributs interprètes et localisés qui ne se retrouvent pas dans le modèle statistique. De cette façon, nous n'avons pas besoin de les inclure dans la table (éventuellement massive) utilisée pour l'analyse.

Détermination par programme des colonnes

S'il y a beaucoup de colonnes dans `pDT` , mais que nous voulons seulement en sélectionner quelques-unes, nous pouvons utiliser

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

Les parenthèses autour de `(p_cols) :=` sont essentielles, comme indiqué dans la documentation sur la création de colonnes .

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

Pensez à `x[i]` en sélectionnant un sous-ensemble de `x` pour chaque ligne de `i` . Cette syntaxe reflète la matrice subsing dans la base R et est cohérente avec le premier argument signifiant

"où", dans `DT[where, select|update|do, by]` .

On peut se demander pourquoi cette nouvelle syntaxe mérite d'être apprise, puisque la `merge(x,i)` fonctionne toujours avec data.tables. La réponse courte est que nous voulons généralement fusionner et ensuite faire quelque chose de plus. La syntaxe `x[i]` capture de manière concise ce modèle d'utilisation et permet également un calcul plus efficace. Pour une explication plus détaillée, lisez la FAQ 1.12 et 2.14 .

Gestion des lignes à plusieurs correspondances

Par défaut, chaque ligne d' `a` correspondant de chaque ligne `b` est renvoyée:

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

Cela peut être modifié avec `mult` :

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

Gestion des lignes sans correspondance

Par défaut, les lignes non appariées d' `a` apparaissent dans le résultat:

```
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
```

Pour les masquer, utilisez `nomatch` :

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

Notez que `x[i]` tentera de faire correspondre les NA dans `i` .

Comptage des correspondances renvoyées

Pour compter le nombre de correspondances pour chaque ligne de `i` , utilisez `.N` et `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
```

Lire Joint et fusionne en ligne: https://riptutorial.com/fr/data-table/topic/4976/joint-et-fusionne

Chapitre 7: Pourquoi mon ancien code ne fonctionne pas?

Introduction

Le package data.table a subi de nombreuses modifications et innovations au fil du temps. Voici quelques pièges potentiels qui peuvent aider les utilisateurs à consulter le code existant ou à consulter les anciens articles de blog.

Examples

unique et dupliqué ne fonctionne plus sur data.table à clé

Ceci est pour ceux qui passent à data.table> = 1.9.8

Vous avez un ensemble de données de propriétaires et de noms d'animaux, mais vous pensez que certaines données répétées ont été capturées.

```
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
```

Rappelez-vous de taper une table pour la trier. Alice a été entrée deux fois.

```
> 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
```

Supposons que vous ayez utilisé `unique` méthode `unique` pour supprimer les doublons dans vos données en fonction de la clé, en utilisant la date de capture de données la plus récente en définissant 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 en double a été supprimée.

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
```

Cela ne fonctionne pas. Encore 4 lignes!

Réparer

Utilisez le paramètre `by=` **qui n'est plus par défaut pour votre clé** mais pour toutes les colonnes.

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

Maintenant tout va bien.

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

Détails et correction provisoire

Voir le point 1 des notes de version NEWS pour plus de détails:

> Changements dans la v1.9.8 (sur CRAN 25 nov. 2016)
>
> CHANGEMENT POTENTIEL DE CHANGEMENTS
>
> 1. Par défaut, toutes les colonnes sont maintenant utilisées par les méthodes `unique()` , `duplicated()` et `uniqueN()` data.table, # 1284 et # 1841. Pour restaurer l'ancien comportement: `options(datatable.old.unique.by.key=TRUE)` . En 1 an, cette option de restauration de l'ancienne valeur par défaut sera déconseillée avec un avertissement. En 2 ans, l'option sera supprimée. Veuillez passer explicitement `by=key(DT)` pour plus de clarté. Seul le code qui repose sur la valeur par défaut est affecté. 266 modules CRAN et Bioconductor utilisant data.table ont été vérifiés avant la publication. 9 devaient changer et ont été notifiés. Toutes les

lignes de code sans couverture de test auront été manquées par ces vérifications. Tous les paquets qui ne sont pas sur CRAN ou Bioconductor n'ont pas été vérifiés.

Vous pouvez donc utiliser les options comme solution temporaire jusqu'à ce que votre code soit corrigé.

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

Lire Pourquoi mon ancien code ne fonctionne pas? en ligne: https://riptutorial.com/fr/data-table/topic/8196/pourquoi-mon-ancien-code-ne-fonctionne-pas-

Chapitre 8: Remodelage, empilement et fractionnement

Remarques

La vignette officielle, "Réorganisation efficace à l'aide de data.tables" , est la meilleure introduction à ce sujet.

De nombreuses tâches de remodelage nécessitent un déplacement entre des formats longs et larges:

- Les données larges sont des données, chaque colonne représentant une variable distincte et les lignes représentant des observations séparées
- Les données longues sont des données avec le formulaire ID | variable | valeur, où chaque ligne représentant une paire de variables d'observation

Examples

fondre et couler avec data.table

`data.table` offre un large éventail de possibilités pour remodeler vos données à la fois efficacement et facilement

Par exemple, en remodelant de long en large, vous pouvez à la fois passer plusieurs variables dans le `value.var` et les paramètres `fun.aggregate` en même temps

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

Long à large

```
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
```

Cela va définir l' `gear` comme la colonne d'index, tandis que la `mean` et la `sum` seront calculées pour `disp` et `hp` pour chaque combinaison de `gear` et `cyl` . Si certaines combinaisons n'existent pas, vous pouvez spécifier des paramètres supplémentaires tels que `na.rm = TRUE` (qui seront transmis aux fonctions `mean` et `sum`) ou spécifier l'argument de `fill` intégré. Vous pouvez également ajouter des marges, supprimer des combinaisons manquantes et sous-ensemble les données. Voir plus dans

```
?data.table::dcast
```

Large à long

Tout en remodelant de long en long, vous pouvez passer des colonnes au paramètre `measure.vars` aide d'expressions régulières, par exemple

```
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
```

Cela fera `melt` les données par `cyl` et `gear` comme les colonnes d'index, alors que toutes les valeurs pour les variables qui commencent par `d` (`disp` & `drat`) seront présentes dans `value1` et les valeurs pour les variables qui contiennent la lettre `e` dedans (`qsec` et `gear`) seront présents dans la colonne `value2` .

Vous pouvez également renommer tous les noms de colonne du résultat en spécifiant `variable.name` arguments `variable.name` et `value.name` ou décider si vous souhaitez que les colonnes de `character` soient automatiquement converties en `factor` s ou en spécifiant `variable.factor` arguments `variable.factor` et `value.factor` . Voir plus dans `?data.table::melt`

Redéfinir en utilisant `data.table`

`data.table` étend les `reshape2` de `melt` et de `dcast`

(Référence: Remodelage efficace à l'aide de 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 à large

```
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
```

Sur plusieurs colonnes (à partir de `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
```

Large à 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
```

Fusion: les bases

La fusion est utilisée pour transformer les données du format large au format long.

En commençant par un large ensemble de données:

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

Nous pouvons fondre nos données en utilisant la fonction `melt` dans data.table. Cela retourne un autre data.table au format long:

```
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"
```

Toutes les colonnes non définies dans le paramètre `id.vars` sont supposées être des variables. Alternativement, nous pouvons les définir explicitement à l'aide de l'argument `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
```

Dans ce cas, toutes les colonnes non définies dans `measure.vars` sont supposées être des identifiants.

Si nous définissons les deux explicitement, il ne retournera que les colonnes sélectionnées:

```
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
```

Nommer les variables et les valeurs dans le résultat

Nous pouvons manipuler les noms de colonne de la table renvoyée en utilisant `variable.name` et `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
```

Définition des types de variables de mesure dans le résultat

Par défaut, la fusion d'une data.table convertit tous les `measure.vars` en facteurs:

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

Pour définir en tant que caractère, utilisez l'argument `variable.factor` :

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

Les valeurs héritent généralement du type de données de la colonne d'origine:

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

En cas de conflit, les types de données seront forcés. Par exemple:

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

Lors de la fusion, toute variable de facteur sera contrainte au type de caractère:

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

Pour éviter cela et conserver la saisie initiale, utilisez l'argument `value.factor` :

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

Manipulation des valeurs manquantes

Par défaut, toutes les valeurs `NA` sont conservées dans les données fondues

```
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
```

Si ceux-ci doivent être supprimés de vos données, définissez `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
```

Passer du format long au format large en utilisant dcast

Casting: les bases

Le casting est utilisé pour transformer les données du format long au format large.

En commençant par un long ensemble de données:

```
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)
```

Nous pouvons `dcast` nos données en utilisant la fonction dcast dans data.table. Cela retourne un autre data.table au format large:

```
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"
```

Lancer une valeur

Un argument `value.var` est nécessaire pour une distribution correcte - si elle n'est pas fournie, Dcast émettra une hypothèse basée sur vos données.

```
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
```

Plusieurs `value.var` peuvent être fournies dans une liste

```
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
```

Formule

Le casting est contrôlé en utilisant l'argument de la formule dans `dcast` . C'est de la forme 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
```

Les lignes et les colonnes peuvent être étendues avec d'autres variables en utilisant `+`

```
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
```

Le casting peut souvent créer des cellules où aucune observation n'existe dans les données. Par défaut, cela est désigné par `NA` , comme ci-dessus. Nous pouvons remplacer cela par l'argument `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
```

Vous pouvez également utiliser deux variables spéciales dans l'objet formule

- . ne représente pas d'autres variables
- . . . représente toutes les autres 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
```

Agréger notre value.var

Nous pouvons également couler et agréger les valeurs en une seule étape. Dans ce cas, nous avons trois observations dans chacune des intersections de Age et ID. Pour définir quelle agrégation nous voulons, nous utilisons l'argument `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
```

On peut aussi passer une liste à `fun.aggregate` pour utiliser plusieurs fonctions

```
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
```

Si on passe plus d'une fonction et plus d'une valeur, on peut calculer toutes les combinaisons en passant un vecteur de 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
```

où chaque paire est calculée dans l'ordre `value1_formula1, value1_formula2, ... , valueN_formula(N-1), valueN_formulaN .`

Alternativement, nous pouvons évaluer nos valeurs et fonctions un à un en passant "value.var" comme une liste:

```
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
```

Nommer des colonnes dans le résultat

Par défaut, les composants de nom de colonne sont séparés par un trait de soulignement _ . Cela peut être remplacé manuellement à l'aide de l'argument `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
```

Cela `fun.aggregate` tous les `fun.aggregate` ou `value.var` nous utilisons:

```
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
```

Empilement de plusieurs tables à l'aide de rbindlist

Un refrain commun en R va dans ce sens:

> Vous ne devriez pas avoir un tas de tables liées avec des noms comme `DT1` , `DT2` , ..., `DT11` . Lire et assigner de manière itérative aux objets par leur nom est compliqué. La solution est une liste de tables de données!

Une telle liste ressemble à

```
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
```

Une autre perspective est que vous devez stocker ces tables *en une seule table* , en les empilant. C'est simple à faire en utilisant `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
```

Ce format est beaucoup plus logique avec la syntaxe data.table, où les opérations "par groupe" sont courantes et simples.

Pour un regard plus profond, la réponse de Gregor pourrait être un bon point de départ. Jetez également un coup d' `?rbindlist` Il y a un autre exemple couvrant la lecture dans un tas de tables à partir de CSV, puis leur empilement .

Lire Remodelage, empilement et fractionnement en ligne: https://riptutorial.com/fr/data-table/topic/4117/remodelage--empilement-et-fractionnement

Chapitre 9: Sous-groupes de lignes

Remarques

Un rappel: la syntaxe `DT[where, select|update|do, by]` est utilisée pour travailler avec des colonnes d'un data.table.

- La partie "where" est l'argument `i`
- La partie "select | update | do" est l'argument `j`

Ces deux arguments sont généralement passés par position plutôt que par nom.

Examples

Sélection des lignes dans chaque groupe

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

Supposons que, pour chaque sexe, nous voulons les rangées avec le plus grand nombre de survie:

```
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` est le sous-ensemble de données associé à chaque `Sex` ; et nous sommes en train de les attribuer aux lignes correspondant à notre condition. Si la vitesse est importante, utilisez plutôt une approche suggérée par eddi sur 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
```

Pièges

Dans la dernière ligne de code, `.I` fait référence aux numéros de ligne de la totalité de data.table. Cependant, ce n'est pas vrai quand il n'y en a pas `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
```

Sélection de groupes

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

Supposons que nous voulons seulement voir la deuxième 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
```

Ici, nous sous-ensemble simplement les données en utilisant `i` , la clause "where".

Sélection de groupes par condition

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

Supposons que nous voulons voir chaque classe seulement si une majorité a survécu:

```
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
```

Ici, nous retournons le sous-ensemble de données `.SD` uniquement si notre condition est remplie. Une alternative est

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

mais cela s'est parfois avéré plus lent.

Lire Sous-groupes de lignes en ligne: https://riptutorial.com/fr/data-table/topic/3784/sous-groupes-de-lignes

Chapitre 10: Utilisation de .SD et .SDcols pour le sous-ensemble de données

Introduction

Le symbole spécial `.SD` est disponible dans `j` de `DT[i,j,by]` , la capture de la **S** ubset de **D** ata pour chacun `by` groupe survivant du filtre, `i` . `.SDcols` est une aide. Tapez `?`special-symbols`` pour les documents officiels.

Remarques

Un rappel: la syntaxe `DT[where, select|update|do, by]` est utilisée pour travailler avec des colonnes d'un data.table.

- La partie "where" est l'argument `i`
- La partie "select | update | do" est l'argument `j`

Ces deux arguments sont généralement passés par position plutôt que par nom.

Examples

Utiliser .SD et .SDcols

.DAKOTA DU SUD

`.SD` fait référence au sous-ensemble de `data.table` pour chaque groupe, à l'exclusion de toutes les colonnes utilisées `by` .

`.SD` avec `lapply` peut être utilisé pour appliquer n'importe quelle fonction à plusieurs colonnes par groupe dans un `data.table`

Nous continuerons à utiliser le même jeu de données `mtcars` , `mtcars` :

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

Moyenne de toutes les colonnes du jeu de données par *nombre de cylindres* , `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
```

En dehors de `cyl` , il existe d'autres colonnes catégoriques dans le jeu de données, telles que `vs` , `am` , `gear` et `carb` . Cela n'a pas vraiment de sens de prendre la `mean` de ces colonnes. Excluons donc ces colonnes. C'est ici que `.SDcols` entre en jeu.

.SDcols

`.SDcols` spécifie les colonnes de `data.table` incluses dans `.SD` .

Moyenne de toutes les colonnes (colonnes continues) dans l'ensemble de données selon le *nombre d'engrenages* `gear` , et le *nombre de cylindres,* `cyl` , disposés par `gear` et `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
```

Peut-être que nous ne voulons pas calculer la `mean` par groupes. Pour calculer la moyenne de toutes les voitures du jeu de données, nous ne spécifions pas la variable `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
```

Remarque: Il n'est pas nécessaire de définir `cols_chosen` au préalable. `.SDcols` peut directement prendre des noms de colonnes

Lire Utilisation de .SD et .SDcols pour le sous-ensemble de données en ligne: https://riptutorial.com/fr/data-table/topic/3787/utilisation-de--sd-et--sdcols-pour-le-sous-ensemble-de-donnees

Chapitre 11: Utilisation de colonnes de liste pour stocker des données

Introduction

Data.table prend en charge les vecteurs de colonne appartenant à la classe de `list` de R.

Remarques

Dans le cas où il semble étrange que nous parlions de listes sans utiliser ce mot dans le code, notez que `.()` Est un alias pour `list()` lorsqu'il est utilisé dans un appel `DT[...]` .

Examples

Lecture dans de nombreux fichiers liés

Supposons que nous voulions lire et empiler un tas de fichiers de format similaire. La solution rapide est la suivante:

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

Nous pourrions ne pas être satisfaits de cela pour deux raisons:

- Il peut se `rbindlist` à des erreurs lors de la lecture avec `fread` ou lors de l'empilement avec `rbindlist` raison d'un formatage de données incohérent ou `rbindlist` .
- Nous pouvons vouloir garder une trace des métadonnées pour chaque fichier, extraites du nom du fichier ou peut-être de certaines lignes d'en-tête dans les fichiers (pas tout à fait tabulaires).

Une façon de gérer cela consiste à créer une "table de fichiers" et à stocker le contenu de chaque fichier en tant qu'entrée de colonne de liste sur la ligne qui lui est associée.

Exemple de données

Avant de créer l'exemple de données ci-dessous, assurez-vous que vous êtes dans un dossier vide dans lequel vous pouvez écrire. Exécutez `getwd()` et lisez `?setwd` si vous devez changer de dossier.

```
# 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))
```

Identifier les fichiers et les métadonnées de fichier

Cette partie est assez simple:

```
# 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
```

Lire dans les fichiers

Lire dans les fichiers en tant que colonne de liste:

```
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>
```

S'il y a un accroc dans la lecture de l'un des fichiers ou si vous devez changer les arguments en `fread` fonction des attributs du fichier, cette étape peut facilement être étendue, ressemblant à:

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

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

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

Pour plus de détails sur les options de lecture dans les fichiers CSV et les fichiers similaires, voir `?fread` .

Empiler les données

De là, nous voulons empiler les données:

```
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
```

Si un problème survient lors de l'empilement (comme les noms de colonne ou les classes non correspondantes), nous pouvons revenir aux tables individuelles dans `fileDT` pour inspecter l'origine du problème. Par exemple,

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

Les extensions

Si les fichiers ne sont pas dans votre répertoire de travail actuel, utilisez

```
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))))]
```

Lire Utilisation de colonnes de liste pour stocker des données en ligne: https://riptutorial.com/fr/data-table/topic/4456/utilisation-de-colonnes-de-liste-pour-stocker-des-donnees

Chapitre 12: Utiliser des clés et des index

Introduction

La clé et les index d'un data.table permettent à certains calculs de s'exécuter plus rapidement, principalement liés aux jointures et aux sous-ensembles. La clé décrit l'ordre de tri actuel de la table; tandis que chaque index stocke des informations sur l'ordre de la table en respectant une séquence de colonnes. Voir la section «Remarques» ci-dessous pour des liens vers les vignettes officielles sur le sujet.

Remarques

Les vignettes officielles sont la meilleure introduction à ce sujet:

- "Sous-ensemble de clés et de recherche binaire rapide"
- "Indices secondaires et indexation automatique"

Clés vs indices

Une data.table peut être "saisie" par une séquence de colonnes, indiquant aux fonctions intéressées que les données sont triées par ces colonnes. Pour obtenir ou définir la clé, utilisez les fonctions documentées à la `?key`

De même, les fonctions peuvent tirer parti des "indices" de data.table. Chaque index - et une table peut en avoir plusieurs - stocke des informations sur l'ordre des données en respectant une séquence de colonnes. Comme une clé, un index peut accélérer certaines tâches. Pour obtenir ou définir des indices, utilisez les fonctions documentées dans `?indices` .

Les indices peuvent également être définis automatiquement (actuellement pour une seule colonne à la fois). Voir `?datatable.optimize` pour plus de détails sur son fonctionnement et sa désactivation si nécessaire.

Vérification et mise à jour

Les valeurs manquantes sont autorisées dans une colonne de clé.

Les clés et les index sont stockés en tant qu'attributs et peuvent, par accident, ne pas correspondre à l'ordre réel des données dans la table. De nombreuses fonctions vérifient la validité de la clé ou de l'index avant de l'utiliser, mais il convient de garder à l'esprit.

Les clés et les index sont supprimés après les mises à jour, où il n'est pas évident que l'ordre de tri est préservé. Par exemple, à partir de `DT = data.table(a=c(1,2,4), key="a")` , si nous mettons à jour comme `DT[2, a := 3]` , la clé est brisée.

Examples

Amélioration des performances pour la sélection des sous-ensembles

```
# 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)
)
```

Correspondance sur une colonne

Après la première exécution d'une opération de sous-ensemble avec `==` ou `%in%` ...

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

Un index a été créé automatiquement pour `g1` . Les opérations subséquentes de sous-ensembles s'exécutent presque instantanément:

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

Pour surveiller la création ou l'utilisation d'un index, ajoutez l'option `verbose=TRUE` ou modifiez les `options(datatable.verbose=TRUE)` paramètres globaux `options(datatable.verbose=TRUE)` .

Correspondance sur plusieurs colonnes

Actuellement, la correspondance sur deux colonnes ne crée pas automatiquement un index:

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

Relancez ceci et il restera lent. Même si nous ajoutons manuellement l'index avec `setindex(DT, g1, g2)` , il restera lent car cette requête n'est pas encore optimisée par le package.

Heureusement, si nous pouvons énumérer les combinaisons de valeurs que nous souhaitons rechercher et qu'un index est disponible, nous pouvons rapidement les joindre:

```
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
```

Avec `CJ` , il est important de faire attention au nombre de combinaisons devenant trop important.

Lire Utiliser des clés et des index en ligne: https://riptutorial.com/fr/data-table/topic/4977/utiliser-des-cles-et-des-index

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec data.table	Community, Frank, micstr
2	Ajout et modification de colonnes	eddi, Frank, jangorecki, micstr
3	Calculer des statistiques sommaires	Frank
4	Créer un data.table	Chris, Frank
5	Données de nettoyage	Frank
6	Joint et fusionne	Chris, Frank
7	Pourquoi mon ancien code ne fonctionne pas?	Frank, micstr
8	Remodelage, empilement et fractionnement	Chris, David Arenburg, Frank, SymbolixAU
9	Sous-groupes de lignes	Frank, micstr
10	Utilisation de .SD et .SDcols pour le sous-ensemble de données	Frank
11	Utilisation de colonnes de liste pour stocker des données	Frank
12	Utiliser des clés et des index	Frank