Package data.table

Contents

1. Working on Large Data
2. Introduction to package data.table
3. Importing Data
4. Creating Data
5. Merging Data
6. Creating Subsets
7. Sorting Data
8. Modifying Data
9. Aggregating Data

Working on Large Data

Many times R users struggle helplessly while dealing with large data sets. When your machine fail to work on large data sets, it gives repetitive warnings, error messages of insufficient memory usage.

Below are some practices which impede R's performance on large data sets:

- Using `read.csv()` to load large files.
- Using `aggregate()` to perform aggregation on large datasets.
- Using web browser : Opening several tabs in chrome consumes a significant amount of system's memory.
- Machine Specification : R reads objects into RAM. Your system should have enough free RAM space available which could seamlessly work with large data.

There are more such factors and functions which hinder R's performance.

* In case of large data sets, a data.frame can be limiting, the time it takes to do certain things is too long. The package **`data.table`** solves this by reducing computing time.

Introduction to Package data.table

The R package data.table is written by Matt Dowle in year 2008.

It provides an enhanced version of **`data.frame`** that allows you to do :

- Fast aggregation of large data (e.g. 100GB in RAM)
- Fast ordered joins
- Fast add/modify/delete of columns by group using no copies at all
- List columns
- Fast file import (`fread`).

How to use package **`data.table?`**

- Import and load the package
- You can create your own data using **`data.table()`** exactly the same way you create dataframe or convert a dataframe into a datatable using **`data.table()`**

Data Snapshot

sal_data consists information about Employee's Basic Salary, their ID & full Name

Employee_ID	First_Name	Last_Name	Basic_Salary
E-1001	Mahesh	Joshi	16860
E-1002	Ra		
E-1004	Pr		
E-1005	Sn		
E-1007	R		
E-1008	N		
E-1009	Har		

Columns	Description	Type	Measurement	Possible values
Employee_ID	Employee ID	character	-	-
First_Name	First Name	character	-	-
Last_Name	Last Name	character	-	-
Basic_Salary	Basic Salary	numeric	Rs.	positive values

bonus_data has information of only Bonus given to Employees.

"Employee ID" is the common column in both datasets

Employee_ID	Bonus
E-1001	16070
E-1003	
E-1004	
E-1006	
E-1008	
E-1010	

Columns	Description	Type	Measurement	Possible values
Employee_ID	Employee ID	character	-	-
Bonus	Bonus amount	Numeric	Rs.	Positive values

Importing Data

The **fread()** of package **data.table** loads large datasets faster and more conveniently than other base functions in R. **fread()** imports data in data.table format.

```
# Install and load package data.table
# Compare read.csv() with fread()
```

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

```
system.time(dt<-read.csv("data.csv"))
```

```
system.time(dt<-fread("data.csv"))
```

Here **data.csv** is a dummy file. Run these speed test codes on large data, you will see loading data with **fread()** is much faster than the base function **read.csv().**
system.time() is used to measure the execution time for an R function.

* **fread()** is faster than read.csv() because, **read.csv()** first read rows as character and then converts them into integer and factor as data types. On the other hand, **fread()** simply reads everything as character.

Merging Data

```
# Import data :
```

```
sal_data <- fread("sal_data.csv")
bonus_data <- fread("bonus_data.csv")
```

```
# Set key as ID for both data tables
```

```
setkey(sal_data,Employee_ID)
setkey(bonus_data,Employee_ID)
```

setkey() sorts the rows of the data table by the column(s) provided in ascending order. Keys in data table delivers incredibly fast results. You can set keys on any type of column i.e. numeric, factor, integer, character.

Merging Data

```
# Merge data1 and data2

merged <- merge(sal_data,bonus_data,all = TRUE)
```

- ❑ **merge()** function in data.table is same as base R function, the only difference is that we don't have to mention "**by**" which variable we want to do the merging as we have already mentioned that in **setkey()** function.
- ❑ Also, here we have performed outer join by specifying **all = TRUE**
- ❑ We can perform others as follows:

all.x = TRUE for left join, **all.y = TRUE** for right join & **merge(data1,data2)** for inner join

Here, **merge()** is the function of package data.table used for quick merging of data tables based on common keys (by default).

This function acts very similarly to that of data.frame's, with the major exception being that the default columns used to merge two datatables are the shared key columns, and not the shared columns with the same names.

Creating Data

Create a large datatable to perform data manipulation tasks

```
# dt1 with 70,00,000 rows and 4 columns using random sampling
```

```
dt1<- data.table(ID=1:7000000,
                 Capacity=sample(100:1000,size=50,replace=F),
                 Code=sample(LETTERS[1:4],50,replace=T),
                 State=rep(c("Alabama","Indiana","Texas","Nevada")))
head(dt1, 4)
```

```
   ID Capacity Code   State
1:  1      918    C Alabama
2:  2      836    C Indiana
3:  3      488    B   Texas
4:  4      647    A  Nevada
```

- **data.table()** is used to create a datatable and convert a dataframe into a datatable
- **sample**(x, size, replace = FALSE)
- **x =** either a vector of one or more elements from which to choose, or a positive integer
- **size =** a non-negative integer giving the number of items to choose.
- **replace =** should sampling be with replacement?

- `Capacity=sample(100:1000,size=50,replace=F) :` For column "Capacity" a sample of size 50 from numbers 100 to 1000 is generated & sampling is done without replacement.
- `Code=sample(LETTERS[1:4],50,replace=T):` Similarly, for column "Code" a sample of size 50 from 1st 4 letters(A:D) is generated & sampling is done with replacement.
- And the process is going to be repeated till 7000000 observations are generated.
- `State=rep(c("Alabama","Indiana","Texas","Nevada")) :`Column "State" uses rep() function where the values are repeated in same order as mentioned till all 7000000 observations are generated.

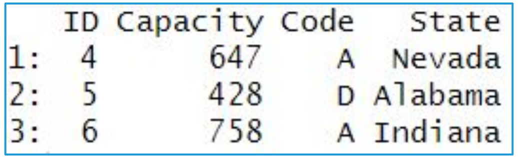
* Note : Here the Output will vary for each user as we are generating random sample.

Creating Subsets

```
# Subsetting rows by numbers
```

```
dt1[4:6,]
```

```
#Output
```

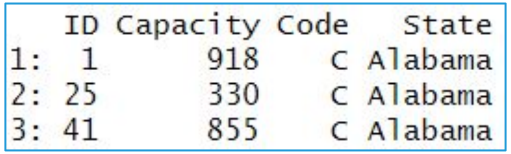


```
   ID Capacity Code   State
1:  4      647    A  Nevada
2:  5      428    D Alabama
3:  6      758    A Indiana
```

```
# Use column names to select rows
```

```
sub_rows<-dt1[Code=="C" & State=="Alabama"]
head(sub_rows,3)
```

```
#Output
```



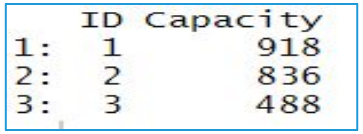
```
   ID Capacity Code   State
1:  1      918    C Alabama
2: 25      330    C Alabama
3: 41      855    C Alabama
```

Creating Subsets

```
# Subsetting columns
```

```
sub_columns<-dt1[,.(ID,Capacity)]
head(sub_columns, 3)
```

```
#Output
```

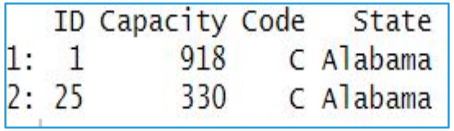


```
    ID Capacity
1:   1      918
2:   2      836
3:   3      488
```

```
# Subset all rows using key columns where first key column “Code” matches
“C” and second key column “State” matches “Alabama”
```

```
setkey(dt1,Code,State)
sub_key<-dt1[.("C","Alabama")]
head(sub_key, 2)
```

```
#Output
```



```
   ID Capacity Code   State
1:  1      918    C Alabama
2: 25      330    C Alabama
```

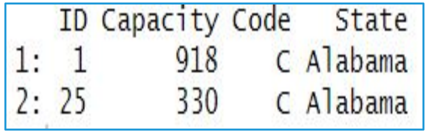
Once the key is set, we no longer need to provide the column name again and again.

Creating Subsets

```
# Subset all rows where just the first key column “Code” matches “C”
```

```
sub_key2<-dt1[.("C")]
head(sub_key2, 2)
```

#Output

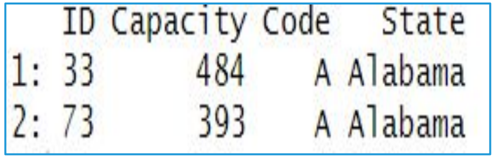


```
   ID Capacity Code   State
1:  1      918    C Alabama
2: 25      330    C Alabama
```

```
# Subset all rows where just the second key column “State” matches “Alabama”
```

```
sub_key3 <- dt1[.(unique(Code),"Alabama")]
head(sub_key3, 2)
```

#Output



```
   ID Capacity Code   State
1: 33      484    A Alabama
2: 73      393    A Alabama
```

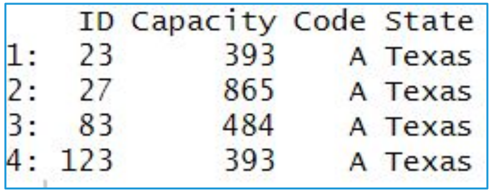
We can not skip the values of key columns before. Therefore we provide all unique values from key column Code.

Sorting Data

```
# Sort dt1 by variable ‘Code’ in ascending order and variable ‘State’
# in descending order
```

```
dt_order1<-dt1[order(Code,-State)]
head(dt_order1,4)
```

```
#Output
```



```
      ID Capacity Code State
1:    23      393    A Texas
2:    27      865    A Texas
3:    83      484    A Texas
4:   123      393    A Texas
```

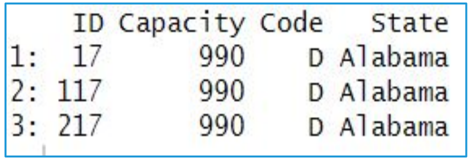
By default **order()** sorts variables in ascending order.
‘-’ sign results in descending order.

NOTE : order() in package data.table is much faster than base function **order()** because it uses **radix order sort** which imparts additional boost.

```
# Sort dt1 by variables ‘Code’ and ‘Capacity’ in descending order
```

```
dt_order2<-dt1[order(-Code,-Capacity)]
head(dt_order2,3)
```

```
#Output
```



```
      ID Capacity Code   State
1:    17      990    D Alabama
2:   117      990    D Alabama
3:   217      990    D Alabama
```

Modifying Data

```
# Add a new column 'new_capacity' to dt1
```

```
dt1[,new_capacity:=Capacity+5]
head(dt1,4)
```

```
#Output
```

```
    ID Capacity Code   State new_capacity
1:  33      484    A Alabama          489
2:  73      393    A Alabama          398
3:  77      865    A Alabama          870
4: 133      484    A Alabama          489
```

Using ':=' operator, new columns can be added and assigned values.

Remember while sorting we had set the key dt1 on 'Code' and 'State' so any operation on dt1 will return the output sorted on keys.

```
# Update row values 'Alabama' to 'Al' in column 'State'
```

```
dt1[State=="Alabama",State:="Al"]
head(dt1,3)
```

```
#Output
```

```
   ID Capacity Code State new_capacity
1: 33      484    A    Al          489
2: 73      393    A    Al          398
3: 77      865    A    Al          870
```

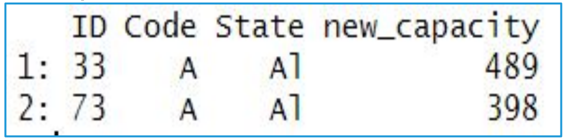
Columns are updated using ':=' operator

Modifying Data

```
# Delete column ‘Capacity’ from dt1
```

```
dt1[,c("Capacity"):=NULL]
head(dt1,2)
```

```
#Output
```



```
   ID Code State new_capacity
1: 33    A    Al          489
2: 73    A    Al          398
```

Column can be removed, by assigning **NULL** to it, using **‘:=‘** operator.

```
# Concept of Chaining of commands
# Execute last three commands together in one command
```

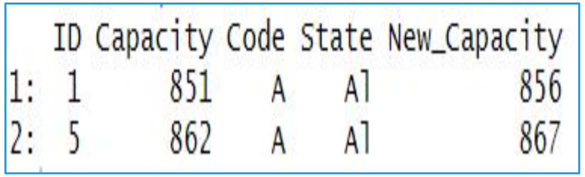
```
dt1[,new_capacity:=Capacity + 5] [State=="Alabama",State:="Al"]
[,"Capacity":=NULL]
```

Modifying Data

```
# Renaming column ‘new_capacity’ from dt1
```

```
setnames(dt1,old=″new_capacity” , new = ”New_Capacity”)
head(dt1,2)
```

```
#Output
```



```
   ID Capacity Code State New_Capacity
1:  1      851    A    Al          856
2:  5      862    A    Al          867
```

* Note : Multiple columns can be renamed just in one code

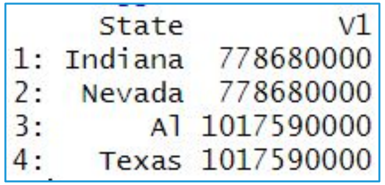
setnames(data, old=c(“old_name”,…),new=c(“new_names”,…))

Aggregating Data

```
# To calculate sum of variable ‘New_Capacity’ by variable ‘State’
```

```
setkey(dt1, New_Capacity,State)
DT_agg <- dt1[,sum(New_Capacity),by=State]
DT_agg
```

```
#Output
```



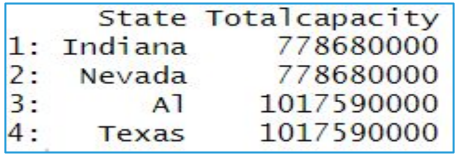
```
     State         V1
1: Indiana  778680000
2:  Nevada  778680000
3:      Al 1017590000
4:   Texas 1017590000
```

- ❑ **New_Capacity** and **State** are set as keys for faster aggregation.
- ❑ **by=** takes the factors by which data will be aggregated. Multiple factors can be specified with **list()**
- ❑ A new column **V1** is created with sum of **New_Capacity, State** wise.

```
# To change the name of the column while aggregating, see the below command.
# Rename the column ‘V1’ to ‘Totalcapacity’
```

```
DT_agg <- dt1[,.(Totalcapacity=sum(New_Capacity)),by=State]
DT_agg
```

```
#Output
```



```
     State Totalcapacity
1: Indiana     778680000
2:  Nevada     778680000
3:      Al    1017590000
4:   Texas    1017590000
```

.() notation allows you to rename the columns inside datatable.

Quick Recap

In this session, we learnt how useful package **`data.table`** is for working on large data.

Here is the quick recap:

Importing, creating data and merging data	• **`fread()`** is used to import data. It is faster than read.csv() • **`data.table()`** is used to create data • **`merge()`** is used to merge data sets on common columns
Sub setting	• Subsetting can be done by rows and columns. • **`setkey()`** sets the variables as keys and speeds up the subsetting
Sorting	• **`order()`** is used to sort data.table in ascending/ descending order
Modifying	• Adding new column, updating values, deleting columns, chaining command & renaming columns for all these functions are possible with **`data.table`**
Aggregating	• **`setkey()`** is used to set keys on variables as it provides faster aggregation. • Aggregation through package **`data.table`** is much faster than through R's built-in function **`aggregate()`**.