Introduction to the **data.table** Package in R

Matthew Dowle

April 12, 2010

Introduction

This vignette is aimed at those who are already familiar with R, in particular creating and using objects of class data.frame. We aim for this quick introduction to be readable in **10 minutes**, covering the main features in brief. The main features are the 3 numbered section titles: 1.Keys, 2.Fast Grouping, 3.Fast Merging/Joining. For the context that this document sits please briefly check the last section, Further Resources.

data.table is not *automatically* better or faster. The user has to climb a short learning curve, experiment, and then use the features well. For example this document explains the difference between a *vector scan* and a *binary search*. Both extract methods are available. If a user continues to use vector scans though, as they are used to in a data.frame, it will work, but they will miss out on the benefits that the package provides.

Creation

Recall that we create a `data.frame` using the function `data.frame()`.

```
> df = data.frame(x=c("b","b","b","a","a"),v=rnorm(5))
> df

  x          v
1 b  0.5897057
2 b  0.4194377
3 b  0.5577020
4 a  1.8397292
5 a -0.1164002
```

We create a `data.table` in exactly the same way.

```
> dt = data.table(x=c("b","b","b","a","a"),v=rnorm(5))
> dt

     x          v
[1,] b -1.3473165
[2,] b  0.4249705
[3,] b  0.3268059
[4,] a -0.7267139
[5,] a  0.1424192
```

Observe that a data.table prints the row numbers slightly differently. There is nothing significant about that. We can also convert existing data.frame objects to data.table.

```
> cars = data.table(cars)
> head(cars)
```

```
     speed dist
[1,]     4    2
[2,]     4   10
[3,]     7    4
[4,]     7   22
[5,]     8   16
[6,]     9   10
```

We have just created two data.tables: dt and cars. It is often useful to see a list of all our data.tables in memory.

```
> tables()
```

```
     NAME NROW MB COLS       KEY
[1,] cars   50 1  speed,dist
[2,] dt      5 1  x,v
Total: 2MB
```

The MB column is useful to quickly assess memory use and to spot if any redundant tables can be removed to free up memory. Just like data.frame's, data.table's must fit inside RAM.

Some users regularly work with 20 or more tables in memory, rather like a database. The result of tables() is itself a data.table, returned silently, so that tables() can be used in programs. tables() is unrelated to the base function table().

Also note that data.table() automatically converted character vectors to factor.

```
> sapply(dt,class)
```

```
        x         v
 "factor" "numeric"
```

This is for efficiency. As the user you should vary rarely need know that this has occurred. See `?factor` if you are unfamiliar with factors. Factors will appear to you as though they are character columns.

You may have noticed the empty column KEY from `tables()` above. This is the subject of the next section, the first of the 3 main features of the package.

1. Keys

Lets start by considering data.frame, specifically ”rownames”. Or in English ”row names”. That is, the multiple names belonging to the single row. The multiple names belonging to the single row? No, that is not what we are used to in a data.frame. We know that each row has at most one name, but never *more* than one name. A person has at least two names, a first name and a second name. That is useful to organise a phone directory of people. But each row in a data.frame can only have one name.

A *key* is one or more columns of rownames. These columns may be integer, factor or other classes, not just character. Furthermore, the rows are sorted by the key. Therefore a data.table can have at most one key, because it cannot be sorted in more than one way.

Uniqueness is not enforced i.e. duplicate key values are allowed. Since the rows are sorted by the key, any duplicates in the key will appear consecutively.

Lets remind ourselves of our tables :

```
> tables()
```

```
     NAME NROW MB COLS       KEY
[1,] cars   50 1  speed,dist
[2,] dt      5 1  x,v
Total: 2MB
```

```
> dt
```

```
     x          v
[1,] b -1.3473165
[2,] b  0.4249705
[3,] b  0.3268059
[4,] a -0.7267139
[5,] a  0.1424192
```

No keys have been set yet. We *can* use data.frame syntax without a key.

```
> dt[2,]
```

```
     x         v
[1,] b 0.4249705
```

```
> dt[ dt$x == "b", ]
```

```
     x          v
[1,] b -1.3473165
[2,] b  0.4249705
[3,] b  0.3268059
```

But since there are no rownames the following does not work.

```
> cat(try(dt["b",]))
```

```
Error in `[.data.table`(dt, "b", ) :
  The data.table has no key but i is character. Call setkey first, see ?setkey.
```

The error message tells us we need to use setkey().

```
> setkey(dt,x)
> dt
```

```
     x          v
[1,] a -0.7267139
[2,] a  0.1424192
[3,] b -1.3473165
[4,] b  0.4249705
[5,] b  0.3268059
```

Notice that the rows in dt have been re-ordered by x. The two ”a” rows have moved to the top. We can confirm that dt does indeed have a key using `haskey()`, `key()`, `attributes()`, or just running `tables()`.

```
> tables()
```

```
     NAME NROW MB COLS       KEY
[1,] cars   50 1  speed,dist
[2,] dt      5 1  x,v        x
Total: 2MB
```

Now we are sure that dt has a key, lets try again.

```
> dt["b",]
```

```
     x         v
[1,] b -1.347316
```

Since there are duplicates in this key (i.e. repeated values of ”b”) the subset returns the first row in that group, by default. The `mult` argument (short for *multiple*) controls this.

```
> dt["b",mult="first"]

     x        v
[1,] b -1.347316

> dt["b",mult="last"]

     x        v
[1,] b 0.3268059

> dt["b",mult="all"]

     x          v
[1,] b -1.3473165
[2,] b  0.4249705
[3,] b  0.3268059
```

Lets now create a data.table with a 2 column key. We can do this in one step this time by using the key argument of data.table(). We will also make it large enough to demonstrate the difference between a *vector scan* and a *binary search.*

```
> n = ceiling(1e7/26^2)   # 10 million rows
> DT = data.table(x=rep(LETTERS,each=26*n),
+                y=rep(letters,each=n),
+                v=rnorm(n*26^2),
+                key="x,y")
> head(DT)

     x y           v
[1,] A a -0.26390884
[2,] A a  0.07575623
[3,] A a -0.45070003
[4,] A a -1.09239659
[5,] A a -0.65857715
[6,] A a -1.26567688

> tables()

     NAME       NROW  MB COLS       KEY
[1,] cars         50   1 speed,dist
[2,] dt            5   1 x,v        x
[3,] DT   10,000,068 153 x,y,v      x,y
Total: 155MB
```

Notice that the data.table automatically sorted the rows by the key, so group (x==”A”,y==”a”) appears at the top, group (x==”Z”,y==”z”) at the bottom.

Lets use the same syntax you might use with a data.frame to extract an arbitrary group :

```
> tt=system.time(ans1 <- DT[DT$x=="R" & DT$y=="h",]); tt

   user  system elapsed
  4.229   1.092   5.393

> head(ans1)
```

```
     x y          v
[1,] R h  0.7304412
[2,] R h -0.8299844
[3,] R h  2.1697774
[4,] R h  1.1621616
[5,] R h  0.8140029
[6,] R h -0.7145970
```

```
> dim(ans1)
```

```
[1] 14793     3
```

When we used `DT$x=="R"` we *scanned* the entire column x, testing each and every value to see it equalled "R". We did that again in the y column, testing for "h". Then `&` combined the two logical results to create a single logical vector which was passed to the `[` method which searched it for `TRUE` and returned those rows. These were *vectorized* operations. They occurred internally in R and were very fast, but they were scans. *We* did those scans because *we* wrote that R code.

How do we do this in data.table?

```
> ss=system.time(ans2 <- DT[data.table("R","h"),mult="all"]); ss
```

```
   user  system elapsed 
  0.016   0.000   0.021 
```

```
> identical(ans1,ans2)
```

```
[1] TRUE
```

At 0.021sec, this was **256 times faster** than 5.393sec, and produced precisely the same result. If the phone book analogy helped, then this should not be surprising. We use the key. We take advantage of the fact that the table is sorted and we use binary search to find the matching rows. We didn't vector scan; we didn't use `==`.

When i is itself a data.table, we say that we are *joining* the two data.table's. In this case we are joining DT to the 1 row, 2 column table returned by `data.table("R","h")`. Since we do this a lot, there is an alias for `data.table` called `J()`, short for join.

```
> identical( DT[J("R","h"),mult="all"],
+            DT[data.table("R","h"),mult="all"] )
```

```
[1] TRUE
```

Both vector scanning, and binary search, are available in `data.table`, but one way of using data.table is much better than the other.

The join syntax is short, fast to write and easy to maintain. Passing a data.table into a data.table subset, is similar to base `R` which allows a matrix to be passed into a matrix subset. [1]. There are other types of join and further arguments which are beyond the scope of this quick introduction.

This first section has been about the first argument to the `[`, namely `i`. The next section is do with the 2nd argument.

2. Fast grouping

The second argument to `[` is `j` and may be one or more expressions of column names, as if the column names were variables.

```
> dt[,sum(v)]
```

[1] Subsetting a key'd data.table by an n-column data.table is consistent with subsetting a n-dimension array by an n-column matrix

```
[1] -1.179835
```

When we supply a j expression and a 'by' list of expressions, the j expression is repeated for each group defined by the 'by'.

```
> dt[,sum(v),by=x]

     x         V1
[1,] a -0.5842947
[2,] b -0.5955400
```

The 'by' in data.table is fast. Lets compare to tapply.

```
> ttt=system.time(tt <- tapply(DT$v,DT$x,sum)); ttt

   user  system elapsed
  9.820   1.060  11.003

> sss=system.time(ss <- DT[,sum(v),by=x]); sss

   user  system elapsed
  0.524   0.300   0.825

> head(tt)

         A          B          C          D          E          F
 236.4593   869.1774 -249.3990   362.4604   123.7583 -204.4259

> head(ss)

     x        V1
[1,] A  236.4593
[2,] B  869.1774
[3,] C -249.3990
[4,] D  362.4604
[5,] E  123.7583
[6,] F -204.4259

> identical(as.vector(tt), ss$V1)

[1] TRUE
```

At 0.825sec, this was **13 times faster** than 11.003sec, and produced precisely the same result. Lets group by two columns.

```
> ttt=system.time(tt <- tapply(DT$v,list(DT$x,DT$y),sum)); ttt

   user  system elapsed
 10.969   1.376  12.532

> sss=system.time(ss <- DT[,sum(v),by="x,y"]); sss

   user  system elapsed
  0.544   0.396   0.939

> tt[1:5,1:5]

           a         b         c           d         e
A -148.285344 136.95789 143.30928  -93.546001  32.57468
B -158.225514  69.58285 -94.46516   63.870396 136.43671
C    9.617828 -22.92906 101.12515 -147.057847  66.57162
D   31.502234 114.10637 107.87029 -190.219503 109.32226
E -114.560547 103.87172  42.73399    1.904887  26.80020
```

```
> head(ss)

     x y         V1
[1,] A a -148.28534
[2,] A b  136.95789
[3,] A c  143.30928
[4,] A d  -93.54600
[5,] A e   32.57468
[6,] A f   19.64259

> identical(as.vector(t(tt)), ss$V1)

[1] TRUE
```

This was **13 times faster**, and the syntax a little simpler and easier to read.

To return several expressions, pass a list() to j. TO DO - Example

The list is auto-recycled within each group. Do a condition on that. To DO - Example.

You can pass expressions to by too. And for this its cleaner to pass a list() to by, instead of character. TO DO - Example.

There is a super-charge option for advanced users. Use the dropfactor. To DO - Example.

3. Fast joining/merging of irregularly spaced ordered observation

Recap : x[i] is a join between x and data.table i x[,j,by=list()] is grouping

This section now combines together i, j and by together in one query. We also introduce roll.

merge(x,y) is the same but contains the union of columns. When merge is passed data.table's, it operates much faster than base merge on data.frame's. TO DO - example.

Grouping without setting a key : 1. Fast temporary key creation due to fast radix. 2. Using key without 'by'.

Fast roll join. TO DO - example.

Other resources

This was a quick start guide. Further resources include :

- The help page describes each and every argument `?"[.data.table"`
- The FAQs deal with distinct topics in an easy to digest manner
- The performance tests are more complex real world examples
- test.data.table contains over 100 low level tests of the features.
- Presentations
- YouTube Demo
- R-Forge commit logs: `http://lists.r-forge.r-project.org/pipermail/datatable-commits/`
- Website: `http://datatable.r-forge.r-project.org/`
- Mailing list : datatable-help@lists.r-forge.r-project.org