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

Matthew Dowle

January 8, 2011
(A later revision may be available on the homepage)

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, namely: 1. Keys; 2. Fast Grouping; and 3. Fast time series join. 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 its features well. For example, this document explains the difference between a *vector scan* and a *binary search*. Although both extraction methods are available in data.table, if a user continues to use vector scans (as in a data.frame), it will 'work', but one will miss out on the benefits that **data.table** 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  0.3107706
2 b  0.2238871
3 b -0.9503995
4 a -1.8747981
5 a -0.5698987
```

A `data.table` is created in exactly the same way:

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

```
     x           v
[1,] b  0.05502934
[2,] b -1.24800624
[3,] b -0.70285690
[4,] a  0.60514845
[5,] a  1.47183765
```

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.table`s: dt and cars. It is often useful to see a list of all 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 converts character vectors to factors.

```
> sapply(dt,class)

        x         v
 "factor" "numeric"
```

Users should rarely need to know this has occurred. See `?factor` if you are unfamiliar with factors. Factors will appear to you as though they are character columns. You can refer to them just as though they are character.

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

Let's start by considering `data.frame`, specifically `rownames` (or in English, *row names*). That is, the multiple names belonging to a single row. The multiple names belonging to the single row? That is not what we are used to in a `data.frame`. We know that each row has at most one name. A person has at least two names, a first name and a second name. That is useful to organise a telephone directory, for example, which is sorted by surname, then first name. However, each row in a data.frame can only have one name.

A *key* consists of one or more columns of rownames, which may be integer, factor or some other class, not simply 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.

Let's 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  0.05502934
[2,] b -1.24800624
[3,] b -0.70285690
[4,] a  0.60514845
[5,] a  1.47183765
```

No keys have been set yet. We can use `data.frame` syntax in a `data.table`, too.

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

```
     x         v
[1,] b -1.248006
```

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

```
     x           v
[1,] b  0.05502934
[2,] b -1.24800624
[3,] b -0.70285690
```

But since there are no rownames, the following does not work:

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

```
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)   # or key(dt)="x" if you prefer
> dt
```

```
     x           v
[1,] a  0.60514845
[2,] a  1.47183765
[3,] b  0.05502934
[4,] b -1.24800624
[5,] b -0.70285690
```

Notice that the rows in dt have been re-ordered according to the values of `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 that we are sure `dt` has a key, let's try again:

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

```
     x          v
[1,] b 0.05502934
```

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 0.05502934

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

     x          v
[1,] b -0.7028569

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

     x          v
[1,] b  0.05502934
[2,] b -1.24800624
[3,] b -0.70285690
```

Lets now create a new data.frame. We will make it large enough to demonstrate the difference between a *vector scan* and a *binary search*.

```
> grpsize = ceiling(1e7/26^2)   # 10 million rows, 676 groups

[1] 14793

> tt=system.time( DF <- data.frame(
+   x=rep(factor(LETTERS),each=26*grpsize),
+   y=rep(factor(letters),each=grpsize),
+   v=runif(grpsize*26^2))
+ )

   user  system elapsed
  6.388   1.844   8.247

> head(DF,3)

  x y          v
1 A a 0.39558467
2 A a 0.32557010
3 A a 0.01643204

> tail(DF,3)

         x y          v
10000066 Z z 0.53972058
10000067 Z z 0.01520018
10000068 Z z 0.79667458

> dim(DF)

[1] 10000068        3
```

We might say that R has created a 3 column table and *inserted* 10,000,068 rows. It took 8.247 secs, so it inserted 1,212,570 rows per second. This is normal in base R.

Let's extract an arbitrary group from the data.frame DF:

```
> tt=system.time(ans1 <- DF[DF$x=="R" & DF$y=="h",])   # 'vector scan'
```

```
   user  system elapsed
 12.433   0.956  13.407
```

```
> head(ans1,3)
```

```
        x y         v
6642058 R h 0.1122041
6642059 R h 0.4705904
6642060 R h 0.2490949
```

```
> dim(ans1)
```

```
[1] 14793     3
```

Now convert to a data.table and extract the same group:

```
> DT = data.table(DF)
> setkey(DT,x,y)
```

```
> ss=system.time(ans2 <- DT[J("R","h"),mult="all"])   # 'binary search'
```

```
   user  system elapsed
  0.020   0.004   0.023
```

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

```
   x    y    v
TRUE TRUE TRUE
```

At 0.023 seconds, this was **582** times faster than 13.407 seconds, and produced precisely the same result. If you are thinking that a few seconds is not much to save, it's the relative speedup that's important. The vector scan is linear, but the binary search is O(log n). It scales. If a task taking 10 hours is sped up by 100 times to 6 minutes, that is significant[1].

We can do vector scans in data.table, too. In other words we can use data.table *badly*.

```
> system.time(ans1 <- DT[x=="R" & y=="h",])   # works but is using data.table badly
```

```
   user  system elapsed
 12.489   0.980  13.490
```

```
> system.time(ans2 <- DF[DF$x=="R" & DF$y=="h",])   # the data.frame way
```

```
   user  system elapsed
 12.473   0.972  13.463
```

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

```
   x    y    v
TRUE TRUE TRUE
```

If the phone book analogy helped, the **582** times speedup should not be surprising. We use the key to take advantage of the fact that the table is sorted and use binary search to find the matching rows. We didn't vector scan; we didn't use `==`.

When we used `DT$x=="R"` we *scanned* the entire column x, testing each and every value to see if it equalled "R". We did it 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 in turn 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.

When i is itself a data.table, we say that we are *joining* the two data.tables. 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`s called `J()`, short for join:

[1] We wonder how many people are deploying parallel techniques to code that is vector scanning

```
> 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 allowing a matrix to be passed into a matrix subset in base R.[2]. There are other types of join and further arguments which are beyond the scope of this quick introduction.

The merge method of `data.table` is essentially `x[y]`, but where the columns of `x` are included in the result. See FAQ 1.10.

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

2. Fast grouping

The second argument to `[` is `j`, which may consist of one or more expressions whose arguments are (unquoted) column names, as if the column names were variables.

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

```
[1] 0.1811523
```

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

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

```
     x        V1
[1,] a  2.076986
[2,] b -1.895834
```

The 'by' in data.table is fast. Let's compare it to tapply:

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

```
   user  system elapsed
 18.233   1.024  19.383
```

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

```
   user  system elapsed
  0.444   0.180   0.620
```

```
> head(tt)
```

```
       A        B        C        D        E        F
192345.3 192182.2 192334.1 192540.6 192261.9 192635.0
```

```
> head(ss)
```

```
     x       V1
[1,] A 192345.3
[2,] B 192182.2
[3,] C 192334.1
[4,] D 192540.6
[5,] E 192261.9
[6,] F 192635.0
```

[2]Subsetting a keyed data.table by an n-column data.table is consistent with subsetting a n-dimension array by an n-column matrix

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

[1] TRUE
```

At 0.620 sec, this was **31** times faster than 19.383 sec, and produced precisely the same result.
Next, let's group by two columns:

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

   user  system elapsed
 19.745   1.277  21.062

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

   user  system elapsed
  0.520   0.320   0.844

> tt[1:5,1:5]

         a        b        c        d        e
A 7411.580 7416.870 7494.803 7407.473 7433.573
B 7394.034 7386.227 7417.540 7386.241 7384.024
C 7440.906 7454.504 7362.823 7392.947 7397.988
D 7401.396 7388.128 7344.435 7343.680 7383.997
E 7382.219 7374.323 7488.427 7401.267 7444.362

> head(ss)

     x y       V1
[1,] A a 7411.580
[2,] A b 7416.870
[3,] A c 7494.803
[4,] A d 7407.473
[5,] A e 7433.573
[6,] A f 7373.358

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

[1] TRUE
```

This was **24** times faster, and the syntax a little simpler and easier to read.
The following features are mentioned only briefly here; further examples are in the FAQs.

- To return several expressions, pass a list() to j.
- Each item of the list is recycled to match the length of the longest item.
- You can pass a list() of expressions of column names to by.

3. Fast time series join

This is also known as last observation carried forward (LOCF) or a *rolling join*.

Recall that x[i] is a join between data.table x and data.table i. If i has 2 columns, the first column is matched to the first column of the key of x, and the 2nd column to the 2nd. An equi-join is performed, meaning that the values must be equal.

The syntax for fast rolling join is

`x[i,roll=TRUE]`

As before the first column of i is matched to x where the values are equal. The last column of i though, the 2nd one in this example, is treated specially. If no match is found, then the row before is returned, provided the first column still matches.

For examples see `example("[.data.table")`

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: `vignette("datatable-faq")`
- The performance tests contain more examples: `vignette("datatable-timings")`
- test.data.table contains over 150 low level tests of the features: `test.data.table()`
- Website: http://datatable.r-forge.r-project.org/
- Presentations:
  - http://files.meetup.com/1406240/Data%20munging%20with%20SQL%20and%20R.pdf
  - http://www.londonr.org/LondonR-20090331/data.table.LondonR.pdf
- YouTube Demo: http://www.youtube.com/watch?v=rvT8XThGA8o
- R-Forge commit logs: http://lists.r-forge.r-project.org/pipermail/datatable-commits/
- Mailing list : datatable-help@lists.r-forge.r-project.org
- User reviews : http://crantastic.org/packages/data-table