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

Matthew Dowle

May 3, 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 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 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 -1.8094206
2 b  0.5852195
3 b  0.7463882
4 a -1.1401348
5 a  0.2978855
```

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  0.5637146
[2,] b -0.3064082
[3,] b -0.7271670
[4,] a -2.0425226
[5,] a -0.6564449
```

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 converts character vectors to factor.

```
> sapply(dt,class)

        x         v
 "factor" "numeric"
```

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 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

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? 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. 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  0.5637146
[2,] b -0.3064082
[3,] b -0.7271670
[4,] a -2.0425226
[5,] a -0.6564449
```

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

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

```
     x          v
[1,] b -0.3064082
```

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

```
     x          v
[1,] b  0.5637146
[2,] b -0.3064082
[3,] b -0.7271670
```

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 -2.0425226
[2,] a -0.6564449
[3,] b  0.5637146
[4,] b -0.3064082
[5,] b -0.7271670
```

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 0.5637146
```

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 0.5637146

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

     x         v
[1,] b -0.727167

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

     x          v
[1,] b  0.5637146
[2,] b -0.3064082
[3,] b -0.7271670
```

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
  4.204   1.836   6.042

> head(DF,3)

  x y         v
1 A a 0.9965158
2 A a 0.1437145
3 A a 0.9244377

> tail(DF,3)

         x y         v
10000066 Z z 0.6737062
10000067 Z z 0.2204626
10000068 Z z 0.4198879

> dim(DF)

[1] 10000068        3
```

We might say that R has created a 3 column table and *inserted* 10,000,068 rows. It took 6.042 secs, so it inserted 1,655,092 rows per second. That is normal in base R.

Lets 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
  4.216   0.976   5.194

> head(ans1,3)

        x y         v
6642058 R h 0.9817642
6642059 R h 0.6806054
6642060 R h 0.1608450

> dim(ans1)

[1] 14793     3
```

Now we 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
  0.008   0.004   0.010

> mapply(identical,ans1,ans2)

   x    y    v
TRUE TRUE TRUE
```

At 0.010 seconds, this was **519** times faster than 5.194 seconds, and produced precisely the same result. If you are thinking that a few seconds is not much to save, its the relative speedup thats 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].

What does the J() do?

Was it really this, or was it something slow about using data.frame syntax in a data.table? Its exactly the same :

We can do vector scans in data.table too.

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

   user  system elapsed
  4.217   0.964   5.180

> system.time(ans2 <- DT[DT$x=="R" & DT$y=="h",])

   user  system elapsed
  4.228   1.032   5.260

> mapply(identical,ans1,ans2)

   x    y    v
TRUE TRUE TRUE

> system.time(ans3 <- DT[x=="R" & y=="h",])

   user  system elapsed
  4.252   1.028   5.281

> identical(ans2,ans3)
```

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

```
[1] TRUE
```

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 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.

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. [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 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] -3.168828
```

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 -2.6989675
[2,] b -0.4698606
```

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
  9.245   0.940  10.187
```

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

[2] 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

```
   user  system elapsed
  0.528   0.288   0.816
```

```
> head(tt)
```

```
       A        B        C        D        E        F
192321.2 191800.4 192785.3 192234.7 192159.0 192466.3
```

```
> head(ss)
```

```
     x       V1
[1,] A 192321.2
[2,] B 191800.4
[3,] C 192785.3
[4,] D 192234.7
[5,] E 192159.0
[6,] F 192466.3
```

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

```
[1] TRUE
```

At 0.816sec, this was **12** times faster than 10.187sec, 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
 10.556   1.324  11.880
```

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

```
   user  system elapsed
  0.552   0.360   0.917
```

```
> tt[1:5,1:5]
```

```
         a        b        c        d        e
A 7409.614 7367.789 7398.255 7442.025 7364.313
B 7408.831 7366.444 7428.906 7316.498 7338.072
C 7389.100 7413.650 7442.931 7392.206 7417.141
D 7414.040 7437.353 7412.524 7432.594 7307.279
E 7407.788 7347.737 7413.891 7408.933 7382.637
```

```
> head(ss)
```

```
     x y       V1
[1,] A a 7409.614
[2,] A b 7367.789
[3,] A c 7398.255
[4,] A d 7442.025
[5,] A e 7364.313
[6,] A f 7426.731
```

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

```
[1] TRUE
```

This was **12** 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 (no content yet): `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