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

Matthew Dowle

Revised: March 29, 2012
(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 in which 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  1.7869131
2 b  0.4978505
3 b -1.9666172
4 a  0.7013559
5 a -0.4727914
```

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 -1.0678237
[2,] b -0.2179749
[3,] b -1.0260044
[4,] a -0.7288912
[5,] a -0.6250393
```

Observe that a `data.table` prints the row numbers slightly differently. There is nothing significant about that. We can easily 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 `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.frames`, `data.tables` 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()`.

To see the column types[1] :

```
> sapply(DT,class)
```

```
          x           v
"character"   "numeric"
```

You may have noticed the empty column KEY in the result of `tables()` earlier 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, character 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
```

[1]As from v1.8.0, `data.table()` no longer converts `character` to `factor`.

```
     x          v
[1,] b -1.0678237
[2,] b -0.2179749
[3,] b -1.0260044
[4,] a -0.7288912
[5,] a -0.6250393
```

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

```
> DT[2,]

     x          v
[1,] b -0.2179749

> DT[DT$x=="b",]

     x          v
[1,] b -1.0678237
[2,] b -0.2179749
[3,] b -1.0260044
```

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

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

Error in `[.data.table`(DT, "b", ) :
  When i is a data.table (or character vector), x must be keyed (i.e. sorted, and, marked as sorte
```

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

```
> setkey(DT,x)
> DT

     x          v
[1,] a -0.7288912
[2,] a -0.6250393
[3,] b -1.0678237
[4,] b -0.2179749
[5,] b -1.0260044
```

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 -1.0678237
[2,] b -0.2179749
[3,] b -1.0260044
```

By default all the rows in the group are returned[2]. The `mult` argument (short for *multiple*) allows the first or last row of the group to be returned instead.

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

     x         v
[1,] b -1.067824

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

     x         v
[1,] b -1.026004
```

The comma is optional.

```
> DT["b"]

     x          v
[1,] b -1.0678237
[2,] b -0.2179749
[3,] b -1.0260044
```

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
 23.517   1.940  25.552

> head(DF,3)

  x y          v
1 A a 0.04583117
2 A a 0.44220007
3 A a 0.79892485

> tail(DF,3)

         x y          v
10000066 Z z 0.05172761
10000067 Z z 0.75044331
10000068 Z z 0.91941278

> dim(DF)

[1] 10000068        3
```

We might say that R has created a 3 column table and *inserted* 10,000,068 rows. It took 25.552 secs, so it inserted 391,361 rows per second. This is normal in base R.

Let's extract an arbitrary group from `DF`:

[2] In contrast to a `data.frame` where only the first rowname is returned when the rownames contain duplicates.

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

   user  system elapsed
 13.013   0.936  13.990

> head(ans1,3)

        x y          v
6642058 R h 0.58033299
6642059 R h 0.89146948
6642060 R h 0.07484815

> 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")])   # binary search

   user  system elapsed
  0.024   0.000   0.024

> head(ans2,3)

     x y          v
[1,] R h 0.58033299
[2,] R h 0.89146948
[3,] R h 0.07484815

> dim(ans2)

[1] 14793     3

> identical(ans1$v, ans2$v)

[1] TRUE
```

At 0.024 seconds, this was **582** times faster than 13.990 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[3].

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
 12.860   0.988  13.886

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

   user  system elapsed
 12.884   0.888  13.811

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

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

```
    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.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`s called `J()`, short for join.

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

[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 a short, fast to write and easy to maintain. Passing a `data.table` into a `data.table` subset is analogous to `A[B]` syntax in base R where `A` is a matrix and `B` is a 2-column matrix[4]. In fact, the `A[B]` syntax in base R inspired the `data.table` package. 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 very similar to `X[Y]`, but there are some differences. See FAQ 1.12.

This first section has been about the first argument to `[`, 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] 4999524
```

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
   A 192321.8
   B 192111.6
   C 192099.9
   D 192186.6
   E 192507.0
   F 192246.6
   G 192455.8
   H 192180.3
   I 192370.5
```

[4] Subsetting a keyed `data.table` by a n-column `data.table` is consistent with subsetting a n-dimension array by a n-column matrix in base R

```
   J 192372.6
   K 192348.7
   L 192296.0
   M 192365.6
   N 192195.0
   O 192329.3
   P 192364.4
   Q 192527.5
   R 192273.9
   S 191894.6
   T 192364.7
   U 192133.7
   V 192362.7
   W 192562.6
   X 192112.0
   Y 192236.9
   Z 192303.7
cn x       V1
```

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
 18.250   1.016  19.299

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

   user  system elapsed
  0.488   0.168   0.660

> head(tt)

       A        B        C        D        E        F
192321.8 192111.6 192099.9 192186.6 192507.0 192246.6

> head(ss)

     x       V1
[1,] A 192321.8
[2,] B 192111.6
[3,] C 192099.9
[4,] D 192186.6
[5,] E 192507.0
[6,] F 192246.6

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

[1] TRUE
```

At 0.660 sec, this was **29** times faster than 19.299 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
 20.281   1.220  21.544

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

```
   user  system elapsed
  0.688   0.240   0.933
```

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

```
         a        b        c        d        e
A 7366.761 7364.410 7332.694 7395.945 7375.200
B 7396.300 7403.709 7400.095 7416.663 7377.056
C 7366.846 7420.488 7395.226 7367.150 7363.280
D 7412.372 7385.739 7411.949 7377.145 7433.720
E 7376.348 7373.090 7440.186 7441.798 7368.630
```

```
> head(ss)
```

```
     x y       V1
[1,] A a 7366.761
[2,] A b 7364.410
[3,] A c 7332.694
[4,] A d 7395.945
[5,] A e 7375.200
[6,] A f 7432.267
```

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

```
[1] TRUE
```

This was **23** times faster, and the syntax is a little simpler and easier to read.

The following features are mentioned only briefly here; further examples are in `?data.table` and the FAQ vignette.

- 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` e.g.
  `DT[,sum(v),by=list(month(dateCol),region)]`
  where calling `month()` on `dateCol` is what we mean by expressions of column names.
- Any R functions from any package can be used in `j` and `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 250 low level tests of the features
- 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