

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

Revised: October 2, 2014
(A later revision may be available on the [homepage](#))

Introduction

This vignette is aimed at those who are already familiar with creating and subsetting `data.frame` in R. We aim for this quick introduction to be readable in **10 minutes**, briefly covering a few features: 1. Keys; 2. Fast Grouping; and 3. Fast *ordered* join.

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  0.1913554
2 b  1.2556559
3 b -0.3871878
4 a  0.7918896
5 a -0.8527794
```

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  1.1279835
2: b -0.9044895
3: b -0.8075297
4: a  1.3814964
5: a -0.3773382
```

Observe that a `data.table` prints the row numbers with a colon so as to visually separate the row number from the first column. 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 NCOL MB COLS      KEY
[1,] CARS   50    2  1 speed,dist
[2,] DT     5    2  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 :

```
> sapply(DT, class)

      x      v
"character" "numeric"
```

You may have noticed the empty column `KEY` in the result of `tables()` above. This is the subject of the next section.

1. Keys

Let's start by considering `data.frame`, specifically `rownames`. We know that each row has exactly one row name. However, a person (for example) has at least two names, a first name and a second name. It's useful to organise a telephone directory sorted by surname then first name.

In `data.table`, a *key* consists of one *or more* columns. These columns may be integer, factor or numeric as well as 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. We can think of a key as like super-charged row names; i.e., multi-column and multi-type.

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 NCOL MB COLS      KEY
[1,] CARS   50    2  1 speed,dist
[2,] DT     5    2  1 x,v
Total: 2MB
```

```
> DT

      x      v
1: b  1.1279835
2: b -0.9044895
3: b -0.8075297
4: a  1.3814964
5: a -0.3773382
```

No keys have been set yet.

```
> DT[2,]      # select row 2
```

```
      x          v
1: b -0.9044895
```

```
> DT[x=="b",] # select rows where column x == "b"
```

```
      x          v
1: b  1.1279835
2: b -0.9044895
3: b -0.8075297
```

Aside: notice that we did not need to prefix `x` with `DT$x`. In `data.table` queries, we can use column names as if they are variables directly.

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 sorted)
```

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

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

```
> DT
```

```
      x          v
1: a  1.3814964
2: a -0.3773382
3: b  1.1279835
4: b -0.9044895
5: b -0.8075297
```

Notice that the rows in `DT` have now 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 NCOL MB COLS      KEY
[1,] CARS   50   2  1 speed,dist
[2,] DT     5   2  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  1.1279835
2: b -0.9044895
3: b -0.8075297
```

By default all the rows in the group are returned¹. 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  1.127983
```

¹In contrast to a `data.frame` where only the first rowname is returned when the rownames contain duplicates.

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

```
      x          v
1: b -0.8075297
```

Also, the comma is optional.

```
> DT["b"]
```

```
      x          v
1: b  1.1279835
2: b -0.9044895
3: b -0.8075297
```

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

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

```
[1] 14793
```

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

```
      user system elapsed
5.696    0.272    5.981
```

```
> head(DF,3)
```

```
      x y          v
1 A a 0.59188461
2 A a 0.09419788
3 A a 0.35631785
```

```
> tail(DF,3)
```

```
      x y          v
10000066 Z z 0.6131090
10000067 Z z 0.8320838
10000068 Z z 0.6368332
```

```
> dim(DF)
```

```
[1] 10000068      3
```

We might say that R has created a 3 column table and *inserted* 10,000,068 rows. It took 5.981 secs, so it inserted 1,671,972 rows per second. This is normal in base R. Notice that we set `stringsAsFactors=FALSE`. This makes it a little faster for a fairer comparison, but feel free to experiment.

Let's extract an arbitrary group from DF:

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

```
      user system elapsed
13.436    0.376   13.845
```

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

```

      x y      v
6642058 R h 0.7503649
6642059 R h 0.9192344
6642060 R h 0.6846727

```

```
> dim(ans1)
```

```
[1] 14793      3
```

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

```
> DT = as.data.table(DF)      # but normally use fread() or data.table() directly, originally
> system.time(setkey(DT,x,y)) # one-off cost, usually
```

```

user  system elapsed
0.432  0.040  0.475

```

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

```

user  system elapsed
0.012  0.000  0.010

```

```
> head(ans2,3)
```

```

      x y      v
1: R h 0.7503649
2: R h 0.9192344
3: R h 0.6846727

```

```
> dim(ans2)
```

```
[1] 14793      3
```

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

```
[1] TRUE
```

At 0.010 seconds, this was **1384** times faster than 13.845 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 significant².

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
6.428  0.132  6.572

```

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

```

user  system elapsed
12.592  0.340  13.024

```

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

```

      x  y  v
TRUE TRUE TRUE

```

²We wonder how many people are deploying parallel techniques to code that is vector scanning

If the phone book analogy helped, the **1384** 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 `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 a `list` (and `data.table` is a `list` too), we say that we are *joining*. In this case, we are joining DT to the 1 row, 2 column table returned by `list("R", "h")`. Since we do this a lot, there is an alias for `list`: `.()`.

```
> identical( DT[list("R", "h"), ],
+           DT[.("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 matrix³. In fact, the `A[B]` syntax in base R inspired the `data.table` package. There are other types of ordered joins 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 inside `DT[...]`, namely `i`. The next section is about the 2nd and 3rd arguments: `j` and `by`.

2. Fast grouping

The second argument to `DT[...]` is `j` and may consist of one or more expressions whose arguments are (unquoted) column names, as if the column names were variables. Just as we saw earlier in `i` as well.

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

```
[1] 5000407
```

When we supply a `j` expression and a 'by' expression, the `j` expression is repeated for each 'by' group.

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

```
   x      V1
1: A 192394.9
2: B 192440.5
3: C 192119.2
4: D 192122.4
5: E 192349.2
6: F 192197.9
7: G 192277.7
8: H 192574.9
9: I 192246.2
10: J 192648.9
11: K 192287.5
```

³Subsetting 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

```

12: L 192049.7
13: M 192269.9
14: N 192338.9
15: O 192740.3
16: P 192241.7
17: Q 192269.7
18: R 192373.5
19: S 192115.2
20: T 192625.8
21: U 192179.6
22: V 192301.1
23: W 192043.1
24: X 192340.7
25: Y 192453.1
26: Z 192405.4
    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
7.360  0.976  8.529

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

  user  system elapsed
0.656  0.048  0.706

> head(tt)

      A      B      C      D      E      F
192394.9 192440.5 192119.2 192122.4 192349.2 192197.9

> head(ss)

  x      V1
1: A 192394.9
2: B 192440.5
3: C 192119.2
4: D 192122.4
5: E 192349.2
6: F 192197.9

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

[1] TRUE

```

At 0.706 sec, this was **12** times faster than 8.529 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
10.156  1.444 11.661

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

  user  system elapsed
0.776  0.052  0.826

```

```

> tt[1:5,1:5]

      a      b      c      d      e
A 7340.570 7370.085 7466.794 7350.710 7351.832
B 7383.704 7337.640 7345.794 7458.266 7413.636
C 7387.638 7363.944 7378.682 7366.018 7484.329
D 7413.228 7364.502 7388.166 7432.694 7467.051
E 7397.637 7402.158 7382.014 7336.074 7396.296

> head(ss)

  x y      V1
1: A a 7340.570
2: A b 7370.085
3: A c 7466.794
4: A d 7350.710
5: A e 7351.832
6: A f 7408.410

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

[1] TRUE

```

This was **14** times faster, and the syntax is a little simpler and easier to read.

3. Fast ordered joins

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

Recall that `X[Y]` is a join between `data.table X` and `data.table Y`. If `Y` 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 by default, meaning that the values must be equal.

Instead of an equi-join, a rolling join is :

```
X[Y,roll=TRUE]
```

As before the first column of `Y` is matched to `X` where the values are equal. The last join column in `Y` 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.

Further controls are rolling forwards, backwards, nearest and limited staleness.

For examples type `example(data.table)` and follow the output at the prompt.