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

Revised: September 18, 2015
(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 -0.1758447
2 b -1.1006405
3 b  1.2296532
4 a -1.7714559
5 a -0.2651259
```

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 0.7693975
2: b 2.6219549
3: b 0.6689401
4: a 0.7741039
5: a 0.4980158
```

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., mult-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 0.7693975
2: b 2.6219549
3: b 0.6689401
4: a 0.7741039
5: a 0.4980158
```

No keys have been set yet.

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

```
   x        v
1: b 2.621955
```

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

```
   x         v
1: b 0.7693975
2: b 2.6219549
3: b 0.6689401
```

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 sorte
```

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

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

```
   x         v
1: a 0.7741039
2: a 0.4980158
3: b 0.7693975
4: b 2.6219549
5: b 0.6689401
```

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 0.7693975
2: b 2.6219549
3: b 0.6689401
```

By default all the rows in the group are returned[1]. 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 0.7693975
```

[1] In 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 0.6689401
```

Also, the comma is optional.

```
> DT["b"]

   x         v
1: b 0.7693975
2: b 2.6219549
3: b 0.6689401
```

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*.

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

[1] 14793

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

   user  system elapsed
  0.448   0.008   0.457

> head(DF,3)

  x y         v
1 A a 0.5310106
2 A a 0.1980941
3 A a 0.8835322

> tail(DF,3)

         x y         v
10000066 Z z 0.6231946
10000067 Z z 0.4410910
10000068 Z z 0.9604099

> dim(DF)

[1] 10000068        3
```

We might say that R has created a 3 column table and *inserted* 10,000,068 rows. It took 0.457 secs, so it inserted 21,881,986 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
  0.528   0.016   0.544

> head(ans1,3)
```

```
        x y         v
6642058 R h 0.7437625
6642059 R h 0.5702467
6642060 R h 0.3618726

> 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
  0.052   0.000   0.052

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

   user  system elapsed
  0.004   0.000   0.001

> head(ans2,3)

   x y         v
1: R h 0.7437625
2: R h 0.5702467
3: R h 0.3618726

> dim(ans2)

[1] 14793     3

> identical(ans1$v, ans2$v)

[1] TRUE
```

At 0.001 seconds, this was **544** times faster than 0.544 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[2].

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
  0.464   0.016   0.479

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

   user  system elapsed
  0.536   0.008   0.543

> mapply(identical,ans1,ans2)

   x    y    v
TRUE TRUE TRUE
```

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

If the phone book analogy helped, the **544** 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 matrix[3]. 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] 4999637
```

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 192213.2
 2: B 192183.3
 3: C 192601.7
 4: D 192308.0
 5: E 192428.5
 6: F 192071.0
 7: G 192403.8
 8: H 192423.9
 9: I 192024.5
10: J 192063.1
11: K 192340.2
```

[3]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

```
12: L 192421.5
13: M 192470.2
14: N 192045.5
15: O 192166.7
16: P 192459.4
17: Q 192307.1
18: R 192288.1
19: S 192274.7
20: T 192380.5
21: U 192191.0
22: V 192170.7
23: W 192257.5
24: X 192401.6
25: Y 192429.4
26: Z 192312.4
    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
  0.704   0.064   0.767

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

   user  system elapsed
  0.080   0.000   0.078

> head(tt)

       A        B        C        D        E        F
192213.2 192183.3 192601.7 192308.0 192428.5 192071.0

> head(ss)

   x       V1
1: A 192213.2
2: B 192183.3
3: C 192601.7
4: D 192308.0
5: E 192428.5
6: F 192071.0

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

[1] TRUE
```

At 0.078 sec, this was **9** times faster than 0.767 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
  1.076   0.080   1.155

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

   user  system elapsed
  0.104   0.000   0.106
```

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

         a        b        c        d        e
A 7382.299 7424.815 7345.469 7347.148 7356.512
B 7360.890 7383.625 7348.990 7381.238 7430.159
C 7432.864 7433.346 7398.234 7429.309 7406.106
D 7387.108 7398.470 7390.907 7402.977 7393.608
E 7399.820 7435.018 7374.863 7396.102 7399.262

> head(ss)

   x y       V1
1: A a 7382.299
2: A b 7424.815
3: A c 7345.469
4: A d 7347.148
5: A e 7356.512
6: A f 7411.005

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

[1] TRUE
```

This was **10** 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.