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

Matthew Dowle

April 1, 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 Merging/Joining. 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 -0.6069171
2 b -1.6814848
3 b  1.0631077
4 a  1.0677418
5 a -1.7019284
```

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 -1.48273651
[2,] b -0.69569423
[3,] b  0.06644914
[4,] a -1.42533240
[5,] a  0.11513952
```

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().

You may have noticed the empty column KEY. 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? No, that is not what we are used to in a data.frame. We know that each row has at most one name, but never *more* than one name. A person has at least two names, a first name and a second name. That is useful to organise a phone directory of people. 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 -1.48273651
[2,] b -0.69569423
[3,] b  0.06644914
[4,] a -1.42533240
[5,] a  0.11513952
```

No keys have been set yet. We *can* use data.frame syntax without a key.

```
> dt[2,]

     x          v
[1,] b -0.6956942

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

     x           v
[1,] b -1.48273651
[2,] b -0.69569423
[3,] b  0.06644914
```

But since there are no rownames the following does not work.

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

```
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)
> dt

     x           v
[1,] a -1.42533240
[2,] a  0.11513952
[3,] b -1.48273651
[4,] b -0.69569423
[5,] b  0.06644914
```

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 -1.482737
```

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 -1.482737

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

     x          v
[1,] b 0.06644914

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

     x           v
[1,] b -1.48273651
[2,] b -0.69569423
[3,] b  0.06644914
```

Lets now create a data.table with a 2 column key. We can do this in one step this time by using the key argument of data.table(). We will also make it large enough to demonstrate the difference between a vector scan and a binary search.

```
> n=1e6
> dt2 = data.table(x=sample(LETTERS,n,replace=TRUE),
+                  y=sample(LETTERS,n,replace=TRUE),
+                  v=rnorm(n),
+                  key="x,y")
> head(dt2)
```

```
     x y          v
[1,] A A  0.1277313
[2,] A A -0.2073835
[3,] A A -1.9834157
[4,] A A  2.2463114
[5,] A A -1.7608138
[6,] A A -2.3686216
```

```
> tables()
```

```
     NAME      NROW MB COLS       KEY
[1,] cars        50  1 speed,dist
[2,] dt           5  1 x,v        x
[3,] dt2  1,000,000 16 x,y,v      x,y
Total: 18MB
```

TO DO- explain difference between vector scan and binary search. Advise to unlearn ==.
First section was to do with i. Second section to do with j.

2. Fast grouping

So far we have dealt with the first argument inside []. The first is call i. The second is called j and may be one or more expressions of column names as if the column names were variables.

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

subsection : Scope

Think of the subset as an environment where all the column names are variables. When a variable is used in the j expression, it is looked for in the following order :

1. The scope of the subset i.e. the column names
2. The scope of the calling frame e.g. the line that appears before the data.table query
3. TO CHECK. Does it ripple up or go straight to .GlobalEnv?
4. The global environment

This is "lexical scoping" explained by R FAQ 3.3.1
TO DO - examples.

3. Fast joining/merging

x[y] is a join between x and y, a subset of x defined by y merge(x,y) is the same but contains the union of columns. When merge is passed data.table's, it operates much faster than base merge on data.frame's.

TO DO - examples.

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 in an easy to digest manner
- The performance tests are more complex real world examples
- test.data.table contains over 100 low level tests of the features.
- Presentations
- YouTube Demo
- R-Forge commit logs: `http://lists.r-forge.r-project.org/pipermail/datatable-commits/`
- Website: `http://datatable.r-forge.r-project.org/`
- Mailing list : datatable-help@lists.r-forge.r-project.org