FAQs about the **data.table** package in R

Revised: September 18, 2015
(A later revision may be available on the homepage)

The first section, Beginner FAQs, is intended to be read in order, from start to finish.

Contents

1 Beginner FAQs

1.1 Why does `DT[,5]` return `5`?

Because by default, unlike a `data.frame`, the 2nd argument is an *expression* which is evaluated within the scope of `DT`. 5 evaluates to 5. It is generally bad practice to refer to columns by number rather than name. If someone else comes along and reads your code later, they may have to hunt around to find out which column is number 5. Furthermore, if you or someone else changes the column ordering of `DT` higher up in your `R` program, you might get bugs if you forget to change all the places in your code which refer to column number 5.

Say column 5 is called ”region”, just do `DT[,region]` instead. Notice there are no quotes around the column name. This is what we mean by `j` being evaluated within the scope of the `data.table`. That scope consists of an environment where the column names are variables.

You can place *any* `R` expression in `j`; e.g., `DT[,colA*colB/2]`. Further, `j` may be a set of `R` expressions (including calls to any `R` package) wrapped with `list()`, `.()` or an anonymous code block wrapped with braces . A simple example is `DT[,fitdistr(d1-d1,"normal")]`.

Having said this, there are some circumstances where referring to a column by number is ok, such as a sequence of columns. In these situations just do `DT[,5:10,with=FALSE]` or `DT[,c(1,4,10),with=FALSE]`. See `?data.table` for an explanation of the `with` argument. It lets you use `data.table` the same way as `data.frame`, when you need to.

Note that `with()` has been a base function for a long time. That's one reason we say `data.table` builds upon base functionality. There is little new here really, `data.table` is just making use of `with()` and building it into the syntax.

1.2 Why does `DT[,"region"]` return `"region"`?

See answer to 1.1 above. Try `DT[,region]` instead. Or `DT[,"region",with=FALSE]`.

1.3 Why does `DT[,region]` return a vector? I'd like a 1-column `data.table`. There is no `drop` argument like I'm used to in `data.frame`.

Try `DT[,.(region)]` instead. `.()` is an alias for `list()` and ensures a `data.table` is returned.

1.4 Why does `DT[,x,y,z]` not work? I wanted the 3 columns `x`,`y` and `z`.

The `j` expression is the 2nd argument. The correct way to do this is `DT[,.(x,y,z)]`.

1.5 I assigned a variable `mycol="x"` but then `DT[,mycol]` returns `"x"`. How do I get it to look up the column name contained in the `mycol` variable?

This is what we mean when we say the `j` expression 'sees' objects in the calling scope. The variable `mycol` does not exist as a column name of `DT` so `R` then looked in the calling scope and found `mycol` there and returned its value `"x"`. This is correct behaviour. Had `mycol` been a column name, then that column's data would have been returned. What you probably meant was `DT[,mycol,with=FALSE]`, which will return the `x` column's data as you wanted. Alternatively, since a `data.table` *is* a `list`, too, you can write `DT[["x"]]` or `DT[[mycol]]`.

1.6 Ok but I don't know the expressions in advance. How do I programatically pass them in?

To create expressions use the `quote()` function. We refer to these as *quote()-ed* expressions to save confusion with the double quotes used to create a character vector such as `c("x")`. The simplest quote()-ed expression is just one column name :

```
q = quote(x)
DT[,eval(q)] # returns the column x as a vector
```

```
q = quote(list(x))
DT[,eval(q)] # returns the column x as a 1-column data.table
```

Since these are *expressions*, we are not restricted to column names only :

```
q = quote(mean(x))
DT[,eval(q)] # identical to DT[,mean(x)]
q = quote(list(x,sd(y),mean(y*z)))
DT[,eval(q)] # identical to DT[,list(x,sd(y),mean(y*z))]
```

However, if it's just simply a vector of column names you need, it may be simpler to pass a character vector to `j` and use `with=FALSE`.

To pass an expression into your own function, one idiom is as follows :

```
> DT = as.data.table(iris)
> setkey(DT,Species)
> myfunction = function(dt, expr) {
+     e = substitute(expr)
+     dt[,eval(e),by=Species]
+ }
> myfunction(DT,sum(Sepal.Width))

      Species    V1
1:     setosa 171.4
2: versicolor 138.5
3:  virginica 148.7
```

`quote()` and `eval()` are like macros in other languages. Instead of `j=myfunction()` (which won't work without laboriously passing in all the arguments) it's `j=eval(mymacro)`. This can be more efficient than a function call, and convenient. When data.table sees `j=eval(mymacro)` it knows to find `mymacro` in calling scope so as not to be tripped up if a column name happens to be called `mymacro`, too.

For example, let's make sure that exactly the same `j` is run for a set of different grouping criteria :

```
> DT = as.data.table(iris)
> whatToRun = quote( .(AvgWidth = mean(Sepal.Width),
+                      MaxLength = max(Sepal.Length)) )
> DT[, eval(whatToRun), by=Species]

      Species AvgWidth MaxLength
1:     setosa    3.428       5.8
2: versicolor    2.770       7.0
3:  virginica    2.974       7.9

> DT[, eval(whatToRun), by=.(FirstLetter=substring(Species,1,1))]

   FirstLetter AvgWidth MaxLength
1:           s    3.428       5.8
2:           v    2.872       7.9

> DT[, eval(whatToRun), by=.(Petal.Width=round(Petal.Width,0))]

   Petal.Width AvgWidth MaxLength
1:           0 3.426531       5.8
2:           1 2.708108       7.0
3:           2 2.976562       7.9
```

1.7 What are the benefits of being able to use column names as if they are variables inside `DT[...]`?

`j` doesn't have to be just column names. You can write any R *expression* of column names directly as the `j`; e.g., `DT[,mean(x*y/z)]`. The same applies to `i`; e.g., `DT[x>1000, sum(y*z)]`. This runs the `j` expression on the set of rows where the `i` expression is true. You don't even need to return data; e.g., `DT[x>1000, plot(y,z)]`. Finally, you can do `j` by group by adding `by=`; e.g., `DT[x>1000, sum(y*z), by=w]`. This runs `j` for each group in column `w` but just over the rows where `x>1000`. By placing the 3 parts of the query (where, select and group by) inside the square brackets, `data.table` sees this query as a whole before any part of it is evaluated. Thus it can optimize the query for performance.

1.8 OK, I'm starting to see what `data.table` is about, but why didn't you enhance `data.frame` in R? Why does it have to be a new package?

As FAQ 1.1 highlights, `j` in `[.data.table` is fundamentally different from `j` in `[.data.frame`. Even something as simple as `DF[,1]` would break existing code in many packages and user code. This is by design. We want it to work this way for more complicated syntax to work. There are other differences, too (see FAQ 2.17).

Furthermore, `data.table` *inherits* from `data.frame`. It *is* a `data.frame`, too. A `data.table` can be passed to any package that only accepts `data.frame` and that package can use `[.data.frame` syntax on the `data.table`.

We *have* proposed enhancements to R wherever possible, too. One of these was accepted as a new feature in R 2.12.0 :

> unique() and match() are now faster on character vectors where all elements are in the global CHARSXP cache and have unmarked encoding (ASCII). Thanks to Matt Dowle for suggesting improvements to the way the hash code is generated in unique.c.

A second proposal was to use memcpy in duplicate.c, which is much faster than a for loop in C. This would improve the *way* that R copies data internally (on some measures by 13 times). The thread on r-devel is here : http://tolstoy.newcastle.edu.au/R/e10/devel/10/04/0148.html.

1.9 Why are the defaults the way they are? Why does it work the way it does?

The simple answer is because the main author originally designed it for his own use. He wanted it that way. He finds it a more natural, faster way to write code, which also executes more quickly.

1.10 Isn't this already done by `with()` and `subset()` in base?

Some of the features discussed so far are, yes. The package builds upon base functionality. It does the same sorts of things but with less code required and executes many times faster if used correctly.

1.11 Why does `X[Y]` return all the columns from `Y` too? Shouldn't it return a subset of `X`?

This was changed in v1.5.3. `X[Y]` now includes `Y`'s non-join columns. We refer to this feature as *join inherited scope* because not only are `X` columns available to the j expression, so are `Y` columns. The downside is that `X[Y]` is less efficient since every item of `Y`'s non-join columns are duplicated to match the (likely large) number of rows in `X` that match. We therefore strongly encourage `X[Y,j]` instead of `X[Y]`. See next FAQ.

1.12 What is the difference between X[Y] and merge(X,Y)?

X[Y] is a join, looking up X's rows using Y (or Y's key if it has one) as an index.
Y[X] is a join, looking up Y's rows using X (or X's key if it has one) as an index.
merge(X,Y)[1] does both ways at the same time. The number of rows of X[Y] and Y[X] usually differ; whereas the number of rows returned by merge(X,Y) and merge(Y,X) is the same.

BUT that misses the main point. Most tasks require something to be done on the data after a join or merge. Why merge all the columns of data, only to use a small subset of them afterwards? You may suggest merge(X[,ColsNeeded1],Y[,ColsNeeded2]), but that takes copies of the subsets of data and it requires the programmer to work out which columns are needed. X[Y,j] in data.table does all that in one step for you. When you write X[Y,sum(foo*bar)], data.table automatically inspects the j expression to see which columns it uses. It will only subset those columns only; the others are ignored. Memory is only created for the columns the j uses and Y columns enjoy standard R recycling rules within the context of each group. Let's say foo is in X and bar is in Y (along with 20 other columns in Y). Isn't X[Y,sum(foo*bar)] quicker to program and quicker to run than a merge followed by a subset?

1.13 Anything else about X[Y,sum(foo*bar)]?

This behaviour changed in v1.9.4 (Sep 2014). It now does the X[Y] join and then runs sum(foo*bar) over all the rows; i.e., X[Y][,sum(foo*bar)]. It used to run j for each *group* of X that each row of Y matches to. That can still be done as it's very useful but you now need to be explicit and specify by=.EACHI; i.e., X[Y,sum(foo*bar),by=.EACHI]. We call this *grouping by each i*. For example, and making it complicated by using *join inherited scope*, too :

```
> X = data.table(grp=c("a","a","b","b","b","c","c"), foo=1:7)
> setkey(X,grp)
> Y = data.table(c("b","c"), bar=c(4,2))
> X

   grp foo
1:   a   1
2:   a   2
3:   b   3
4:   b   4
5:   b   5
6:   c   6
7:   c   7

> Y

   V1 bar
1:  b   4
2:  c   2

> X[Y,sum(foo*bar)]

[1] 74

> X[Y,sum(foo*bar),by=.EACHI]

   grp V1
1:   b 48
2:   c 26
```

[1] Here we mean either the merge *method* for data.table or the merge method for data.frame since both methods work in the same way in this respect. See ?merge.data.table and FAQ 2.24 for more information about method dispatch.

1.14 That's nice. How did you manage to change it?

The request to change came from users. The feeling was that if a query is doing grouping then an explicit 'by=' should be present for code readability reasons. An option is provided to return the old behaviour: `options(datatable.old.bywithoutby)`, by default FALSE. This enables upgrading to test the other new features / bug fixes in v1.9.4, with later migration of any by-without-by queries when ready (by adding `by=.EACHI` to them). We retained 47 pre-change tests and added them back as new tests, tested under `options(datatable.old.bywithoutby=TRUE)`. After a year we'll remove this option.

Of the 66 packages on CRAN or Bioconductor that depend or import data.table at the time of releasing v1.9.4, only one was affected by the change. That could be because many packages don't have comprehensive tests, or just that grouping by each row in i wasn't being used much by downstream packages. We always test the new version with all dependent packages before release and coordinate any changes with those maintainers. So this release was quite straightforward in that regard.

Another compelling reason to make the change was that previously, there was no efficient way to achieve what `X[Y,sum(foo*bar)]` does now. You had to write `X[Y][,sum(foo*bar)]`. That was suboptimal because `X[Y]` joined all the columns and passed them all to the second compound query without knowing that only foo and bar are needed. To solve that efficiency problem, extra programming effort was required: `X[Y,list(foo,bar)][,sum(foo*bar)]`. The change to `by=.EACHI` has simplified this by allowing both queries to be expressed inside a single `DT[...]` query for efficieny.

2 General syntax

2.1 How can I avoid writing a really long `j` expression? You've said I should use the column *names*, but I've got a lot of columns.

When grouping, the `j` expression can use column names as variables, as you know, but it can also use a reserved symbol `.SD` which refers to the **S**ubset of the **D**`ata.table` for each group (excluding the grouping columns). So to sum up all your columns it's just `DT[,lapply(.SD,sum),by=grp]`. It might seem tricky, but it's fast to write and fast to run. Notice you don't have to create an anonymous `function`. The `.SD` object is efficiently implemented internally and more efficient than passing an argument to a function. But if the `.SD` symbol appears in `j` then `data.table` has to populate `.SD` fully for each group even if `j` doesn't use all of it. So please don't do this, for example, `DT[,sum(.SD[["sales"]]),by=grp]`. That works but is inefficient and inelegant. This is what was intended: `DT[,sum(sales),by=grp]` and could be 100's of times faster. If you do use all the data in `.SD` for each group (such as in `DT[,lapply(.SD,sum),by=grp]`) then that's very good usage of `.SD`. Also see `?data.table` for the `.SDcols` argument which allows you to specify a subset of columns for `.SD`.

2.2 Why is the default for `mult` now `"all"`?

In v1.5.3 the default was changed to `"all"`. When `i` (or `i`'s key if it has one) has fewer columns than `x`'s key, `mult` was already set to `"all"` automatically. Changing the default makes this clearer and easier for users as it came up quite often.

In versions up to v1.3, `"all"` was slower. Internally, `"all"` was implemented by joining using `"first"`, then again from scratch using `"last"`, after which a diff between them was performed to work out the span of the matches in `x` for each row in `i`. Most often we join to single rows, though, where `"first"`,`"last"` and `"all"` return the same result. We preferred maximum performance for the majority of situations so the default chosen was `"first"`. When working with a non-unique key (generally a single column containing a grouping variable), `DT["A"]` returned the first row of that group so `DT["A",mult="all"]` was needed to return all the rows in that group.

In v1.4 the binary search in C was changed to branch at the deepest level to find first and last. That branch will likely occur within the same final pages of RAM so there should no longer be a speed disadvantage in defaulting `mult` to `"all"`. We warned that the default might change and made the change in v1.5.3.

A future version of `data.table` may allow a distinction between a key and a *unique key*. Internally `mult="all"` would perform more like `mult="first"` when all `x`'s key columns were joined to and `x`'s key was a unique key. `data.table` would need checks on insert and update to make sure a unique key is maintained. An advantage of specifying a unique key would be that `data.table` would ensure no duplicates could be inserted, in addition to performance.

2.3 I'm using `c()` in the `j` and getting strange results.

This is a common source of confusion. In `data.frame` you are used to, for example:

```
> DF = data.frame(x=1:3,y=4:6,z=7:9)
> DF

  x y z
1 1 4 7
2 2 5 8
3 3 6 9

> DF[,c("y","z")]

  y z
1 4 7
2 5 8
3 6 9
```

which returns the two columns. In `data.table` you know you can use the column names directly and might try :

```
> DT = data.table(DF)
> DT[,c(y,z)]

[1] 4 5 6 7 8 9
```

but this returns one vector. Remember that the `j` expression is evaluated within the environment of `DT` and `c()` returns a vector. If 2 or more columns are required, use `list()` or `.()` instead:

```
> DT[,.(y,z)]

   y z
1: 4 7
2: 5 8
3: 6 9
```

`c()` can be useful in a `data.table` too, but its behaviour is different from that in `[.data.frame`.

2.4 I have built up a complex table with many columns. I want to use it as a template for a new table; i.e., create a new table with no rows, but with the column names and types copied from my table. Can I do that easily?

Yes. If your complex table is called `DT`, try `NEWDT = DT[0]`.

2.5 Is a null data.table the same as DT[0]?

No. By ”null data.table” we mean the result of `data.table(NULL)` or `as.data.table(NULL)`; i.e.,

```
> data.table(NULL)

Null data.table (0 rows and 0 cols)

> data.frame(NULL)

data frame with 0 columns and 0 rows

> as.data.table(NULL)

Null data.table (0 rows and 0 cols)

> as.data.frame(NULL)

data frame with 0 columns and 0 rows

> is.null(data.table(NULL))

[1] FALSE

> is.null(data.frame(NULL))

[1] FALSE
```

The null `data.table|frame` is `NULL` with some attributes attached, making it not NULL anymore. In R only pure `NULL` is `NULL` as tested by `is.null()`. When referring to the ”null data.table” we use lower case null to help distinguish from upper case `NULL`. To test for the null data.table, use `length(DT)==0` or `ncol(DT)==0` (`length` is slightly faster as it's a primitive function). An *empty* data.table (`DT[0]`) has one or more columns, all of which are empty. Those empty columns still have names and types.

```
> DT = data.table(a=1:3,b=c(4,5,6),d=c(7L,8L,9L))
> DT[0]

Empty data.table (0 rows) of 3 cols: a,b,d

> sapply(DT[0],class)

        a         b         d
"integer" "numeric" "integer"
```

2.6 Why has the DT() alias been removed?

`DT` was introduced originally as a wrapper for a list of `j` expressions. Since `DT` was an alias for `data.table`, this was a convenient way to take care of silent recycling in cases where each item of the `j` list evaluated to different lengths. The alias was one reason grouping was slow, though. As of v1.3, `list()` or `.()` should be passed instead to the `j` argument. These are much faster, especially when there are many groups. Internally, this was a nontrivial change. Vector recycling is now done internally, along with several other speed enhancements for grouping.

2.7 But my code uses j=DT(...) and it works. The previous FAQ says that DT() has been removed.

Then you are using a version prior to 1.5.3. Prior to 1.5.3 `[.data.table` detected use of `DT()` in the `j` and automatically replaced it with a call to `list()`. This was to help the transition for existing users.

2.8 What are the scoping rules for j expressions?

Think of the subset as an environment where all the column names are variables. When a variable foo is used in the j of a query such as X[Y,sum(foo)], foo is looked for in the following order :

1. The scope of X's subset; i.e., X's column names.
2. The scope of each row of Y; i.e., Y's column names (*join inherited scope*)
3. The scope of the calling frame; e.g., the line that appears before the data.table query.
4. Exercise for reader: does it then ripple up the calling frames, or go straight to globalenv()?
5. The global environment

This is *lexical scoping* as explained in R FAQ 3.3.1. The environment in which the function was created is not relevant, though, because there is *no function*. No anonymous *function* is passed to the j. Instead, an anonymous *body* is passed to the j; for example,

```
> DT = data.table(x=rep(c("a","b"),c(2,3)),y=1:5)
> DT

   x y
1: a 1
2: a 2
3: b 3
4: b 4
5: b 5

> DT[,{z=sum(y);z+3},by=x]

   x V1
1: a  6
2: b 15
```

Some programming languages call this a *lambda*.

2.9 Can I trace the j expression as it runs through the groups?

Try something like this:

```
> DT[,{
+   cat("Objects:",paste(objects(),collapse=","),"\n")
+   cat("Trace: x=",as.character(x)," y=",y,"\n")
+   sum(y)
+ },by=x]

Objects: Cfastmean,mean,print,x,y
Trace: x= a  y= 1 2
Objects: Cfastmean,mean,print,x,y
Trace: x= b  y= 3 4 5
   x V1
1: a  3
2: b 12
```

2.10 Inside each group, why are the group variables length 1?

In the previous FAQ, x is a grouping variable and (as from v1.6.1) has length 1 (if inspected or used in j). It's for efficiency and convenience. Therefore, there is no difference between the following two statements:

```
> DT[,.(g=1,h=2,i=3,j=4,repeatgroupname=x,sum(y)),by=x]

   x g h i j repeatgroupname V6
1: a 1 2 3 4               a  3
2: b 1 2 3 4               b 12

> DT[,.(g=1,h=2,i=3,j=4,repeatgroupname=x[1],sum(y)),by=x]

   x g h i j repeatgroupname V6
1: a 1 2 3 4               a  3
2: b 1 2 3 4               b 12
```

If you need the size of the current group, use `.N` rather than calling `length()` on any column.

2.11 Only the first 10 rows are printed, how do I print more?

There are two things happening here. First, if the number of rows in a `data.table` are large (`> 100` by default), then a summary of the `data.table` is printed to the console by default. Second, the summary of a large `data.table` is printed by takingthe top and bottom `n` rows of the `data.table` and only printing those. Both of these parameters (when to trigger a summary and how much of a table to use as a summary) are configurable by R's `options` mechanism, or by calling the `print` function directly.

For instance, to enforce the summary of a `data.table` to only happen when a `data.table` is greater than 50 rows, you could `options(datatable.print.nrows=50)`. To disable the summary-by-default completely, you could `options(datatable.print.nrows=Inf)`. You could also call `print` directly, as in `print(your.data.table, nrows=Inf)`.

If you want to show more than just the top (and bottom) 10 rows of a `data.table` summary (say you like 20), set `options(datatable.print.topn=20)`, for example. Again, you could also just call `print` directly, as in `print(your.data.table, topn=20)`

2.12 With an `X[Y]` join, what if `X` contains a column called `"Y"`?

When `i` is a single name such as `Y` it is evaluated in the calling frame. In all other cases such as calls to `.()` or other expressions, `i` is evaluated within the scope of `X`. This facilitates easy *self joins* such as `X[J(unique(colA)),mult="first"]`.

2.13 `X[Z[Y]]` is failing because `X` contains a column `"Y"`. I'd like it to use the table `Y` in calling scope.

The `Z[Y]` part is not a single name so that is evaluated within the frame of `X` and the problem occurs. Try `tmp=Z[Y];X[tmp]`. This is robust to `X` containing a column `"tmp"` because `tmp` is a single name. If you often encounter conflics of this type, one simple solution may be to name all tables in uppercase and all column names in lowercase, or some similar scheme.

2.14 Can you explain further why `data.table` is inspired by `A[B]` syntax in base?

Consider `A[B]` syntax using an example matrix `A` :

```
> A = matrix(1:12,nrow=4)
> A

     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

To obtain cells (1,2)=5 and (3,3)=11 many users (we believe) may try this first :

```
> A[c(1,3),c(2,3)]

     [,1] [,2]
[1,]    5    9
[2,]    7   11
```

That returns the union of those rows and columns, though. To reference the cells, a 2-column matrix is required. `?Extract` says :

> When indexing arrays by [a single argument i can be a matrix with as many columns as there are dimensions of x; the result is then a vector with elements corresponding to the sets of indices in each row of i.

Let's try again.

```
> B = cbind(c(1,3),c(2,3))
> B

     [,1] [,2]
[1,]    1    2
[2,]    3    3

> A[B]

[1]  5 11
```

A matrix is a 2-dimension structure with row names and column names. Can we do the same with names?

```
> rownames(A) = letters[1:4]
> colnames(A) = LETTERS[1:3]
> A

  A B  C
a 1 5  9
b 2 6 10
c 3 7 11
d 4 8 12

> B = cbind(c("a","c"),c("B","C"))
> A[B]

[1]  5 11
```

So, yes we can. Can we do the same with `data.frame`?

```
> A = data.frame(A=1:4,B=letters[11:14],C=pi*1:4)
> rownames(A) = letters[1:4]
> A

  A B         C
a 1 k  3.141593
b 2 l  6.283185
c 3 m  9.424778
d 4 n 12.566371

> B
```

```
     [,1] [,2]
[1,] "a"  "B"
[2,] "c"  "C"
```

```
> A[B]
```

```
[1] "k"         " 9.424778"
```

But, notice that the result was coerced to character. R coerced A to matrix first so that the syntax could work, but the result isn't ideal. Let's try making B a data.frame.

```
> B = data.frame(c("a","c"),c("B","C"))
> cat(try(A[B],silent=TRUE))
```

```
Error in `[.default`(A, B) : invalid subscript type 'list'
```

So we can't subset a data.frame by a data.frame in base R. What if we want row names and column names that aren't character but integer or float? What if we want more than 2 dimensions of mixed types? Enter data.table.

Furthermore, matrices, especially sparse matrices, are often stored in a 3 column tuple: (i,j,value). This can be thought of as a key-value pair where i and j form a 2-column key. If we have more than one value, perhaps of different types it might look like (i,j,val1,val2,val3,...). This looks very much like a data.frame. Hence data.table extends data.frame so that a data.frame X can be subset by a data.frame Y, leading to the X[Y] syntax.

2.15 Can base be changed to do this then, rather than a new package?

data.frame is used *everywhere* and so it is very difficult to make *any* changes to it. data.table *inherits* from data.frame. It *is* a data.frame, too. A data.table *can* be passed to any package that *only* accepts data.frame. When that package uses [.data.frame syntax on the data.table, it works. It works because [.data.table looks to see where it was called from. If it was called from such a package, [.data.table diverts to [.data.frame.

2.16 I've heard that data.table syntax is analogous to SQL.

Yes :

- i <==> where
- j <==> select
- := <==> update
- by <==> group by
- i <==> order by (in compound syntax)
- i <==> having (in compound syntax)
- nomatch=NA <==> outer join
- nomatch=0 <==> inner join
- mult="first"|"last" <==> N/A because SQL is inherently unordered
- roll=TRUE <==> N/A because SQL is inherently unordered

The general form is :

```
DT[where,select|update,group by][order by][...]  ...  [...]
```

A key advantage of column vectors in R is that they are *ordered*, unlike SQL[2]. We can use ordered functions in `data.table` queries such as `diff()` and we can use *any* R function from any package, not just the functions that are defined in SQL. A disadvantage is that R objects must fit in memory, but with several R packages such as ff, bigmemory, mmap and indexing, this is changing.

2.17 What are the smaller syntax differences between `data.frame` and `data.table`?

- `DT[3]` refers to the 3rd row, but `DF[3]` refers to the 3rd column
- `DT[3,]` == `DT[3]`, but `DF[,3]` == `DF[3]` (somewhat confusingly)
- For this reason we say the comma is *optional* in `DT`, but not optional in `DF`
- `DT[[3]]` == `DF[3]` == `DF[[3]]`
- `DT[i,]` where `i` is a single integer returns a single row, just like `DF[i,]`, but unlike a matrix single row subset which returns a vector.
- `DT[,j,with=FALSE]` where `j` is a single integer returns a one column `data.table`, unlike `DF[,j]` which returns a vector by default
- `DT[,"colA",with=FALSE][[1]]` == `DF[,"colA"]`.
- `DT[,colA]` == `DF[,"colA"]`
- `DT[,list(colA)]` == `DF[,"colA",drop=FALSE]`
- `DT[NA]` returns 1 row of `NA`, but `DF[NA]` returns a copy of `DF` containing `NA` throughout. The symbol `NA` is type logical in R and is therefore recycled by `[.data.frame`. Intention was probably `DF[NA_integer_]`. `[.data.table` does this automatically for convenience.
- `DT[c(TRUE,NA,FALSE)]` treats the `NA` as `FALSE`, but `DF[c(TRUE,NA,FALSE)]` returns `NA` rows for each `NA`
- `DT[ColA==ColB]` is simpler than `DF[!is.na(ColA) & !is.na(ColB) & ColA==ColB,]`
- `data.frame(list(1:2,"k",1:4))` creates 3 columns, `data.table` creates one `list` column.
- `check.names` is by default `TRUE` in `data.frame` but `FALSE` in `data.table`, for convenience.
- `stringsAsFactors` is by default `TRUE` in `data.frame` but `FALSE` in `data.table`, for efficiency. Since a global string cache was added to R, characters items are a pointer to the single cached string and there is no longer a performance benefit of coverting to factor.
- Atomic vectors in `list` columns are collapsed when printed using ”, ” in `data.frame`, but ”,” in `data.table` with a trailing comma after the 6th item to avoid accidental printing of large embedded objects.

In `[.data.frame` we very often set `drop=FALSE`. When we forget, bugs can arise in edge cases where single columns are selected and all of a sudden a vector is returned rather than a single column `data.frame`. In `[.data.table` we took the opportunity to make it consistent and drop `drop`.

When a `data.table` is passed to a `data.table`-unaware package, that package it not concerned with any of these differences; it just works.

[2] It may be a surprise to learn that `select top 10 * from ...` does *not* reliably return the same rows over time in SQL. You do need to include an `order by` clause, or use a clustered index to guarantee row order; i.e., SQL is inherently unordered.

2.18 I'm using j for its side effect only, but I'm still getting data returned. How do I stop that?

In this case j can be wrapped with invisible(); e.g., DT[,invisible(hist(colB)),by=colA][3].

2.19 Why does [.data.table now have a drop argument from v1.5?

So that data.table can inherit from data.frame without using .... If we used ... then invalid argument names would not be caught.

The drop argument is never used by [.data.table. It is a placeholder for non data.table aware packages when they use the [.data.frame syntax directly on a data.table.

2.20 Rolling joins are cool and very fast! Was that hard to program?

The prevailing row on or before the i row is the final row the binary search tests anyway. So roll=TRUE is essentially just a switch in the binary search C code to return that row.

2.21 Why does DT[i,col:=value] return the whole of DT? I expected either no visible value (consistent with <-), or a message or return value containing how many rows were updated. It isn't obvious that the data has indeed been updated by reference.

This has changed in v1.8.3 to meet your expectations. Please upgrade. The whole of DT is returned (now invisibly) so that compound syntax can work; e.g., DT[i,done:=TRUE][,sum(done)]. The number of rows updated is returned when verbosity is on, either on a per query basis or globally using options(datatable.verbose=TRUE).

2.22 Ok, thanks. What was so difficult about the result of DT[i,col:=value] being returned invisibly?

R internally forces visibility on for [. The value of FunTab's eval column (see src/main/names.c) for [is 0 meaning force R_Visible on (see R-Internals section 1.6). Therefore, when we tried invisible() or setting R_Visible to 0 directly ourselves, eval in src/main/eval.c would force it on again.

To solve this problem, the key was to stop trying to stop the print method running after a :=. Instead, inside := we now (from v1.8.3) set a global flag which the print method uses to know whether to actually print or not.

2.23 I've noticed that base::cbind.data.frame (and base::rbind.data.frame) appear to be changed by data.table. How is this possible? Why?

It is a temporary, last resort solution until we discover a better way to solve the problems listed below. Essentially, the issue is that data.table inherits from data.frame, *and*, base::cbind and base::rbind (uniquely) do their own S3 dispatch internally as documented by ?cbind. The change is adding one for loop to the start of each function directly in base; e.g.,

```
> base::cbind.data.frame

function (..., deparse.level = 1)
{
    if (!identical(class(..1), "data.frame"))
        for (x in list(...)) {
            if (inherits(x, "data.table"))
                return(data.table::data.table(...))
        }
```

[3] hist() returns the breakpoints in addition to plotting to the graphics device

```
    data.frame(..., check.names = FALSE)
}
<environment: namespace:base>
```

That modification is made dynamically; i.e., the base definition of `cbind.data.frame` is fetched, the `for` loop added to the beginning and then assigned back to base. This solution is intended to be robust to different definitions of `base::cbind.data.frame` in different versions of R, including unknown future changes. Again, it is a last resort until a better solution is known or made available. The competing requirements are :

- `cbind(DT,DF)` needs to work. Defining `cbind.data.table` doesn't work because `base::cbind` does its own S3 dispatch and requires that the *first* `cbind` method for each object it is passed is *identical*. This is not true in `cbind(DT,DF)` because the first method for `DT` is `cbind.data.table` but the first method for `DF` is `cbind.data.frame`. `base::cbind` then falls through to its internal `bind` code which appears to treat `DT` as a regular list and returns very odd looking and unusable `matrix` output. See FAQ 4.4. We cannot just advise users not to call `cbind(DT,DF)` because packages such as ggplot2 make such a call (test 168.5).

- This naturally leads to trying to mask `cbind.data.frame` instead. Since a `data.table` is a `data.frame`, `cbind` would find the same method for both `DT` and `DF`. However, this doesn't work either because `base::cbind` appears to find methods in `base` first; i.e., `base::cbind.data.frame` isn't maskable. This is reproducible as follows :

```
> foo = data.frame(a=1:3)
> cbind.data.frame = function(...)cat("Not printed\n")
> cbind(foo)

  a
1 1
2 2
3 3
```

- Finally, we tried masking `cbind` itself (v1.6.5 and v1.6.6). This allowed `cbind(DT,DF)` to work, but introduced compatibility issues with package IRanges, since IRanges also masks `cbind`. It worked if IRanges was lower on the search() path than `data.table`, but if IRanges was higher then `data.table`'s `cbind` would never be called and the strange looking matrix output occurs again (FAQ 4.4).

If you know of a better solution, that still solves all the issues above, then please let us know and we'll gladly change it.

2.24 I've read about method dispatch (e.g. `merge` may or may not dispatch to `merge.data.table`) but *how* does R know how to dispatch? Are dots significant or special? How on earth does R know which function to dispatch and when?

This comes up quite a lot, but it's really earth shatteringly simple. A function such as `merge` is *generic* if it consists of a call to `UseMethod`. When you see people talking about whether or not functions are *generic* functions they are merely typing the function, without `()` afterwards, looking at the program code inside it and if they see a call to `UseMethod` then it is *generic*. What does `UseMethod` do? It literally slaps the function name together with the class of the first argument, separated by period (`.`) and then calls that function, passing along the same arguments. It's that simple. For example, `merge(X,Y)` contains a `UseMethod` call which means it then *dispatches* (i.e. calls) `paste("merge",class(X),sep=".")`. Functions with dots in may or may not be methods. The dot is irrelevant really. Other than dot being the separator that `UseMethod` uses. Knowing this background should now highlight why, for example, it is obvious to R folk that `as.data.table.data.frame` is the `data.frame` method for the `as.data.table` generic function. Further, it may help to elucidate that, yes you are correct, it is not obvious from its name alone that `ls.fit` is not the fit method of the `ls` generic function. You only know that by typing `ls` (not `ls()`) and observing it isn't a single call to `UseMethod`.

You might now ask: where is this documented in R? Answer: it's quite clear, but, you need to first know to look in `?UseMethod` and *that* help file contains :

"When a function calling UseMethod('fun') is applied to an object with class attribute c('first', 'second'), the system searches for a function called fun.first and, if it finds it, applies it to the object. If no such function is found a function called fun.second is tried. If no class name produces a suitable function, the function fun.default is used, if it exists, or an error results."

Happily, an internet search for "How does R method dispatch work" (at the time of writing) returns the `?UseMethod` help page as the top link. Admittedly, other links rapidly descend into the intracies of S3 vs S4, internal generics and so on.

However, features like basic S3 dispatch (pasting the function name together with the class name) is why some R folk love R. It's so simple. No complicated registration or signature is required. There isn't much needed to learn. To create the `merge` method for `data.table` all that was required, literally, was to merely create a function called `merge.data.table`.

3 Questions relating to compute time

3.1 I have 20 columns and a large number of rows. Why is an expression of one column so quick?

Several reasons:

- Only that column is grouped, the other 19 are ignored because `data.table` inspects the `j` expression and realises it doesn't use the other columns.
- One memory allocation is made for the largest group only, then that memory is re-used for the other groups. There is very little garbage to collect.
- R is an in-memory column store; i.e., the columns are contiguous in RAM. Page fetches from RAM into L2 cache are minimised.

3.2 I don't have a key on a large table, but grouping is still really quick. Why is that?

`data.table` uses radix sorting. This is significantly faster than other sort algorithms. See our presentations on our homepage for more information.

This is also one reason why `setkey()` is quick.

When no key is set, or we group in a different order from that of the key, we call it an *ad hoc by*.

3.3 Why is grouping by columns in the key faster than an ad hoc by?

Because each group is contiguous in RAM, thereby minimising page fetches and memory can be copied in bulk (memcpy in C) rather than looping in C.

4 Error messages

4.1 `Could not find function "DT"`

See FAQ 2.6 and FAQ 2.7.

4.2 `unused argument(s) (MySum = sum(v))`

This error is generated by `DT[,MySum=sum(v)]`. `DT[,.(MySum=sum(v))]` was intended, or `DT[,j=.(MySum=sum(v))]`.

4.3 'translateCharUTF8' must be called on a CHARSXP

This error (and similar; e.g., `'getCharCE' must be called on a CHARSXP`) may be nothing do with character data or locale. Instead, this can be a symptom of an earlier memory corruption. To date these have been reproducible and fixed (quickly). Please report it to datatable-help.

4.4 cbind(DT,DF) returns a strange format e.g. 'Integer,5'

This occurs prior to v1.6.5, for `rbind(DT,DF)` too. Please upgrade to v1.6.7 or later.

4.5 cannot change value of locked binding for '.SD'

`.SD` is locked by design. See `?data.table`. If you'd like to manipulate `.SD` before using it, or returning it, and don't wish to modify `DT` using `:=`, then take a copy first (see `?copy`); e.g.,

```
> DT = data.table(a=rep(1:3,1:3),b=1:6,c=7:12)
> DT

   a b  c
1: 1 1  7
2: 2 2  8
3: 2 3  9
4: 3 4 10
5: 3 5 11
6: 3 6 12

> DT[,{ mySD = copy(.SD)
+       mySD[1,b:=99L]
+       mySD },
+     by=a]

   a  b  c
1: 1 99  7
2: 2 99  8
3: 2  3  9
4: 3 99 10
5: 3  5 11
6: 3  6 12
```

4.6 cannot change value of locked binding for '.N'

Please upgrade to v1.8.1 or later. From this version, if `.N` is returned by `j` it is renamed to `N` to avoid any abiguity in any subsequent grouping between the `.N` special variable and a column called `".N"`. The old behaviour can be reproduced by forcing `.N` to be called `.N`, like this :

```
> DT = data.table(a=c(1,1,2,2,2),b=c(1,2,2,2,1))
> DT

   a b
1: 1 1
2: 1 2
3: 2 2
4: 2 2
5: 2 1

> DT[,list(.N=.N),list(a,b)]   # show intermediate result for exposition
```

```
   a b .N
1: 1 1  1
2: 1 2  1
3: 2 2  2
4: 2 1  1
```

```
> cat(try(
+     DT[,list(.N=.N),by=list(a,b)][,unique(.N),by=a]   # compound query more typical
+ ,silent=TRUE))
```

```
Error in `[.data.table`(DT[, list(.N = .N), by = list(a, b)], , unique(.N),  :
  The column '.N' can't be grouped because it conflicts with the special .N variable. Try setnames
```

If you are already running v1.8.1 or later then the error message is now more helpful than the `cannot change value of locked binding` error. As you can see above, since this vignette was produced using v1.8.1 or later. The more natural syntax now works :

```
> if (packageVersion("data.table") >= "1.8.1") {
+     DT[,.N,by=list(a,b)][,unique(N),by=a]
+ }
```

```
   a V1
1: 1  1
2: 2  2
3: 2  1
```

```
> if (packageVersion("data.table") >= "1.9.3") {
+     DT[,.N,by=.(a,b)][,unique(N),by=a]   # same
+ }
```

```
   a V1
1: 1  1
2: 2  2
3: 2  1
```

5 Warning messages

5.1 The following object(s) are masked from 'package:base': cbind, rbind

This warning was present in v1.6.5 and v.1.6.6 only, when loading the package. The motivation was to allow `cbind(DT,DF)` to work, but as it transpired, broke (full) compatibility with package IRanges. Please upgrade to v1.6.7 or later.

5.2 Coerced numeric RHS to integer to match the column's type

Hopefully, this is self explanatory. The full message is :

```
   Coerced numeric RHS to integer to match the column's type; may have truncated
precision.  Either change the column to numeric first by creating a new numeric
vector length 5 (nrows of entire table) yourself and assigning that (i.e.
'replace' column), or coerce RHS to integer yourself (e.g.  1L or as.integer)
to make your intent clear (and for speed).  Or, set the column type correctly
up front when you create the table and stick to it, please.
```

To generate it, try :

```
> DT = data.table(a=1:5,b=1:5)
> suppressWarnings(
+ DT[2,b:=6]        # works (slower) with warning
+ )
```

```
   a b
1: 1 1
2: 2 6
3: 3 3
4: 4 4
5: 5 5

> class(6)           # numeric not integer

[1] "numeric"

> DT[2,b:=7L]        # works (faster) without warning

   a b
1: 1 1
2: 2 7
3: 3 3
4: 4 4
5: 5 5

> class(7L)          # L makes it an integer

[1] "integer"

> DT[,b:=rnorm(5)]   # 'replace' integer column with a numeric column

   a          b
1: 1 -0.6789287
2: 2  0.7617814
3: 3  0.1430932
4: 4 -0.1027536
5: 5 -2.5685330
```

6 General questions about the package

6.1 v1.3 appears to be missing from the CRAN archive?

That is correct. v1.3 was available on R-Forge only. There were several large changes internally and these took some time to test in development.

6.2 Is `data.table` compatible with S-plus?

Not currently.

- A few core parts of the package are written in C and use internal R functions and R structures.
- The package uses lexical scoping which is one of the differences between R and S-plus explained by R FAQ 3.3.1.

6.3 Is it available for Linux, Mac and Windows?

Yes, for both 32-bit and 64-bit on all platforms. Thanks to CRAN. There are no special or OS-specific libraries used.

6.4 I think it's great. What can I do?

Please file suggestions, bug reports and enhancement requests on GitHub. This helps make the package better.

Please do vote for the package on Crantastic. This helps encourage the developers and helps other R users find the package. If you have time to write a comment too, that can help others in the community. Just simply clicking that you use the package, though, is much appreciated.

You can submit pull requests to change the code and/or documentation yourself.

6.5 I think it's not great. How do I warn others about my experience?

Please put your vote and comments on Crantastic. Please make it constructive so we have a chance to improve.

6.6 I have a question. I know the r-help posting guide tells me to contact the maintainer (not r-help), but is there a larger group of people I can ask?

Yes, there are two options. You can post to datatable-help. It's like r-help, but just for this package. Or the `data.table` tag on Stack Overflow. Feel free to answer questions in those places, too.

6.7 Where are the datatable-help archives?

The homepage contains links to the archives in several formats.

6.8 I'd prefer not to contact datatable-help, can I mail just one or two people privately?

Sure. You're more likely to get a faster answer from datatable-help or Stack Overflow, though. Asking publicly in those places helps build the knowledge base.

6.9 I have created a package that depends on `data.table`. How do I ensure my package is `data.table`-aware so that inheritance from `data.frame` works?

Either i) include `data.table` in the `Depends:` field of your DESCRIPTION file, or ii) include `data.table` in the `Imports:` field of your DESCRIPTION file AND `import(data.table)` in your NAMESPACE file.

6.10 Why is this FAQ in pdf format? Can it moved to HTML?

Yes we'd like to move it to a HTML vignette. Just haven't got to that yet. The benefits of vignettes (rather than a wiki) including the following:

- We include R code in the vignettes. This code is *actually run* when the file is created, not copy and pasted.
- This document is *reproducible*. Grab the .Rnw and you can run it yourself.
- CRAN checks the package (including running vignettes) every night on Linux, Mac and Windows, both 32bit and 64bit. Results are posted to `http://cran.r-project.org/web/checks/check_results_data.table.html`. Included there are results from r-devel; i.e., not yet released R. That serves as a very useful early warning system for any potential future issues as R itself develops.
- This file is bound into each version of the package. The package is not accepted on CRAN unless this file passes checks. Each version of the package will have its own FAQ file which will be relevant for that version. Contrast this to a single website, which can be ambiguous if the answer depends on the version.
- You can open it offline at your R prompt using `vignette()`.
- You can extract the code from the document and play with it using `edit(vignette("datatable-faq"))`.