

data.table vs dplyr: can one do something well the other can't or does poorly?

Overview

I'm relatively familiar with `data.table`, not so much with `dplyr`. I've read through some [dplyr vignettes](#) and examples that have popped up on SO, and so far my conclusions are that:

1. `data.table` and `dplyr` are comparable in speed, except when there are many (i.e. >10-100K) groups, and in some other circumstances (see benchmarks below)
2. `dplyr` has more accessible syntax
3. `dplyr` abstracts (or will) potential DB interactions
4. There are some minor functionality differences (see "Examples/Usage" below)

In my mind 2. doesn't bear much weight because I am fairly familiar with it `data.table`, though I understand that for users new to both it will be a big factor. I would like to avoid an argument about which is more intuitive, as that is irrelevant for my specific question asked from the perspective of someone already familiar with `data.table`. I also would like to avoid a discussion about how "more intuitive" leads to faster analysis (certainly true, but again, not what I'm most interested about here).

Question

What I want to know is:

1. Are there analytical tasks that are a lot easier to code with one or the other package for people familiar with the packages (i.e. some combination of keystrokes required vs. required level of esotericism, where less of each is a good thing).
2. Are there analytical tasks that are performed substantially (i.e. more than 2x) more efficiently in one package vs. another.

One [recent SO question](#) got me thinking about this a bit more, because up until that point I didn't think `dplyr` would offer much beyond what I can already do in `data.table`. Here is the `dplyr` solution (data at end of Q):

```
dat %>%
  group_by(name, job) %>%
  filter(job != "Boss" | year == min(year)) %>%
  mutate(cumu_job2 = cumsum(job2))
```

Which was much better than my hack attempt at a `data.table` solution. That said, good `data.table` solutions are also pretty good (thanks Jean-Robert, Arun, and note here I favored single statement over the strictly most optimal solution):

```
setDT(dat)[,
  .SD[job != "Boss" | year == min(year)][, cumjob := cumsum(job2)],
  by=list(id, job)
]
```

The syntax for the latter may seem very esoteric, but it actually is pretty straightforward if you're used to `data.table` (i.e. doesn't use some of the more esoteric tricks).

Ideally what I'd like to see is some good examples where the `dplyr` or `data.table` way is substantially more concise or performs substantially better.

Examples

Usage

- `dplyr` does not allow grouped operations that return arbitrary number of rows (from [eddi's question](#), note: this looks like it will be implemented in [dplyr 0.5](#), also, @beginneR shows a potential work-around using `do` in the answer to @eddi's question).
- `data.table` supports [rolling joins](#) (thanks @dholstius) as well as [overlap joins](#)
- `data.table` internally optimises expressions of the form `DT[col == value]` or `DT[col %in% values]` for *speed* through *automatic indexing* which uses *binary search* while using the same base R syntax. [See here](#) for some more details and a tiny benchmark.
- `dplyr` offers standard evaluation versions of functions (e.g. `regroup`, `summarize_each`) that can simplify the programmatic use of `dplyr` (note programmatic use of `data.table` is definitely possible, just requires some careful thought, substitution/quoting, etc., at least to my knowledge)

Benchmarks

- I ran [my own benchmarks](#) and found both packages to be comparable in "split apply combine" style analysis, except when there are very large numbers of groups (>100K) at which point `data.table` becomes substantially faster.
- @Arun ran some [benchmarks on joins](#), showing that `data.table` scales better than `dplyr` as the number of groups increase (updated with recent enhancements in both packages and recent version of R). Also, a benchmark when trying to get [unique values](#) has `data.table` ~6x faster.
- (Unverified) has `data.table` 75% faster on larger versions of a group/apply/sort while `dplyr` was 40% faster on the smaller ones

([another SO question from comments](#), thanks danas).

- Matt, the main author of `data.table`, has [benchmarked grouping operations on `data.table`, `dplyr` and python `pandas` on up to 2 billion rows \(~100GB in RAM\)](#).
- An [older benchmark on 80K groups](#) has `data.table` ~8x faster

Data


This is for the first example I showed in the question section.

```
dat <- structure(list(id = c(1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 2L, 2L,
2L, 2L, 2L, 2L, 2L, 2L), name = c("Jane", "Jane", "Jane", "Jane",
"Jane", "Jane", "Jane", "Jane", "Bob", "Bob", "Bob", "Bob", "Bob",
"Bob", "Bob", "Bob"), year = c(1980L, 1981L, 1982L, 1983L, 1984L,
1985L, 1986L, 1987L, 1985L, 1986L, 1987L, 1988L, 1989L, 1990L,
1991L, 1992L), job = c("Manager", "Manager", "Manager", "Manager",
"Manager", "Manager", "Boss", "Boss", "Manager", "Manager", "Manager",
"Boss", "Boss", "Boss", "Boss"), job2 = c(1L, 1L, 1L,
1L, 1L, 1L, 0L, 0L, 1L, 1L, 1L, 0L, 0L, 0L, 0L, 0L)), .Names = c("id",
"name", "year", "job", "job2"), class = "data.frame", row.names = c(NA,
-16L))
```

r `data.table` `dplyr`

edited Jan 20 at 16:14

asked Jan 29 '14 at 15:21

 **BrodieG**
25.5k ● 4 ● 22 ● 49

- 5 The solution that's similar in reading to the `dplyr` one is: `as.data.table(dat)[, .SD[job != "Boss" | year == min(year)][, cumjob := cumsum(job2)], by = list(name, job)]` — [eddi](#) Jan 29 '14 at 16:12
- 4 For #1 both `dplyr` and `data.table` teams are working on benchmarks, so an answer will be there at some point. #2 (syntax) imO is strictly false, but that clearly ventures into opinion territory, so I'm voting to close as well. — [eddi](#) Jan 29 '14 at 16:16
- 7 well, again imO, the set of problems that are more cleanly expressed in `(d)plyr` has measure 0 — [eddi](#) Jan 29 '14 at 16:26
- 9 @BrodieG the one thing that really bugs me about both `dplyr` and `plyr` with regards to syntax and is basically the main reason why I dislike their syntax, is that I have to learn way too many (read more than 1) extra functions (with names that *still* don't make sense for me), remember what they do, what arguments they take, etc. That has always been a huge turn off for me from plyr-philosophy. — [eddi](#) Jan 29 '14 at 21:48
- 18 @eddi [tongue-in-cheek] the one thing that really bugs me about `data.table` syntax is that I have to learn how way too many function arguments interact, and what cryptic shortcuts mean (e.g. `.SD`). [seriously] I think these are legitimate design differences that will appeal to different people — [hadley](#) Jan 29 '14 at 22:53

3 Answers

We need to cover at least these aspects to provide a comprehensive answer/comparison (in no particular order of importance): `Speed`, `Memory usage`, `Syntax` and `Features`.

My intent is to cover each one of these as clearly as possible from `data.table` perspective.

Note: unless explicitly mentioned otherwise, by referring to `dplyr`, we refer to `dplyr`'s `data.frame` interface whose internals are in C++ using `Rcpp`.

1. Speed

Quite a few benchmarks (though mostly on grouping operations) have been added to the question already showing `data.table` gets *faster* than `dplyr` as the number of groups and/or rows to group by increase, including [benchmarks by Matt](#) on grouping from *10 million to 2 billion rows* (100GB in RAM) on *100 - 10 million groups* and varying grouping columns, which also compares `pandas`.

On benchmarks, it would be great to cover these remaining aspects as well:

- Grouping operations involving a *subset of rows* - i.e., `DT[x > val, sum(y), by=z]` type operations.
- Benchmark other operations such as *update* and *joins*.

- Also benchmark *memory footprint* for each operation in addition to runtime.

2. Memory usage

The data.table syntax is consistent in its form - `DT[i, j, by]`. To keep `i`, `j` and `by` together is by design. By keeping related operations together, it allows to *easily optimise* operations for *speed* and more importantly *memory usage*, and also provide some *powerful features*, all while maintaining the consistency in syntax.

In this section, let's consider *memory usage*:

1. Operations involving `filter()` or `slice()` in dplyr can be memory inefficient (on both data.frames and data.tables). [See this post](#).

Note that [Hadley's comment](#) talks about *speed* (that dplyr is plentiful fast for him), whereas the major concern here is *memory*.

2. data.table interface at the moment allows one to modify/update columns *by reference*.

```
# sub-assign by reference
DT[x >= 1L, y := NA] ## updates 'y' in-place
```

But dplyr *will never* update by reference. The dplyr equivalent would be:

```
DF %>% mutate(y = replace(y, which(x >= 1L), NA)) ## copies the entire 'y' column
```

A concern for this is *referential transparency*. Updating a data.table object by reference, especially within a function may not be always desirable. But this is an incredibly useful feature: see [this](#) and [this](#) posts for interesting cases. And we want to keep it.

Therefore we are working towards exporting `shallow()` function in data.table that will provide the user with *both possibilities*. For example, if it is desirable to not modify the input data.table within a function, one can then do:

```
foo <- function(DT) {
  DT = shallow(DT)      ## shallow copy DT
  DT[, newcol := 1L]    ## does not affect the original DT
  DT[x > 2L, newcol := 2L] ## no need to copy (internally), as this column exists only
  ## in shallow copied DT
  DT[x > 2L, x := 3L]    ## have to copy (like base R / dplyr does always); otherwise
  ## original DT will   ## also get modified.
}
```

By not using `shallow()`, the old functionality is retained:

```
bar <- function(DT) {
  DT[, newcol := 1L]    ## old behaviour, original DT gets updated by reference
  DT[x > 2L, x := 3L]    ## old behaviour, update column x in original DT.
}
```

By creating a *shallow copy* using `shallow()`, we understand that you don't want to modify the original object. We take care of everything internally to ensure that while also ensuring to copy columns you modify *only when it is absolutely necessary*. When implemented, this should settle the *referential transparency* issue altogether while providing the user with both possibilities.

Also, once `shallow()` is exported dplyr's data.table interface should avoid almost all copies. So those who prefer dplyr's syntax can use it with data.tables.

But it will still lack many features that data.table provides, including (sub)-assignment by reference.

3. Aggregate while joining:

Suppose you have two data.tables as follows:

```
DT1 = data.table(x=c(1,1,1,1,2,2,2,2), y=c("a", "a", "b", "b"), z=1:8, key=c("x", "y"))
#   x y z
# 1: 1 a 1
# 2: 1 a 2
# 3: 1 b 3
# 4: 1 b 4
# 5: 2 a 5
# 6: 2 a 6
# 7: 2 b 7
# 8: 2 b 8
DT2 = data.table(x=1:2, y=c("a", "b"), mul=4:3, key=c("x", "y"))
```

```
#   x y mul
# 1: 1 a   4
# 2: 2 b   3
```

And you would like to get `sum(z) * mul` for each row in `DT2` while joining by columns `x,y`. We can either:

- 1) aggregate `DT1` to get `sum(z)`, 2) perform a join and 3) multiply (or)

```
# data.table way
DT1[, .(z=sum(z)), keyby=.(x,y)][DT2[, z := z*mul][]]

# dplyr equivalent
DF1 %>% group_by(x,y) %>% summarise(z=sum(z)) %>%
  right_join(DF2) %>% mutate(z=z*mul)
```

- 2) do it all in one go (using `by=.EACHI` feature):

```
DT1[DT2, list(z=sum(z) * mul), by=.EACHI]
```

What is the advantage?

- We don't have to allocate memory for the intermediate result.
- We don't have to group/hash twice (one for aggregation and other for joining).
- And more importantly, the operation what we wanted to perform is clear by looking at `j` in (2).

Check [this post](#) for a detailed explanation of `by=.EACHI`. No intermediate results are materialised, and the join+aggregate is performed all in one go.

Have a look at [this](#), [this](#) and [this](#) posts for real usage scenarios.

In `dplyr` you would have to [join and aggregate](#) or [aggregate first and then join](#), neither of which are as efficient, in terms of memory (which in turn translates to speed).

4. Update and joins:

Consider the data.table code shown below:

```
DT1[DT2, col := i.mul]
```

adds/updates `DT1`'s column `col` with `mul` from `DT2` on those rows where `DT2`'s key column matches `DT1`. I don't think there is an exact equivalent of this operation in `dplyr`, i.e., without avoiding a `*_join` operation, which would have to copy the entire `DT1` just to add a new column to it, which is unnecessary.

Check [this post](#) for a real usage scenario.

To summarise, it is important to realise that every bit of optimisation matters. As [Grace Hopper](#) would say, **Mind your nanoseconds!**

3. Syntax

Let's now look at *syntax*. Hadley commented [here](#):

Data tables are extremely fast but I think their concision makes it *harder to learn* and *code that uses it is harder to read after you have written it*...

I find this remark pointless because it is very subjective. What we can perhaps try is to contrast *consistency in syntax*. We will compare data.table and dplyr syntax side-by-side.

We will work with the dummy data shown below:

```
DT = data.table(x=1:10, y=11:20, z=rep(1:2, each=5))
DF = as.data.frame(DT)
```

1. Basic aggregation/update operations.

| ## data.table Syntax | ## dplyr syntax |
|---------------------------------|--|
| # case (a) | |
| DT[, sum(y), by=z] | DF %>% group_by(z) %>% summarise(sum(y)) |
| DT[, y := cumsum(y), by=z] | DF %>% group_by(z) %>% mutate(y = cumsum(y)) |
| # case (b) | |
| DT[x > 2, sum(y), by=z] | DF %>% filter(x>2) %>% group_by(z) %>% |
| summarise(sum(y)) | |
| DT[x > 2, y := cumsum(y), by=z] | DF %>% group_by(z) %>% mutate(y = replace(y, |

```
which(x>2), cumsum(y)))
# case (c)
DT[, if(any(x > 5L)) y[1L]-y[2L]
    else y[2L], by=z]
DT[, if(any(x > 5L))
    y[1L]-y[2L], by=z]
```

```
DF %>% group_by(z) %>% summarise(
  if (any(x > 5L)) y[1L]-y[2L] else y[2L])
DF %>% group_by(z) %>% filter(any(x > 5L))
  %>% summarise(y[1L]-y[2L])
```

- data.table syntax is compact and dplyr's quite verbose. Things are more or less equivalent in case (a).
- In case (b), we had to use `filter()` in dplyr while *summarising*. But while *updating*, we had to move the logic inside `mutate()`. In data.table however, we express both operations with the same logic - operate on rows where `x > 2`, but in first case, get `sum(y)`, whereas in the second case update those rows for `y` with its cumulative sum.

This is what we mean when we say the `DT[i, j, by]` form is *consistent*.

- Similarly in case (c), when we have `if-else` condition, we are able to express the logic "as-is" in both data.table and dplyr. However, if we would like to return just those rows where the `if` condition satisfies and skip otherwise, we cannot use `summarise()` directly (AFAICT). We have to `filter()` first and then summarise because `summarise()` always expects a *single value*.

While it returns the same result, using `filter()` here makes the actual operation less obvious.

It might very well be possible to use `filter()` in the first case as well (does not seem obvious to me), but my point is that we should not have to.

2. Aggregation / update on multiple columns

| ## data.table Syntax | ## dplyr syntax |
|---|---|
| # case (a) | |
| DT[, lapply(.SD, sum), by=z] | DF %>% group_by(z) %>% summarise_each(funs(sum)) |
| DT[, (cols) := lapply(.SD, sum), by=z] | DF %>% group_by(z) %>% mutate_each(funs(sum)) |
| # case (b) | |
| DT[, c(lapply(.SD, sum), lapply(.SD, mean)), by=z] | DF %>% group_by(z) %>% summarise_each(funs(sum, mean)) |
| # case (c) | |
| DT[, c(.N, lapply(.SD, sum)), by=z] | DF %>% group_by(z) %>% summarise_each(funs(n(), mean)) |

- In case (a), the codes are more or less equivalent. data.table uses familiar base function `lapply()`, whereas dplyr introduces `*_each()` along with a bunch of functions to `funs()`.
- data.table's `:=` requires column names to be provided, whereas dplyr generates it automatically.
- In case (b), dplyr's syntax is relatively straightforward. Improving aggregations/updates on multiple functions is on data.table's list.
- In case (c) though, dplyr would return `n()` as many times as many columns, instead of just once. In data.table, all we need to do is to return a list in `j`. Each element of the list will become a column in the result. So, we can use, once again, the familiar base function `c()` to concatenate `.N` to a `list` which returns a `list`.

Note: To remind again, in data.table, all we need to do is return a list in `j`. Each element of the list will become a column in result. You can use `c()`, `as.list()`, `lapply()`, `list()` etc... base functions to accomplish this, without having to learn any new functions.

You will need to learn just the special variables - `.N` and `.SD` at least. The equivalent in dplyr are `n()` and `.`.

3. Joins

dplyr provides separate functions for each type of join where as data.table allows joins using the same syntax `DT[i, j, by]` (and with reason). It also provides an equivalent

`merge.data.table()` function as an alternative.

| ## data.table Syntax | ## dplyr syntax |
|--------------------------------|---|
| setkey(DT1, x, y) | |
| # 1. normal join | |
| DT1[DT2] | left_join(DT2, DT1) |
| # 2. select columns while join | |
| DT1[DT2, .(z, i.mul)] | left_join(select(DT2, x,y,mul), select(DT1, |
| x,y,z)) | |

```
# 3. aggregate while join
DT1[DT2, .(sum(z)*i.mul), by=.EACHI]      DF1 %>% group_by(x, y) %>% summarise(z=sum(z))
%>%                                         inner_join(DF2) %>% mutate(z = z*mul) %>%

select(-mul)
# 4. update while join
DT1[DT2, z := cumsum(z)*i.mul, by=.EACHI]  join and group by + mutate
# 5. rolling join
DT1[DT2, roll = -Inf]                      ??
# 6. other arguments to control output
DT1[DT2, mult = "first"]                   ??
```

- Some might find a separate function for each joins much nicer (left, right, inner, anti, semi etc..), whereas as others might like data.table's `DT[i, j, by]`, or `merge()` which is similar to base R.
- However dplyr joins do just that. Nothing more. Nothing less.
- data.tables can select columns while joining (2), and in dplyr you will need to `select()` first on both data.frames before to join as shown above. Otherwise you would materialise the join with unnecessary columns only to remove them later and that is inefficient.
- data.tables can *aggregate while joining* (3) and also *update while joining* (4), using `by=.EACHI` feature. Why materialise the entire join result to add/update just a few columns?
- data.table is capable of *rolling joins* (5) - roll [forward](#), [LOCF](#), [roll backward](#), [NOCB](#), [nearest](#).
- data.table also has `mult=` argument which selects *first*, *last* or *all* matches (6).
- data.table has `allow.cartesian=TRUE` argument to protect from accidental invalid joins.

Once again, the syntax is consistent with `DT[i, j, by]` with additional arguments allowing for controlling the output further.

4. `do()` ...

dplyr's `summarise` is specially designed for functions that return a single value. If your function returns multiple/unequal values, you will have to resort to `do()`. You have to know beforehand about all your functions return value.

```
## data.table Syntax          ## dplyr syntax
DT[, list(x[1], y[1]), by=z]   DF %>% group_by(z) %>% summarise(x[1], y[1])
DT[, list(x[1:2], y[1]), by=z] DF %>% group_by(z) %>% do(data.frame(.$x[1:2],
.$y[1]))

DT[, quantile(x, 0.25), by=z]   DF %>% group_by(z) %>% summarise(quantile(x,
0.25))
DT[, quantile(x, c(0.25, 0.75)), by=z] DF %>% group_by(z) %>% do(data.frame(quantile(.$x,
c(0.25, 0.75))))

DT[, as.list(summary(x)), by=z]  DF %>% group_by(z) %>%
do(data.frame(as.list(summary(.$x))))
```

- `.SD` 's equivalent is `.`
- In data.table, you can throw pretty much anything in `j` - the only thing to remember is for it to return a list so that each element of the list gets converted to a column.
- In dplyr, cannot do that. Have to resort to `do()` depending on how sure you are as to whether your function would always return a single value. And it is quite slow.

Once again, data.table's syntax is consistent with `DT[i, j, by]`. We can just keep throwing expressions in `j` without having to worry about these things.

Have a look at [this SO question](#) and [this one](#). I wonder if it would be possible to express the answer as straightforward using dplyr's syntax...

To summarise, I have particularly highlighted *several* instances where dplyr's syntax is either inefficient, limited or fails to make operations straightforward. This is particularly because data.table gets quite a bit of backlash about "harder to read/learn" syntax (like the one pasted/linked above). Most posts that cover dplyr talk about most straightforward operations. And that is great. But it is important to realise its syntax and feature limitations as well, and I am yet to see a post on it.

data.table has its quirks as well (some of which I have pointed out that we are attempting to fix). We are also attempting to improve data.table's joins as I have highlighted [here](#).

But one should also consider the number of features that dplyr lacks in comparison to data.table.

4. Features

I have pointed out most of the features [here](#) and also in this post. In addition:

- `fread` - fast file reader has been available for a long time now.
- Ad-hoc grouping. dplyr automatically sorts the results by grouping variables during `summarise()`, which may not be always desirable.
- [Overlapping range joins](#) was implemented in data.table recently. Check [this post](#) for an overview with benchmarks.
- [Automatic indexing](#) - another incredible feature to optimise base R syntax as is, internally.
- Numerous advantages in data.table joins (for speed / memory efficiency and syntax) mentioned above.
- `setorder()` function in data.table that allows really fast reordering of data.tables by reference.
- dplyr provides [interface to databases](#) using the same syntax, which data.table does not at the moment.
- dplyr provides `setdiff()`, `intersect()` and `union()` functions which data.table does not (yet).

Finally:

- On databases - there is no reason why data.table cannot provide similar interface, but this is not a priority now. It might get bumped up if users would very much like that feature.. not sure.
- On parallelism - Everything is difficult, until someone goes ahead and does it. Of course it will take effort (being thread safe). But radix ordering (on which data.table is built on) can be parallelised. There is no reason not to.

edited Jan 5 at 19:51

community wiki
18 revs, 3 users 99%
Arun

... So what you're doing (somewhat implicitly) is a subset of rows and that's what `filter` is made for, in dplyr. – [docendo discimus](#) Dec 31 '14 at 14:09

@docendodiscimus, sorry but what's the idiomatic way of doing `DT[, if(any(x>1)) y[1], by=z]` again please? I'm confused now. Should or shouldn't I use filter here? – [Arun](#) Dec 31 '14 at 14:21

3 @bluefeet: I don't think you did the rest of us any great service by moving that discussion to chat. I was under the impression that Arun was one of the developers and this might have resulted in useful insights. – [BondedDust](#) Jan 5 at 21:20

1 I think that every where where you are using assignment by reference (`:=`), dplyr equivalent should be also using `<-` as in `DF <- DF %>% mutate...` instead of just `DF %>% mutate...` – [David Arenburg](#) Jan 7 at 15:21

1 Regarding the syntax. I believe the `dplyr` can be easier for users who used to `plyr` syntax, but `data.table` can be easier for users who used to query languages syntax like `SQL`, and the relational algebra behind it, which is all about the tabular data transformation. @Arun you should note that **set operators** are very easy do-able by wrapping `data.table` function and of course brings significant speedup. – [jangorecki](#) Jan 14 at 19:33

Here's my attempt at a comprehensive answer from the dplyr perspective, following the broad outline of Arun's answer (but somewhat rearranged based on differing priorities).

Syntax

There is some subjectivity to syntax, but I stand by my statement that the concision of data.table makes it harder to learn and harder to read. This is partly because dplyr is solving a much easier problem!

One really important thing that dplyr does for you is that it *constrains* your options. I claim that most single table problems can be solved with just five key verbs `filter`, `select`, `mutate`, `arrange` and `summarise`, along with "by group" adverb. That constraint is big help when you're learning data manipulation, because it helps order your thinking about the problem. In dplyr, each of these verbs is mapped to a single function. Each function does one job, and is easy to understand in isolation.

You create complexity by piping these simple operations together with `%>%`. Here's an example from one of the posts Arun [linked to](#):

```
diamonds %>%
  filter(cut != "Fair") %>%
  group_by(cut) %>%
  summarize(
    AvgPrice = mean(price),
    MedianPrice = as.numeric(median(price)),
    Count = n()
  ) %>%
  arrange(desc(Count))
```

Even if you've never seen dplyr before (or even R!), you can still get the gist of what's happening because the functions are all English verbs. The disadvantage of English verbs is that they more type than `[]`, but I think that can be largely mitigated by better autocomplete.

Here's the equivalent `data.table` code:

```
diamondsDT <- data.table(diamonds)
diamondsDT[
  cut != "Fair",
  .(AvgPrice = mean(price),
    MedianPrice = as.numeric(median(price)),
    Count = .N
  ),
  by = cut
][
  order(-Count)
]
```

It's harder to follow this code unless you're already familiar with `data.table`. (I also couldn't figure out how to indent the repeated `[]` in a way that looks good to my eye). Personally, when I look at code I wrote 6 months ago, it's like looking at a code written by a stranger, so I've come to prefer straightforward, if verbose, code.

Two other minor factors that I think slightly decrease readability:

- Since almost every data table operation uses `[]` you need additional context to figure out what's happening. For example, is `x[y]` joining two data tables or extracting columns from a data frame? This is only a small issue, because in well-written code the variable names should suggest what's happening.
- I like that `group_by()` is a separate operation in dplyr. It fundamentally changes the computation so I think should be obvious when skimming the code, and it's easier to spot `group_by()` than the `by` argument to `[.data.table]`.

I also like that the [the pipe](#) isn't just limited to just one package. You can start by tidying your data with [tidyr](#), and finish up with a plot in [ggvis](#). And you're not limited to the packages that I write - anyone can write a function that forms a seamless part of a data manipulation pipe. In fact, I rather prefer the previous `data.table` code rewritten with `%>%`:

```
diamonds %>%
  data.table() %>%
  .[cut != "Fair",
  .(AvgPrice = mean(price),
    MedianPrice = as.numeric(median(price)),
    Count = .N
  ),
  by = cut
  ] %>%
  .[order(-Count)]
```

And the idea of piping with `%>%` is not limited to just data frames and is easily generalised to other contexts: [interactive web graphics](#), [web scraping](#), [gists](#), [run-time contracts](#), ...)

Memory and performance

I've lumped these together, because, to me, they're not that important. Most R users work with well under 1 million rows of data, and dplyr is sufficiently fast enough for that size of data that you're not aware of processing time. We optimise dplyr for expressiveness on medium data; feel free to use `data.table` for raw speed on bigger data.

The flexibility of dplyr also means that you can easily tweak performance characteristics using the same syntax. If the performance of dplyr with the data frame backend is not good enough for you, you can use the data.table backend (albeit a somewhat restricted set of functionality). If the data you're working with doesn't fit in memory, then you can use a database backend.

All that said, dplyr performance will get better in the long-term. We'll definitely implement some of the great ideas of data.table like radix ordering and using the same index for joins & filters. We're also working on parallelisation so we can take advantage of multiple cores.

Features

A few things that we're planning to work on in 2015:

- the fastread package, to make it easy to get files off disk and in to memory, analogous to `fread()`.
- More flexible joins, including support for non-equi-joins.
- More flexible grouping like bootstrap samples, rollups and more

I'm also investing time into improving R's [database connectors](#), the ability to talk to [web apis](#), and making it easier to [scrape html pages](#).

edited Jan 8 at 16:55

answered Jan 8 at 12:39



hadley

45.2k ● 11 ● 84 ● 152

-
- 7 Just a side note, I do agree with many of your arguments (although I prefer the `data.table` syntax myself), but you can easily use `%>%` in order to pipe `data.table` operations if you don't like `[]` style. `%>%` is not specific to `dplyr`, rather comes from a separate package (which you happen to be the co-author of too), so I'm not sure I understand what are you trying to say in most of your **Syntax** paragraph. – [David Arenburg](#) Jan 8 at 12:54
-
- 2 @DavidArenburg good point. I've re-written syntax to hopefully make more clear what my main points are, and to highlight that you can use `%>%` with `data.table` – [hadley](#) Jan 8 at 13:28
-
- 2 Thanks Hadley, this is a useful perspective. Re indenting I typically do `DT[\n\texpression\n]` `[\texpression\n]` ([gist](#)) which actually works pretty well. I'm keeping Arun's answer as the answer as he more directly answers my specific questions which are not so much about the accessibility of syntax, but I think this a good answer for people trying to get a general feel for the differences / commonalities between `dplyr` and `data.table`. – [BrodieG](#) Jan 8 at 13:39
-
- 3 @BrodieG putting "everything" which is needed for relation algebra in the `[]` operator forms **data.table queries** which can be direct analogous (and deep extension) to **SQL queries**. At the moment I don't see any of `[]` args redundant (except the `drop`), and yes there are already many of them :) – [jangorecki](#) Jan 8 at 20:01
-
- 3 Why working on fastread when there is already `fread()`? Wouldn't time be spent better on improving `fread()` or working on other (underdeveloped) things? – [EDi](#) Jan 28 at 12:17
-

In direct response to the Question Title...

`dplyr` **definitely** does things that `data.table` can not.

Your point #3

`dplyr` abstracts (or will) potential DB interactions

is a direct answer to your own question but isn't elevated to a high enough level. `dplyr` is truly an extendable front-end to multiple data storage mechanisms where as `data.table` is an extension to a single one.

Look at `dplyr` as a back-end agnostic interface, with all of the targets using the same grammar, where you can extend the targets and handlers at will. `data.table` is, from the `dplyr` perspective, one of those targets.

You will never (I hope) see a day that `data.table` attempts to translate your queries to create SQL statements that operate with on-disk or networked data stores.

`dplyr` **can possibly do things** `data.table` **will not or might not do as well.**

Based on the design of working in-memory, `data.table` could have a much more difficult time extending itself into parallel processing of queries than `dplyr`.

In response to the in-body questions...

Usage

Are there analytical tasks that are a lot easier to code with one or the other package *for people familiar with the packages* (i.e. some combination of keystrokes required vs. required level of esotericism, where less of each is a good thing).

This may seem like a punt but the real answer is no. People *familiar* with tools seem to use the either the one most familiar to them or the one that is actually the right one for the job at hand. With that being said, sometimes you want to present a particular readability, sometimes a level of performance, and when you have need for a high enough level of both you may just need another tool to go along with what you already have to make clearer abstractions.

Performance

Are there analytical tasks that are performed substantially (i.e. more than 2x) more efficiently in one package vs. another.

Again, no. `data.table` excels at being efficient in everything *it* does where `dplyr` gets the burden of being limited in some respects to the underlying data store and registered handlers.

This means when you run into a performance issue with `data.table` you can be pretty sure it is in your query function and if it *is* actually a bottleneck with `data.table` then you've won yourself the joy of filing a report. This is also true when `dplyr` is using `data.table` as the back-end; you *may* see *some* overhead from `dplyr` but odds are it is your query.

When `dplyr` has performance issues with back-ends you can get around them by registering a function for hybrid evaluation or (in the case of databases) manipulating the generated query prior to execution.

Also see the accepted answer to [when is plyr better than data.table?](#)

edited Nov 17 '14 at 13:56

answered Nov 16 '14 at 22:39



Thell

3,704 ● 17 ● 43

- 1 Cant dplyr wrap a data.table with tbl_dt? Why not just get the best of both worlds? – [aaa90210](#) Dec 9 '14 at 9:08
- 1 @aaa90210, see this [post](#) – [BrodieG](#) Dec 17 '14 at 1:26
- 5 You forget to mention the reverse statement "*data.table definitely does things that dplyr can not*" which is also true. – [jangorecki](#) Jan 5 at 16:26
- 1 @JanGorecki That's because I am not aware of a *capability* that `data.table` has that `dplyr` isn't capable of either directly or via handlers. There are *features* (as discussed in terms of speed, memory and syntax) that have been discussed as differences in response to the OPs qualitative (*poorly*) portion of the question yet I don't recall seeing *capabilities* that couldn't/can't be generalized and abstracted up a layer. – [Thell](#) Jan 5 at 18:53
- 10 Arun answer explains it well. Most important (in terms of performance) would be fread, update by reference, rolling joins, overlapping joins. I believe there are no any package (not only dplyr) which can compete with those features. A nice example can be last slide from [this](#) presentation. – [jangorecki](#) Jan 5 at 19:46

protected by [David Arenburg](#) Jan 12 at 9:47

Thank you for your interest in this question. Because it has attracted low-quality answers, posting an answer now requires 10 [reputation](#) on this site.

Would you like to answer one of these [unanswered questions](#) instead?