Working with list-columns in `data.table`

Tyson S. Barrett[1]

[1] Utah State University

Abstract

The use of *list-columns* in data frames and tibbles in the R statistical environment is well documented (e.g. Bryan, 2018), providing a cognitively efficient way to organize results of complex data (e.g. several statistical models, groupings of text, data summaries, or even graphics) with corresponding data. For example, one can store student information within classrooms, player information within teams, or even analyses within groups. This allows the text to be of variable sizes without overly complicating or adding redundancies to the structure of the data. In turn, this can reduce the difficulty to appropriately analyze the data stored in the list-column. Because of its efficiency and speed, being able to use `data.table` to work with list-columns would be beneficial in many data contexts (i.e. to reduce memory usage in large data sets). I show how one can create list-columns in a data table using the `by` argument in `data.table` and `purrr::map()`. This is done using an example data set on professional basketball player information. I compare the behavior of the `data.table` approach to the `dplyr::group_nest()` function. Results using `bench::mark()` show the speed and efficiency of using `data.table` to work with list-columns.

Keywords: data.table, dplyr, list-columns, nesting

Introduction

The use of *list-columns* in data frames and tibbles in the R statistical environment (R Core Team, 2018) provides a cognitively efficient way to organize complex data (e.g. several statistical models, groupings of text, data summaries, or even graphics) with corresponding data in a concise manner. It has become a common approach to wrangling data in the `tidyverse`, with functions across `dplyr` and `tidyr` providing functionality to work with list-columns (Bryan, 2018; Wickham et al., 2019; Wickham & Henry, 2019).

Correspondence concerning this article should be addressed to Tyson S. Barrett, 2800 Old Main, Logan, UT 84322. E-mail: tyson.barrett@usu.edu

This format is often called "nested" data, where information is, in essence, nested within a column of data. For example, one can store individual words of a verse in a list for each verse, chapter, book, and volume. This allows the text to be of variable sizes without overly complicating or adding redundancies to the structure of the data.

List-columns can also be used to nest students within classrooms, players within teams, and measures within individuals. This allows the user to do certain data manipulations or analyses within each group. This can ensure that accidentally including data from other groups does not occur. Furthermore, nesting can reduce the difficulty to appropriately analyze the data stored in the list-column. Using functions like `lapply()` or `purrr::map*()` makes further analysis of the nested data more intuitive and error-free.

Because of its efficiency and speed, being able to use `data.table` to work with list-columns would be beneficial in many data contexts (i.e. to reduce memory usage in large data sets). Herein, I show how one can create list-columns in a data table using `purrr::map()` and the `by` argument in `data.table`. I further highlight the `dplyr::group_nest()` function and show a slightly more efficient approach when using a data table. Results using `bench::mark()` show the speed and efficiency of using `data.table` to work with list-columns.

This article relies on several powerful packages, including `data.table`, `dplyr`, `bench`, `tidyr`, `papaja`, `stringr`, `ggplot2`, `ggbeeswarm`, `performance`, `rvest`, and `lobstr` (Aust & Barth, 2018; Clarke & Sherrill-Mix, 2017; Dowle & Srinivasan, 2019; Hester, 2019; Lüdecke, Makowski, & Waggoner, 2019; Wickham, 2016, 2019c, 2019b, 2019a; Wickham et al., 2019; Wickham & Henry, 2019).

Example Data

Throughout much of this paper, I will demonstrate the use of *list-columns* in `data.table` using data from NBA Stuffer will be scraped to get information on players from the 2017-2018 and 2018-2019 seasons. First, the HTML data are read in, the tables with player data by year are then extracted using a custom function, indicators are added, and then combined into a single data table for the player data.

```
url_2018 <- "https://www.nbastuffer.com/2017-2018-nba-player-stats/"
url_2019 <- "https://www.nbastuffer.com/2018-2019-nba-player-stats/"
players_2018 <- read_html(url_2018)
players_2019 <- read_html(url_2019)

extract_fun <- function(html){
  html_nodes(html, "table")[2] %>%
    html_table(fill = TRUE) %>%
    .[[1]]
}

player_2018 <-
  extract_fun(players_2018) %>%
  mutate(year = 2018,
```

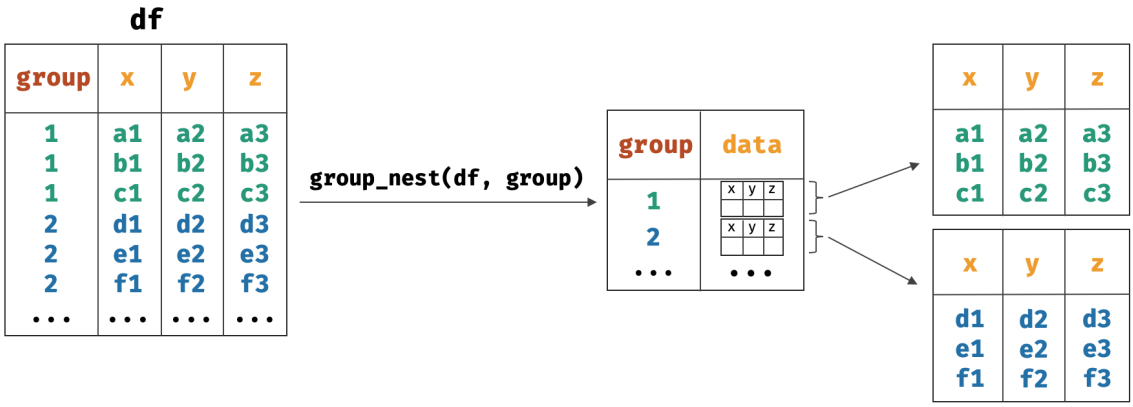

Figure 1. Diagram of one approach to creating a list-column in a data frame (i.e. creating a nested data frame).

```
        AGE = as.numeric(AGE))
player_2019 <-
  extract_fun(players_2019) %>%
  mutate(year = 2019)

players <-
  bind_rows(player_2018, player_2019) %>%
  clean_names() %>%
  rename(ppg = ppg_points_points_per_game,
         apg = apg_assists_assists_per_game) %>%
  data.table()
```

Below is a subset of this imported data set, showing only four of the variables and the first six rows.

```
##          full_name  mpg  ppg apg
## 1:    Aaron Brooks  5.9  2.3 0.6
## 2:    Aaron Gordon 32.9 17.6 2.3
## 3: Aaron Harrison 25.9  6.7 1.2
## 4:  Aaron Jackson 34.5  8.0 1.0
## 5:     Abdel Nader 10.9  3.0 0.5
## 6:  Adreian Payne  8.5  4.2 0.0
```

Nesting with `data.table`

In `dplyr` the `group_nest()` function is valuable when creating list-columns based on a grouping variable. It takes the data by group and puts it all in a list-column. Figure 1 highlights the process of taking a data frame and creating a nested data frame with a list-column. That is, all data from variables `x`, `y`, and `z` relating to each group is split into a distinct data frame and stored within the `data` column.

Overall, this function is efficient and fast but by using `data.table` it can be somewhat faster. This will be shown using the following function:

```
group_nest_dt <- function(dt, ..., .key = "data"){
  stopifnot(is.data.table(dt))

  by <- substitute(list(...))

  express <- dt[, list(list(.SD)), by = eval(by)]
  setnames(express, old = "V1", new = .key)
  express
}
```

In essence, this function takes a data table, then creates a list of the data table per group specified in the `by` argument.

```
group_nest_dt(players, team) %>%
  head()
```

```
##    team         data
## 1:  Min <data.table>
## 2:  Orl <data.table>
## 3:  Dal <data.table>
## 4:  Hou <data.table>
## 5:  Bos <data.table>
## 6:  Ind <data.table>
```

This is nearly identical to the `dplyr::group_nest()` function, in terms of output, but has data tables in the list-column instead of tibbles.

```
group_nest(players, team) %>%
  head()
```

```
## # A tibble: 6 x 2
##   team  data
##   <chr> <list>
## 1 Atl   <tibble [44 x 30]>
## 2 Bos   <tibble [37 x 30]>
## 3 Bro   <tibble [41 x 30]>
## 4 Cha   <tibble [34 x 30]>
## 5 Chi   <tibble [43 x 30]>
## 6 Cle   <tibble [49 x 30]>
```

Importantly, Figure 2 presents the timings from `bench::mark()` across the two approaches, showing `group_nest_dt()` is often faster. The memory allocated is very similar, with `group_nest_dt()` allocating 451KB and `group_nest()` allocating 335KB.

This nesting approach can be used with multiple grouping variables too. For example, I show how a user could nest by both `team` and `year`, as is done below.

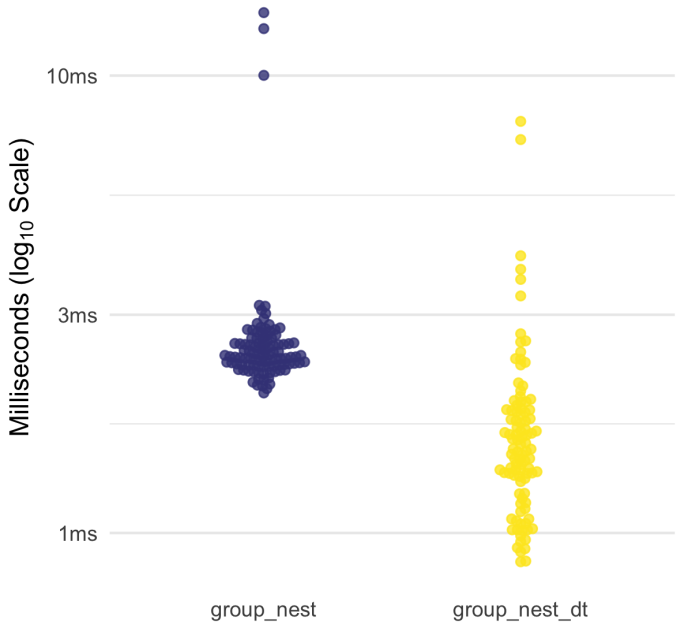

Figure 2. Speed comparisons for each nesting approach. Note the scale of the y-axis is $\log_{10}$.

```
group_nest_dt(players, team, year) %>%
  head()
```

```
##    team year         data
## 1:  Min 2018 <data.table>
## 2:  Orl 2018 <data.table>
## 3:  Dal 2018 <data.table>
## 4:  Hou 2018 <data.table>
## 5:  Bos 2018 <data.table>
## 6:  Ind 2018 <data.table>
```

Modeling within the Nest

Often, the nested data can provide an intuitive format to run several models to understand key features of the data within the groups. Below, the relationship between points-per-game and assists-per-game for each team and year is modeled and then the R^2 of the models are extracted. Since `performance::r2()` provides two versions of R^2, I then grab only the first of the two.

```
players_nested <- group_nest_dt(players, team, year) %>%
  .[, ppg_apg    := purrr::map(data, ~lm(ppg ~ apg, data = .x))] %>%
  .[, r2_list    := purrr::map(ppg_apg, ~performance::r2(.x))] %>%
  .[, r2_ppg_apg := purrr::map_dbl(r2_list, ~.x[[1]])]
head(players_nested)
```

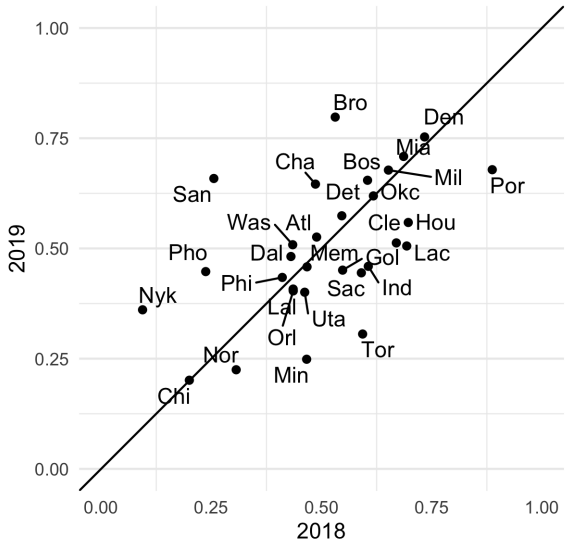

Figure 3. Example analysis performed using nested data to provide information for each team and year.

```
##    team year         data ppg_apg      r2_list r2_ppg_apg
## 1:  Min 2018 <data.table>    <lm> <r2_generic>  0.4662060
## 2:  Orl 2018 <data.table>    <lm> <r2_generic>  0.4357684
## 3:  Dal 2018 <data.table>    <lm> <r2_generic>  0.4305347
## 4:  Hou 2018 <data.table>    <lm> <r2_generic>  0.6967150
## 5:  Bos 2018 <data.table>    <lm> <r2_generic>  0.6043402
## 6:  Ind 2018 <data.table>    <lm> <r2_generic>  0.6060465
```

This produces two list-columns (`ppg_apg` and `r2_list`) and a numeric vector (`r2_ppg_apg`) all organized by team and year. This information is then readily available to plot. For example, one can look at how related points-per-game and assists-per-game are by team and year–in essence, showing which teams have players who both score and assist. This example is shown in Figure 3.

Unnest with `data.table`

After performing the manipulations or analyses within the nest, it can often be good to unnest to finalize analyses. Again, like with `group_nest_dt()`, this `unnest_dt()` function relies solely on the syntax of `data.table`, using the `j` and `by` arguments as shown below.

```
unnest_dt <- function(dt, col, id){
  stopifnot(is.data.table(dt))

  by <- substitute(id)
  col <- substitute(unlist(col, recursive = FALSE))

  dt[, eval(col), by = eval(by)]
```

```
}
```

This function can be used to unnest a data table, like the `players_nested` data table from earlier where the nested column is a data table, data frame, or tibble. Below, the `data` column in the table is unnested by `team` and `year` and then I select just a few of the variables.

```
unnest_dt(players_nested,
          col = data,
          id = list(team, year)) %>%
  .[, .(team, year, full_name, pos, age, gp, mpg)]
```

```
##       team year          full_name pos   age gp  mpg
##    1:  Min 2018       Aaron Brooks  PG 33.00 32  5.9
##    2:  Min 2018     Andrew Wiggins  SF 22.00 82 36.3
##    3:  Min 2018      Anthony Brown  SF 25.00  1  3.7
##    4:  Min 2018       Cole Aldrich   C 29.00 21  2.3
##    5:  Min 2018       Derrick Rose  PG 29.00  9 12.4
##   ---
## 1223:  Det 2019     Svi Mykhailiuk   G 21.84  3  6.6
## 1224:  Det 2019      Zaza Pachulia   C 35.16 68 12.9
## 1225:  Det 2019 Glenn Robinson III G-F 25.26 47 13.0
## 1226:  Det 2019          Ish Smith   G 30.77 56 22.3
## 1227:  Det 2019       Khyri Thomas   G 22.92 26  7.5
```

Again, this function is quick and efficient. Figure 4 presents the timings from `bench::mark()` across the two unnesting approaches, showing the `data.table` approach is often faster. The memory allocated is about half for the `data.table` approach here, with `unnest_dt()` allocating 912KB and `tidyr::unnest()` allocating 1.83MB.

A slight variation of this function can be used for list-columns with atomic vectors instead of data tables. A function like the following works well.

```
unnest_vec_dt <- function(dt, cols, id, nam){
  stopifnot(is.data.table(dt))

  by <- substitute(id)
  cols <- substitute(unlist(cols, recursive = FALSE, use.names = FALSE))

  express <- dt[, eval(cols), by = eval(by)]
  setnames(express, old = paste0("V", 1:length(nam)), new = nam)
  express
}
```

In `players_nested`, the `r2_list` column is a list of numeric vectors. This can be unnested as shown below, providing the two measures of R^2 per team per year.

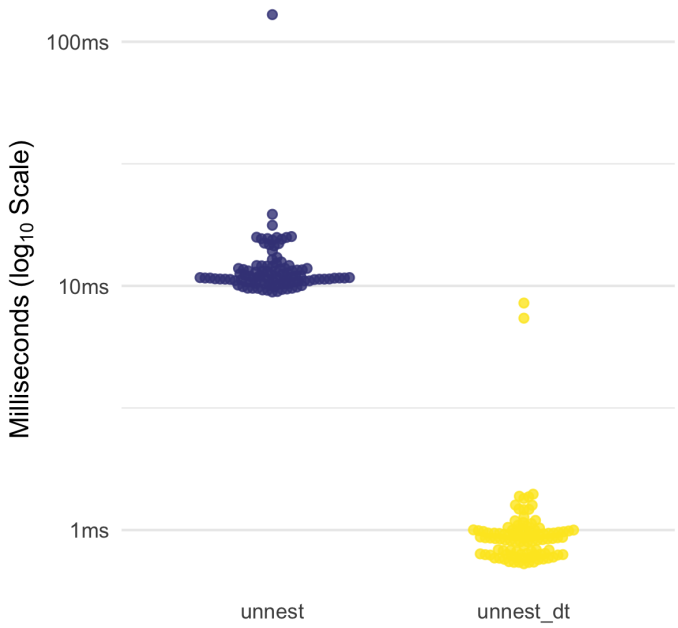

Figure 4. Speed comparisons for each unnesting approach. Note the scale of the y-axis is $\log_{10}$.

```
unnest_vec_dt(players_nested,
              cols = list(r2_list),
              id = list(team, year),
              nam = "r2")
```

```
##       team year        r2
##    1:  Min 2018  0.466206
##    2:  Min 2018 0.4280779
##    3:  Orl 2018 0.4357684
##    4:  Orl 2018 0.4025783
##    5:  Dal 2018 0.4305347
##   ---
## 116:  Lac 2019 0.4808586
## 117:  Phi 2019 0.4342685
## 118:  Phi 2019 0.4106964
## 119:  Det 2019 0.5740963
## 120:  Det 2019  0.550435
```

Memory Usage of List-Columns

Last item to assess herein is the computer memory usage of different formats of data tables with the same data. We can use the following large data sets in wide format, nested wide format, long format, and nested wide format to make brief comparisons.

```
# Wide
wide_format <- data.table(id = 1:1e6,
                          x1 = rnorm(1e6),
                          x2 = rnorm(1e6),
                          y1 = rnorm(1e6),
                          y2 = rnorm(1e6),
                          group = rbinom(1e6, 1, .5))
nested_wide_format <- group_nest_dt(wide_format, group)

# Long
long_format <- melt.data.table(wide_format,
                               id.vars = c("id", "group"),
                               measure.vars = c("x1", "x2", "y1", "y2"))
nested_long_format <- group_nest_dt(long_format, group)
```

I use the `lobstr` package to assess the object size of each format of the same data, shown in the output below.

```
##                Format Memory (MB)
## 1:        wide_format      40.002
## 2: nested_wide_format      36.003
## 3:        long_format      80.001
## 4: nested_long_format      64.003
```

Not surprising, the memory usage of nested data is lower than for its none nested corresponding data. This is directly related to the reduction in redundancies in the data otherwise there. That is, the nested data has far fewer rows containing the `group` variable. That, alone, in this large data saves memory. For example, the size of a single column of the `group` variable in wide format is 4 MB; and in long format it is 16 MB By reducing a single variable in this case, we save several megabytes of memory.

Discussion

List-columns are a useful approach to structuring data into a format that can be safely cleaned, manipulated, and analyzed by groups. It also provides for a more cognitively efficient way for a user to understand their data, allowing large data to be represented more concisely within groups.

The `tidyverse` provides several functions to work with nested data, which are relatively quick and efficient. For most data situations, these functions will do all that a user will need. However, in some situations, `data.table` can perform needed manipulations and analyses that cannot otherwise be done or that would take too long to complete. In these situations, and for users that prefer to use `data.table`, this tutorial can help provide direction in using list-columns.

Furthermore, as expected, the memory usage of nested data is lower than for its none nested corresponding data. This is due to the reduction in the redundancies present in wide

and long format. This suggests that it is not only the cognitive benefits to the user that makes this format more efficient.

Limitations

There are some notable limitations to list-columns in general, and in `data.table` specifically. First, the three custom functions built on `data.table` presented herein are not well-tested and are certainly not expected to work in each case where `dplyr::group_nest()`, `tidyr::unnest()`, and other tidy functions would work. Rather, they were presented to show how a user can leverage the speed and efficiency of `data.table` to create, and work with, list-columns.

Second, it is important to realize that nested data can remove the ability to use vectorization across groups. Depending on the analyses being conducted, this may slow down the computation to the point that nested data actually is a hindrance to performance.

Finally, when using list-columns in tibbles, the print method provides the dimensions of each corresponding nested tibble. This method is helpful in understanding the nested data without any need to extract it. This could be a minor, but valuable, update to the print method in `data.table`.

Conclusions

The use of list-columns in `data.table` is very similar to that in the `tidyverse`. It provides speed and efficiency in both nesting and unnesting the data, and can be used with the `purrr::map*()` and other powerful functions.

References

Aust, F., & Barth, M. (2018). *papaja: Create APA manuscripts with R Markdown*. Retrieved from https://github.com/crsh/papaja

Bryan, J. (2018). List columns (as part of "purrr tutorial"). Retrieved from https://jennybc.github.io/purrr-tutorial/ls13_list-columns.html

Clarke, E., & Sherrill-Mix, S. (2017). *Ggbeeswarm: Categorical scatter (violin point) plots*. Retrieved from https://github.com/eclarke/ggbeeswarm

Dowle, M., & Srinivasan, A. (2019). *Data.table: Extension of 'data.frame'*. Retrieved from https://CRAN.R-project.org/package=data.table

Hester, J. (2019). *Bench: High precision timing of r expressions*. Retrieved from https://CRAN.R-project.org/package=bench

Lüdecke, D., Makowski, D., & Waggoner, P. (2019). *Performance: Assessment of regression models performance*. Retrieved from https://easystats.github.io/performance/

R Core Team. (2018). *R: A language and environment for statistical computing*. Vienna, Austria: R Foundation for Statistical Computing. Retrieved from https://www.R-project.org/

Wickham, H. (2016). *Ggplot2: Elegant graphics for data analysis*. Springer-Verlag New York. Retrieved from https://ggplot2.tidyverse.org

Wickham, H. (2019a). *Lobstr: Visualize r data structures with trees*. Retrieved from https://CRAN.R-project.org/package=lobstr

Wickham, H. (2019b). *Rvest: Easily harvest (scrape) web pages*. Retrieved from https://CRAN.R-project.org/package=rvest

Wickham, H. (2019c). *Stringr: Simple, consistent wrappers for common string operations*. Retrieved from https://CRAN.R-project.org/package=stringr

Wickham, H., François, R., Henry, L., & Müller, K. (2019). *Dplyr: A grammar of data manipulation*. Retrieved from https://CRAN.R-project.org/package=dplyr

Wickham, H., & Henry, L. (2019). *Tidyr: Tidy messy data*. Retrieved from https://CRAN.R-project.org/package=tidyr