One of the concepts we teach in both [Practical Data Science with R](http://www.practicaldatascience.com/) and in our [theory of data shaping](https://winvector.github.io/cdata/) is the importance of identifying the roles of columns in your data.

For example, to [think in terms of multi-row records](http://www.win-vector.com/blog/2018/10/designing-transforms-for-data-reshaping-with-cdata/) it helps to identify:

* Which columns are keys (together identify rows or records).
* Which columns are data/payload (are considered free varying data).
* Which columns are "derived" (functions of the keys).

In this note we will show how to use some of these ideas to write safer data-wrangling code.

In mathematics the statement "y is a function of x" merely means there is an ideal lookup table with which if you knew the value of x, then you could in principle know the value of y.

For example in the following [R](https://www.r-project.org/) data.frame: y is a function of x, as all rows that have the same value for x also have the same value for y.

d1 <- data.frame(

x = c("a", "a", "b", "b", "c"),

y = c(10, 10, 20, 20, 20),

z = c(1, 2, 1, 1, 1),

stringsAsFactors = FALSE)

print(d1)

## x y z

## 1 a 10 1

## 2 a 10 2

## 3 b 20 1

## 4 b 20 1

## 5 c 20 1

Notice if we know the value of x we then, in principle know the value of y. In the same example z is not a function of x, as it does not have this property.

A more concrete example would be: user-name is a function of user-ID. If you know the an individual’s user-ID, then you also (if you have the right lookup table) know the individual’s user-name.

We first taught these concepts in the context of [SQL](https://en.wikipedia.org/wiki/SQL) and SQL grouped aggregation. In SQL once you aggregate on one column, then all other columns in your query must either be the grouping columns, or also aggregated. This is easiest to show in code, but we will use the dplyr package for our example.

library("dplyr")

d1 %>%

group\_by(x) %>%

summarize(y = max(y)) %>%

ungroup()

## # A tibble: 3 x 2

## x y

##

## 1 a 10

## 2 b 20

## 3 c 20

Notice only grouping columns and columns passed through an aggregating calculation (such as max()) are passed through (the column z is not in the result). Now because y is a function of x no substantial aggregation is going on, we call this situation a "pseudo aggregation" and [we have taught this before](http://www.win-vector.com/blog/2017/07/better-grouped-summaries-in-dplyr/).

Our [wrapr](https://cran.r-project.org/package=wrapr) package now supplies a special case pseudo-aggregator (or in a mathematical sense: projection): [psagg()](https://winvector.github.io/wrapr/reference/psagg.html). It works as follows.

library("wrapr")

d1 %>%

group\_by(x) %>%

summarize(y = psagg(y)) %>%

ungroup()

## # A tibble: 3 x 2

## x y

##

## 1 a 10

## 2 b 20

## 3 c 20

psagg() pretty much worked the same as the earlier max(). However, it documents our belief that y is a function of x (that nothing interesting is going on for this column during aggregation). Where psagg() differs is if our assumption that y is a function of x is violated.

d2 <- data.frame(

x = c("a", "a", "b", "b", "c"),

y = c(10, 10, 20, 23, 20),

stringsAsFactors = FALSE)

print(d2)

## x y

## 1 a 10

## 2 a 10

## 3 b 20

## 4 b 23

## 5 c 20

d2 %>%

group\_by(x) %>%

summarize(y = psagg(y)) %>%

ungroup()

## Error in summarise\_impl(.data, dots): Evaluation error: wrapr::psagg argument values are varying.

The code caught that our assumption was false and raised on error. This sort of checking can save a lot of time and prevent erroneous results.

And that is part of what we teach:

* document intent
* check assumptions
* double-check results
* use composable small helper-tools.

psagg() also works well with data.table.

library("data.table")

as.data.table(d1)[

, .(y = psagg(y)), by = "x"]

## x y

## 1: a 10

## 2: b 20

## 3: c 20

as.data.table(d2)[

, .(y = psagg(y)), by = "x"]

## Error in psagg(y): wrapr::psagg argument values are varying

Of course we don’t strictly *need* psagg() as we could insert checks by hand (though this would become burdensome if we had many derived columns).

as.data.table(d1)[

, .(y = max(y),

y\_was\_const = min(y)==max(y)),

by = "x"]

## x y y\_was\_const

## 1: a 10 TRUE

## 2: b 20 TRUE

## 3: c 20 TRUE

as.data.table(d2)[

, .(y = max(y),

y\_was\_const = min(y)==max(y)),

by = "x"]

## x y y\_was\_const

## 1: a 10 TRUE

## 2: b 23 FALSE

## 3: c 20 TRUE

Unfortunately, this sort of checking does not currently work for dplyr.

packageVersion("dplyr")

## [1] '0.7.7'

d1 %>%

group\_by(x) %>%

summarize(y = max(y),

y\_was\_const = min(y)==max(y)) %>%

ungroup()

## # A tibble: 3 x 3

## x y y\_was\_const

##

## 1 a 10 TRUE

## 2 b 20 TRUE

## 3 c 20 TRUE

d2 %>%

group\_by(x) %>%

summarize(y = max(y),

y\_was\_const = min(y)==max(y)) %>%

ungroup()

## # A tibble: 3 x 3

## x y y\_was\_const

##

## 1 a 10 TRUE

## 2 b 23 TRUE

## 3 c 20 TRUE

Notice the per-group variation in y was not detected. This appears to be a dplyr un-caught result corruption issue.

If we don’t attempt to re-use the variable name we get the correct result.

d2 %>%

group\_by(x) %>%

summarize(y\_for\_group = max(y),

y\_was\_const = min(y)==max(y)) %>%

ungroup()

## # A tibble: 3 x 3

## x y\_for\_group y\_was\_const

##

## 1 a 10 TRUE

## 2 b 23 FALSE

## 3 c 20 TRUE

However, being forced to rename variables during aggregation is a needless user burden.

Our [seplyr](https://winvector.github.io/seplyr/) package (a package that is a very thin adapter on top of dplyr) does issue a warning in the failing situation.

library("seplyr")

d2 %>%

group\_by\_se("x") %>%

summarize\_nse(y = max(y),

y\_was\_const = min(y)==max(y)) %>%

ungroup()

## Warning in summarize\_se(res, summarizeTerms, warn = summarize\_nse\_warn, :

## seplyr::summarize\_se possibly confusing column name re-use c('y' =

## 'max(y)', 'y\_was\_const' = 'min(y) == max(y)')

## # A tibble: 3 x 3

## x y y\_was\_const

##

## 1 a 10 TRUE

## 2 b 23 TRUE

## 3 c 20 TRUE

The result is still wrong, but at least in this situation the user has a better chance at noticing and working around the issue.

The problem may not be common (it may or may not be in any of your code, or code you use), and is of course easy to avoid (once you know the nature of the issue).