Let’s try some "ugly corner cases" for data manipulation in [R](https://www.r-project.org/). Corner cases are examples where the user might be running to the edge of where the package developer intended their package to work, and thus often where things can go wrong.

Let’s see what happens when we try to stick a fork in the power-outlet.

For our example let’s set up some data.frames.

# create some exmaple tables

d1 <- data.frame(v1 = c("a", "b"),

stringsAsFactors = FALSE)

d2 <- data.frame(v2 = c("x", "y"),

stringsAsFactors = FALSE)

d3 <- data.frame(x = 1, y = 2, z = 3)

And we will also copy this data into a database.

# copy the example tables to a database

db <- DBI::dbConnect(RSQLite::SQLite(),

":memory:")

DBI::dbWriteTable(db, "d1d", d1)

DBI::dbWriteTable(db, "d2d", d2)

DBI::dbWriteTable(db, "d3d", d3)

Now let’s try a couple of basic tasks using dplyr.

# try to use dplyr to work with the original tables

# in memory

suppressPackageStartupMessages(library("dplyr"))

## Warning: package 'dplyr' was built under R version 3.5.2

packageVersion("dplyr")

## [1] '0.8.0.1'

First we try a [cross-join](https://en.wikipedia.org/wiki/Join_(SQL)#Cross_join).

# try to do a cross-join

left\_join(d1, d2, by = character(0))

## Error: `by` must specify variables to join by

# join is rejected

Obviously that is something deliberately prohibited by the package authors.

Let’s try dividing all columns by a value.

# try to divide all columns by y

mutate\_all(d3, ~(./y))

## x y z

## 1 0.5 1 3

# z is divided by 1, not by the original

# y-value

Something went wrong, notice z was not altered. We will return to this later.

Now let’s try the same operations using dplyr on a database.

# try the same calculations in database

# using dplyr

packageVersion("dbplyr")

## [1] '1.3.0'

# get references to the database tables

d1d <- dplyr::tbl(db, "d1d")

d2d <- dplyr::tbl(db, "d2d")

d3d <- dplyr::tbl(db, "d3d")

# try to do a cross-join

left\_join(d1d, d2d,

by = character(0))

## # Source: lazy query [?? x 2]

## # Database: sqlite 3.22.0 [:memory:]

## v1 v2

##

## 1 a x

## 2 a y

## 3 b x

## 4 b y

# this time it works

In this case the cross join works (as databases expect to support this operation).

Okay, let’s try the divide by columns example again.

# try to divide all columns by y

mutate\_all(d3d, ~(./y))

## Error in (structure(function (..., .x = ..1, .y = ..2, . = ..1) : object 'x' not found

# this time it errors-out

Boom (ouch).

Now let’s try that again with the rquery package.

# try the join task using rquery

library("rquery")

# get references to the database tables

d1r <- rquery::db\_td(db, "d1d")

d2r <- rquery::db\_td(db, "d2d")

d3r <- rquery::db\_td(db, "d3d")

# try the cross-join

ops <- natural\_join(d1r, d2r,

by = character(0),

jointype = "LEFT")

execute(db, ops)

## v1 v2

## 1 a x

## 2 a y

## 3 b x

## 4 b y

The cross-join worked in the database.

Now we start on the column division. We have to specify the set of operations explicitly, but that really isn’t too big of burden (as rquery supplies tools for this).

# try to get column alterations

vars <- colnames(d3)

expressions <- paste(vars, "%:=%", vars, "/", "y")

print(expressions)

## [1] "x %:=% x / y" "y %:=% y / y" "z %:=% z / y"

ops2 <- extend\_se(d3r, expressions)

cat(format(ops2))

## table(`d3d`;

## x,

## y,

## z) %.>%

## extend(.,

## x %:=% x / y) %.>%

## extend(.,

## y %:=% y / y) %.>%

## extend(.,

## z %:=% z / y)

# Oh! This isn't what I want, see

# how y gets updated before it is used on column z.

We took the extra step of examining the operations before we tried them. Notice the rquery::extend() operation got factored into explicit steps. This introduces inspectable guarantees as to what value each column has at each step. This is enough to show us that the y column will be all ones before it is used to try and adjust the z column. This is not what we wanted, though this query is guaranteed to have similar semantics both on-database and in-memory.

Now let’s build, examine and execute a better version of the query.

# try to get column alterations again

vars <- setdiff(colnames(d3), "y")

expressions <- paste(vars, "%:=%", vars, "/", "y")

print(expressions)

## [1] "x %:=% x / y" "z %:=% z / y"

ops2b <- d3r %.>%

extend\_se(., expressions) %.>%

extend(., y = 1)

cat(format(ops2b))

## table(`d3d`;

## x,

## y,

## z) %.>%

## extend(.,

## x %:=% x / y,

## z %:=% z / y) %.>%

## extend(.,

## y := 1)

execute(db, ops2b)

## x z y

## 1 0.5 1.5 1

With the rqdatatable package we can use data.table to process in-R data.

# try the join task using rqdatatable

library("rqdatatable")

# immediate notation

natural\_join(d1, d2, by = character(0))

## v1 v2

## 1: a x

## 2: a y

## 3: b x

## 4: b y

# join task in operators notation

ex\_data\_table(ops,

tables = list(d1d = d1, d2d = d2))

## v1 v2

## 1: a x

## 2: a y

## 3: b x

## 4: b y

# column alterations in operators notation

ex\_data\_table(ops2b,

tables = list(d3d = d3))

## x y z

## 1: 0.5 1 1.5

And there we are. Frankly we didn’t really have to stretch things very far to break things (including building up non-signalling data mistakes; inspect your intermediate results!!).

We have been trying to keep the number of unexpected behaviors in rquery down by keeping the rquery implementation very simple (and very thin) and then relying on either the database or data.table for the actual implementation and semantics. There are, of course, going to be cases where rquery needs a fix- but we have been able to find and apply such fixes quite quickly. We have also found fixing rquery is *much* faster than coding around bugs.

# clean up

DBI::dbDisconnect(db)