This example may seem extreme or unnatural. However we have seen once you expose a system to enough users you see a lot more extreme use cases than you would at first expect. We have actually seen large tens of columns added to a mart in a large irregular block (so not the same transform for each columns) by building up long pipelines, so this simplified example is in fact relevant to production deployments.

First set up our packages, database connection, and remote table.

library("dplyr")

## Warning: package 'dplyr' was built under R version 3.5.1

##

## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':

##

## filter, lag

## The following objects are masked from 'package:base':

##

## intersect, setdiff, setequal, union

library("rquery")

library("microbenchmark")

library("ggplot2")

library("WVPlots")

library("rqdatatable")

library("cdata")

use\_spark <- TRUE

# connect

if(use\_spark) {

conf <- sparklyr::spark\_config()

conf$spark.yarn.am.cores <- 2

conf$spark.executor.cores <- 2

mem\_size <- "4G"

conf$spark.executor.memory <- mem\_size

conf$spark.yarn.am.memory <- mem\_size

conf$`sparklyr.shell.driver-memory` <- mem\_size

conf$`sparklyr.shell.executor-memory` <- mem\_size

conf$`spark.yarn.executor.memoryOverhead` <- mem\_size

con <- sparklyr::spark\_connect(version='2.2.0',

master = "local",

config = conf)

} else {

con <- DBI::dbConnect(RPostgreSQL::PostgreSQL(),

host = 'localhost',

port = 5432,

user = 'johnmount',

password = '')

}

# configure rquery connection options

dbopts <- rq\_connection\_tests(con)

db\_hdl <- rquery\_db\_info(

connection = con,

is\_dbi = TRUE,

connection\_options = dbopts)

print(db\_hdl)

## [1] "rquery\_db\_info(DBIConnection\_spark\_connection\_spark\_shell\_connection, is\_dbi=TRUE, note=\"\")"

nrow <- 1000000

td <- rq\_copy\_to(db\_hdl,

"d",

data.frame(x = seq\_len(nrow)),

overwrite = TRUE,

temporary = TRUE)

tbl <- dplyr::tbl(con, "d")

ncol <- 100

rquery torture function: add 100 columns to a 1000000 row table.

rquery\_fn <- function(db\_hdl, td, ncol, return\_sql = FALSE) {

expressions <- character(0)

for(i in seq\_len(ncol)) {

expri <- paste0("x\_", i) %:=% paste0("x + ", i)

expressions <- c(expressions, expri)

}

ops <- td %.>%

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

select\_rows\_nse(., x == 3)

if(return\_sql) {

return(to\_sql(ops, db\_hdl))

}

# force execution

db\_hdl %.>% ops

}

cat(rquery\_fn(db\_hdl, td, 5, return\_sql = TRUE))

## SELECT \* FROM (

## SELECT

## `x`,

## `x` + 1 AS `x\_1`,

## `x` + 2 AS `x\_2`,

## `x` + 3 AS `x\_3`,

## `x` + 4 AS `x\_4`,

## `x` + 5 AS `x\_5`

## FROM (

## SELECT

## `x`

## FROM

## `d`

## ) tsql\_97238965696940256963\_0000000000

## ) tsql\_97238965696940256963\_0000000001

## WHERE `x` = 3

rquery\_fn(db\_hdl, td, 5)

## x x\_1 x\_2 x\_3 x\_4 x\_5

## 1 3 4 5 6 7 8

The row-selection step is cut down on the in-memory cost of bringing the result back to R. Obviously we could optimize the example away by pivoting the filter to earlier in the example pipeline. We ask the reader to take this example as a stand-in for a more complicated (though nasty) real-world example where such optimizations are not available.

Same torture for dplyr.

dplyr\_fn <- function(tbl, ncol, return\_sql = FALSE) {

pipeline <- tbl

xvar <- rlang::sym("x")

for(i in seq\_len(ncol)) {

res\_i <- rlang::sym(paste0("x\_", i))

pipeline <- pipeline %>%

mutate(., !!res\_i := !!xvar + i)

}

pipeline <- pipeline %>%

filter(., x == 3)

if(return\_sql) {

return(dbplyr::remote\_query(pipeline))

}

# force execution

pipeline %>% collect(.)

}

cat(dplyr\_fn(tbl, 5, return\_sql = TRUE))

## SELECT \*

## FROM (SELECT `x`, `x\_1`, `x\_2`, `x\_3`, `x\_4`, `x` + 5 AS `x\_5`

## FROM (SELECT `x`, `x\_1`, `x\_2`, `x\_3`, `x` + 4 AS `x\_4`

## FROM (SELECT `x`, `x\_1`, `x\_2`, `x` + 3 AS `x\_3`

## FROM (SELECT `x`, `x\_1`, `x` + 2 AS `x\_2`

## FROM (SELECT `x`, `x` + 1 AS `x\_1`

## FROM `d`) `tyiyhhxjag`) `gshhunpiup`) `teowzjcshb`) `hdrfwlzycc`) `lsniejpwft`

## WHERE (`x` = 3.0)

dplyr\_fn(tbl, 5)

## # A tibble: 1 x 6

## x x\_1 x\_2 x\_3 x\_4 x\_5

##

## 1 3 4 5 6 7 8

We can also collect expressions efficiently using seplyr (seplyr is a thin wrapper over dplyr, so seplyr‘s method mutate\_se() is essentially instructions how to do the same thing using rlang).

seplyr\_fn <- function(tbl, ncol, return\_sql = FALSE) {

expressions <- character(0)

for(i in seq\_len(ncol)) {

expri <- paste0("x\_", i) %:=% paste0("x + ", i)

expressions <- c(expressions, expri)

}

pipeline <- tbl %>%

seplyr::mutate\_se(., expressions) %>%

filter(., x == 3)

if(return\_sql) {

return(dbplyr::remote\_query(pipeline))

}

# force execution

pipeline %>% collect(.)

}

cat(seplyr\_fn(tbl, 5, return\_sql = TRUE))

## SELECT \*

## FROM (SELECT `x`, `x` + 1.0 AS `x\_1`, `x` + 2.0 AS `x\_2`, `x` + 3.0 AS `x\_3`, `x` + 4.0 AS `x\_4`, `x` + 5.0 AS `x\_5`

## FROM `d`) `gaktzzkzxq`

## WHERE (`x` = 3.0)

seplyr\_fn(tbl, 5)

## # A tibble: 1 x 6

## x x\_1 x\_2 x\_3 x\_4 x\_5

##

## 1 3 4 5 6 7 8

Time the functions. Timing is not going to be certain given issues such as cluster state and query caching.

timings <- microbenchmark(

rquery = rquery\_fn(db\_hdl, td, ncol),

dplyr = dplyr\_fn(tbl, ncol),

seplyr = seplyr\_fn(tbl, ncol),

times = 10L)

saveRDS(timings, "CollectExprs\_timings.RDS")

Present the results.

print(timings)

## Unit: milliseconds

## expr min lq mean median uq max neval

## rquery 995.955 1018.481 1153.364 1065.092 1281.715 1502.717 10

## dplyr 2156.264 2219.900 2899.534 2473.929 3791.673 4714.063 10

## seplyr 1074.775 1180.591 1453.980 1273.424 1598.804 2398.883 10

#autoplot(timings)

timings <- as.data.frame(timings)

timings$seconds <- timings$time/10^9

timings$method <- factor(timings$expr)

timings$method <- reorder(timings$method, timings$seconds)

WVPlots::ScatterBoxPlotH(timings, "seconds", "method", "task time by method")

tratio <- timings %.>%

project\_nse(.,

groupby = "method",

mean\_seconds = mean(seconds)) %.>%

pivot\_to\_rowrecs(.,

columnToTakeKeysFrom = "method",

columnToTakeValuesFrom = "mean\_seconds",

rowKeyColumns = NULL) %.>%

extend\_nse(.,

ratio = dplyr/rquery)

tratio[]

## dplyr rquery seplyr ratio

## 1: 2.899534 1.153364 1.45398 2.513979

ratio\_str <- sprintf("%.2g", tratio$ratio)

rquery is about 2.5 times faster than dplyr for this task at this scale for this data implementation and configuration (we have also seen an over 8 times difference for this example on PostgreSQL).

if(use\_spark) {

sparklyr::spark\_disconnect(con)

} else {

DBI::dbDisconnect(con)

}