**Introduction**

In this note we will show how to speed up work in [R](https://www.r-project.org/) by partitioning data and process-level parallelization. We will show the technique with three different R packages: rqdatatable, [data.table](https://cran.r-project.org/package=data.table), and [dplyr](https://cran.r-project.org/package=dplyr). The methods shown will also work with base-R and other packages.

The partitioning is specified by the user preparing a grouping column that tells the system which sets of rows must be kept together in a correct calculation. We are going to try to demonstrate everything with simple code examples, and minimal discussion.

Keep in mind: unless the pipeline steps have non-trivial cost, the overhead of partitioning and distributing the work may overwhelm any parallel speedup. Also data.table itself already seems to exploit some thread-level parallelism (notice user time is greater than elapsed time). That being said, in this note we will demonstrate a synthetic example where computation is expensive due to a blow-up in an intermediate join step.

**Our example**

First we set up our execution environment and example (some details: OSX 10.13.4 on a 2.8 GHz Intel Core i5 Mac Mini (Late 2015 model) with 8GB RAM and hybrid disk drive).

library("rqdatatable")

Library(“rquery”)

library("microbenchmark")

library("ggplot2")

library("WVPlots")

suppressPackageStartupMessages(library("dplyr"))

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

base::date()

## [1] "Sun Jul 8 09:05:25 2018"

R.version.string

## [1] "R version 3.5.0 (2018-04-23)"

parallel::detectCores()

## [1] 4

packageVersion("parallel")

## [1] '3.5.0'

packageVersion("rqdatatable")

## [1] '0.1.2'

packageVersion("rquery")

## [1] '0.5.1'

packageVersion("dplyr")

## [1] '0.7.6'

ncore <- parallel::detectCores()

print(ncore)

## [1] 4

cl <- parallel::makeCluster(ncore)

print(cl)

## socket cluster with 4 nodes on host 'localhost'

set.seed(2362)

mk\_example <- function(nkey, nrep, ngroup = 20) {

keys <- paste0("key\_", seq\_len(nkey))

key\_group <- sample(as.character(seq\_len(ngroup)),

length(keys), replace = TRUE)

names(key\_group) <- keys

key\_table <- data.frame(

key = rep(keys, nrep),

stringsAsFactors = FALSE)

key\_table$data <- runif(nrow(key\_table))

instance\_table <- data.frame(

key = rep(keys, nrep),

stringsAsFactors = FALSE)

instance\_table$id <- seq\_len(nrow(instance\_table))

instance\_table$info <- runif(nrow(instance\_table))

# groups should be no finer than keys

key\_table$key\_group <- key\_group[key\_table$key]

instance\_table$key\_group <- key\_group[instance\_table$key]

list(key\_table = key\_table,

instance\_table = instance\_table)

}

dlist <- mk\_example(10, 10)

data <- dlist$instance\_table

annotation <- dlist$key\_table

**rquery / rqdatatable**

rquery and rqdatatable can implement a non-trivial calculation as follows.

# possible data lookup: find rows that

# have lookup data <= info

optree <- local\_td(data) %.>%

natural\_join(.,

local\_td(annotation),

jointype = "INNER",

by = "key") %.>%

select\_rows\_nse(., data <= info) %.>%

pick\_top\_k(.,

k = 1,

partitionby = "id",

orderby = "data",

reverse = "data",

keep\_order\_column = FALSE) %.>%

orderby(., "id")

cat(format(optree))

## table('data';

## key,

## id,

## info,

## key\_group) %.>%

## natural\_join(.,

## table('annotation';

## key,

## data,

## key\_group),

## j= INNER, by= key) %.>%

## select\_rows(.,

## data <= info) %.>%

## extend(.,

## row\_number := row\_number(),

## p= id,

## o= "data" DESC) %.>%

## select\_rows(.,

## row\_number <= 1) %.>%

## drop\_columns(.,

## row\_number) %.>%

## orderby(., id)

res1 <- ex\_data\_table(optree)

head(res1)

## data id info key key\_group

## 1: 0.9152014 1 0.9860654 key\_1 20

## 2: 0.5599810 2 0.5857570 key\_2 8

## 3: 0.3011882 3 0.3334490 key\_3 10

## 4: 0.3650987 4 0.3960980 key\_4 5

## 5: 0.1469254 5 0.1753649 key\_5 14

## 6: 0.2567631 6 0.3510280 key\_6 7

nrow(res1)

## [1] 94

And we can execute the operations in parallel.

parallel::clusterEvalQ(cl,

library("rqdatatable"))

## [[1]]

## [1] "rqdatatable" "rquery" "stats" "graphics" "grDevices"

## [6] "utils" "datasets" "methods" "base"

##

## [[2]]

## [1] "rqdatatable" "rquery" "stats" "graphics" "grDevices"

## [6] "utils" "datasets" "methods" "base"

##

## [[3]]

## [1] "rqdatatable" "rquery" "stats" "graphics" "grDevices"

## [6] "utils" "datasets" "methods" "base"

##

## [[4]]

## [1] "rqdatatable" "rquery" "stats" "graphics" "grDevices"

## [6] "utils" "datasets" "methods" "base"

res2 <- ex\_data\_table\_parallel(optree,

"key\_group",

cl)

head(res2)

## data id info key key\_group

## 1: 0.9152014 1 0.9860654 key\_1 20

## 2: 0.5599810 2 0.5857570 key\_2 8

## 3: 0.3011882 3 0.3334490 key\_3 10

## 4: 0.3650987 4 0.3960980 key\_4 5

## 5: 0.1469254 5 0.1753649 key\_5 14

## 6: 0.2567631 6 0.3510280 key\_6 7

nrow(res2)

## [1] 94

**data.table**

[data.table](http://r-datatable.com/) can implement the same function.

library("data.table")

##

## Attaching package: 'data.table'

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

##

## between, first, last

packageVersion("data.table")

## [1] '1.11.4'

data\_table\_f <- function(data, annotation) {

data <- data.table::as.data.table(data)

annotation <- data.table::as.data.table(annotation)

joined <- merge(data, annotation,

by = "key",

all=FALSE,

allow.cartesian=TRUE)

joined <- joined[joined$data <= joined$info, ]

data.table::setorderv(joined, cols = "data")

joined <- joined[, .SD[.N], id]

data.table::setorderv(joined, cols = "id")

}

resdt <- data\_table\_f(data, annotation)

head(resdt)

## id key info key\_group.x data key\_group.y

## 1: 1 key\_1 0.9860654 20 0.9152014 20

## 2: 2 key\_2 0.5857570 8 0.5599810 8

## 3: 3 key\_3 0.3334490 10 0.3011882 10

## 4: 4 key\_4 0.3960980 5 0.3650987 5

## 5: 5 key\_5 0.1753649 14 0.1469254 14

## 6: 6 key\_6 0.3510280 7 0.2567631 7

nrow(resdt)

## [1] 94

We can also run data.table in parallel using wrapr::execute\_parallel:

Library(wrapr)

parallel::clusterEvalQ(cl, library("data.table"))

## [[1]]

## [1] "data.table" "rqdatatable" "rquery" "stats" "graphics"

## [6] "grDevices" "utils" "datasets" "methods" "base"

##

## [[2]]

## [1] "data.table" "rqdatatable" "rquery" "stats" "graphics"

## [6] "grDevices" "utils" "datasets" "methods" "base"

##

## [[3]]

## [1] "data.table" "rqdatatable" "rquery" "stats" "graphics"

## [6] "grDevices" "utils" "datasets" "methods" "base"

##

## [[4]]

## [1] "data.table" "rqdatatable" "rquery" "stats" "graphics"

## [6] "grDevices" "utils" "datasets" "methods" "base"

parallel::clusterExport(cl, "data\_table\_f")

dt\_f <- function(tables\_list) {

data <- tables\_list$data

annotation <- tables\_list$annotation

data\_table\_f(data, annotation)

}

data\_table\_parallel\_f <- function(data, annotation) {

respdt <- wrapr::execute\_parallel(

tables = list(data = data,

annotation = annotation),

f = dt\_f,

partition\_column = "key\_group",

cl = cl) %.>%

data.table::rbindlist(.)

data.table::setorderv(respdt, cols = "id")

respdt

}

respdt <- data\_table\_parallel\_f(data, annotation)

head(respdt)

## id key info key\_group.x data key\_group.y

## 1: 1 key\_1 0.9860654 20 0.9152014 20

## 2: 2 key\_2 0.5857570 8 0.5599810 8

## 3: 3 key\_3 0.3334490 10 0.3011882 10

## 4: 4 key\_4 0.3960980 5 0.3650987 5

## 5: 5 key\_5 0.1753649 14 0.1469254 14

## 6: 6 key\_6 0.3510280 7 0.2567631 7

nrow(respdt)

## [1] 94

**dplyr**

[dplyr](https://cran.r-project.org/package=dplyr) can also implement the example.

dplyr\_pipeline <- function(data, annotation) {

res <- data %>%

inner\_join(annotation, by = "key") %>%

filter(data <= info) %>%

group\_by(id) %>%

arrange(-data) %>%

mutate(rownum = row\_number()) %>%

ungroup() %>%

filter(rownum == 1) %>%

arrange(id)

res

}

resd <- dplyr\_pipeline(data, annotation)

head(resd)

## # A tibble: 6 x 7

## key id info key\_group.x data key\_group.y rownum

##

## 1 key\_1 1 0.986 20 0.915 20 1

## 2 key\_2 2 0.586 8 0.560 8 1

## 3 key\_3 3 0.333 10 0.301 10 1

## 4 key\_4 4 0.396 5 0.365 5 1

## 5 key\_5 5 0.175 14 0.147 14 1

## 6 key\_6 6 0.351 7 0.257 7 1

nrow(resd)

## [1] 94

And we can use wrapr::execute\_parallel to parallelize the dplyr solution.

parallel::clusterEvalQ(cl, library("dplyr"))

## [[1]]

## [1] "dplyr" "data.table" "rqdatatable" "rquery" "stats"

## [6] "graphics" "grDevices" "utils" "datasets" "methods"

## [11] "base"

##

## [[2]]

## [1] "dplyr" "data.table" "rqdatatable" "rquery" "stats"

## [6] "graphics" "grDevices" "utils" "datasets" "methods"

## [11] "base"

##

## [[3]]

## [1] "dplyr" "data.table" "rqdatatable" "rquery" "stats"

## [6] "graphics" "grDevices" "utils" "datasets" "methods"

## [11] "base"

##

## [[4]]

## [1] "dplyr" "data.table" "rqdatatable" "rquery" "stats"

## [6] "graphics" "grDevices" "utils" "datasets" "methods"

## [11] "base"

parallel::clusterExport(cl, "dplyr\_pipeline")

dplyr\_f <- function(tables\_list) {

data <- tables\_list$data

annotation <- tables\_list$annotation

dplyr\_pipeline(data, annotation)

}

dplyr\_parallel\_f <- function(data, annotation) {

respdt <- wrapr::execute\_parallel(

tables = list(data = data,

annotation = annotation),

f = dplyr\_f,

partition\_column = "key\_group",

cl = cl) %>%

dplyr::bind\_rows() %>%

arrange(id)

}

respdplyr <- dplyr\_parallel\_f(data, annotation)

head(respdplyr)

## # A tibble: 6 x 7

## key id info key\_group.x data key\_group.y rownum

##

## 1 key\_1 1 0.986 20 0.915 20 1

## 2 key\_2 2 0.586 8 0.560 8 1

## 3 key\_3 3 0.333 10 0.301 10 1

## 4 key\_4 4 0.396 5 0.365 5 1

## 5 key\_5 5 0.175 14 0.147 14 1

## 6 key\_6 6 0.351 7 0.257 7 1

nrow(respdplyr)

## [1] 94

**Benchmark**

We can benchmark the various realizations.

dlist <- mk\_example(300, 300)

data <- dlist$instance\_table

annotation <- dlist$key\_table

timings <- microbenchmark(

data\_table\_parallel =

nrow(data\_table\_parallel\_f(data, annotation)),

data\_table = nrow(data\_table\_f(data, annotation)),

rqdatatable\_parallel =

nrow(ex\_data\_table\_parallel(optree, "key\_group", cl)),

rqdatatable = nrow(ex\_data\_table(optree)),

dplyr\_parallel =

nrow(dplyr\_parallel\_f(data, annotation)),

dplyr = nrow(dplyr\_pipeline(data, annotation)),

times = 10L)

saveRDS(timings, "Parallel\_rqdatatable\_timings.RDS")

**Conclusion**

print(timings)

## Unit: seconds

## expr min lq mean median uq

## data\_table\_parallel 5.274560 5.457105 5.609827 5.546554 5.686829

## data\_table 9.401677 9.496280 9.701807 9.541218 9.748159

## rqdatatable\_parallel 7.165216 7.497561 7.587663 7.563883 7.761987

## rqdatatable 12.490469 12.700474 13.320480 12.898154 14.229233

## dplyr\_parallel 6.492262 6.572062 6.784865 6.787277 6.875076

## dplyr 20.056555 20.450064 20.647073 20.564529 20.800350

## max neval

## 6.265888 10

## 10.419316 10

## 7.949404 10

## 14.282269 10

## 7.328223 10

## 21.332103 10

# autoplot(timings)

timings <- as.data.frame(timings)

timings$seconds <- timings$time/1e+9

ScatterBoxPlotH(timings,

xvar = "seconds", yvar = "expr",

title="task duration by method")

The benchmark timings show parallelized data.table is the fastest, followed by parallelized dplyr, and parallelized rqdatatable. In the non-paraellized case data.table is the fastest, followed by rqdatatable, and then dplyr.

A reason dplyr sees greater speedup relative to its own non-parallel implementation (yet does not beat data.table) is that data.table starts already multi-threaded, so data.table is exploiting some parallelism even before we added the process level parallelism (and hence sees less of a speed up, though it is fastest).

rquery pipelines [exhibit superior performance on big data systems](https://github.com/WinVector/rquery/blob/master/extras/PerfTest.md) (Spark, PostgreSQL, Amazon Redshift, and hopefully soon Google bigquery), and rqdatatable supplies [a very good in-memory implementation of the rquery system](http://www.win-vector.com/blog/2018/06/rqdatatable-rquery-powered-by-data-table/) based on data.table. rquery also speeds up solution development by supplying higher order operators and early debugging features.

In this note we have demonstrated simple procedures to reliably parallelize any of rqdatatable, data.table, or dplyr.

Note: we did not include alternatives such as multidplyr or dtplyr in the timings, as they did not appear to work on this example.