Pipelines in [R](https://www.r-project.org/) are popular, the most popular one being magrittr as used by dplyr.

**Piping**

Piping is not much more than having a system that lets one treat “x %.>% f(.)” as a near synonym for “f(x)”. For the wrapr dot arrow pipe the semantics are intentionally closer to (x %.>% f(.)) ~ {. <- x; f(.)}.

The pipe notation may be longer, but it avoids nesting and reversed right to left reading for many-stage operations (such as “x %.>% f1(.) %.>% f2(.) %.>% f3(.)” versus “f3(f2(f1(x)))”).

In addition to allowing users to write operations in this notation, most piping systems allow users to save pipelines for later re-use (though some others have issues serializing or saving such pipelines due to entanglement with the defining environment).

wrapr and rquery/rqdatatable supply a number of piping tools that are re-usable, serializable, and very powerful (via R S3 and S4 dispatch features). One of the most compelling features are “function objects” which mans objects can be treated like functions (applied to other objects by pipelines). We will discuss some of these features in the context of rquery/rqdatatable and wrapr.

**rquery/rqdatatable**

For quite a while the RQUERY R Package and RQDATATABLE package, have supplied a sequence of operators abstraction called an “operator tree” or “operator pipeline”.

These pipelines are (deliberately) fairly strict:

* They must start with a table description or definition.
* Each step must be a table to table transform meeting certain column pre-conditions.
* Each step must advertise what columns it makes available or produces, for later condition checking.

For a guiding example suppose we want to row-subset some data, get per-group means, and then sort the data by those means.

# our example data

d <- data.frame(

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

value = c( 1, 2, 2, -10),

stringsAsFactors = FALSE

)

# load our package

library("rqdatatable")

## Loading required package: rquery

# build an operator tree

threshold <- 0.0

ops <-

# define the data format

local\_td(d) %.>%

# restrict to rows with value >= threshold

select\_rows\_nse(.,

value >= threshold) %.>%

# compute per-group aggegations

project\_nse(.,

groupby = "group",

mean\_value = mean(value)) %.>%

# sort rows by mean\_value decreasing

orderby(.,

cols = "mean\_value",

reverse = "mean\_value")

# show the tree/pipeline

cat(format(ops))

## table(d;

## group,

## value) %.>%

## select\_rows(.,

## value >= 0) %.>%

## project(., mean\_value := mean(value),

## g= group) %.>%

## orderby(., desc(mean\_value))

Of course the purpose of such a pipeline is to be able to apply it to data. This is done simply with the wrapr dot arrow pipe:

d %.>% ops

## group mean\_value

## 1: b 2.0

## 2: a 1.5

rquery pipelines are designed to specify and execute data wrangling tasks. An important feature of rquery pipelines is: they are designed for serialization. This means we can save them and also send them to multiple nodes for parallel processing.

# save the optree

saveRDS(ops, "rquery\_optree.RDS")

# simulate a fresh R session

rm(list=setdiff(ls(), "d"))

library("rqdatatable")

# read the optree back in

ops <- readRDS('rquery\_optree.RDS')

# look at it

cat(format(ops))

## table(d;

## group,

## value) %.>%

## select\_rows(.,

## value >= 0) %.>%

## project(., mean\_value := mean(value),

## g= group) %.>%

## orderby(., desc(mean\_value))

# use it again

d %.>% ops

## group mean\_value

## 1: b 2.0

## 2: a 1.5

# clean up

rm(list=setdiff(ls(), "d"))

We can also run rqdatatable operations in “immediate mode”, without pre-defining the pipeline or tables:

threshold <- 0.0

d %.>%

select\_rows\_nse(.,

value >= threshold) %.>%

project\_nse(.,

groupby = "group",

mean\_value = mean(value)) %.>%

orderby(.,

cols = "mean\_value",

reverse = "mean\_value")

## group mean\_value

## 1: b 2.0

## 2: a 1.5

**wrapr function objects**

A natural question is: given we already have rquery pipelines why do we need wrapr function objects? The reason is: rquery/rdatatable pipelines are strict and deliberately restricted to operations that can be hosted both in R (via data.table) or on databases (examples: PostgreSQL and Spark). One might also want a more general pipeline with fewer constraints optimized for working in R directly.

The wrapr “function object” pipelines allow treatment of arbitrary objects as items we can pipe into. Their primary purpose is to convert arbitrary objects and functions into single-argument (or unary) functions. This converted form is perfect for pipelining. This, in a sense, lets us treat these objects as functions. The wrapr function object pipeline also has less constraint checking than rquery pipelines, so is more suitable for “black box” steps that do not publish their column use and production details (in fact wrapr function object pipelines work on arbitrary objects, not just data.frames or tables).

Let’s adapt our above example into a simple wrapr dot arrow pipeline.

library("wrapr")

threshold <- 0

d %.>%

.[.$value >= threshold, , drop = FALSE] %.>%

tapply(.$value, .$group, 'mean') %.>%

sort(., decreasing = TRUE)

## b a

## 2.0 1.5

All we have done is replace the rquery steps with typical base-R commands. As we see the wrapr dot arrow can route data through a sequence of such commands to repeat our example.

Now let’s adapt our above example into a re-usable wrapr function object pipeline.

library("wrapr")

threshold <- 0

pipeline <-

srcfn(

".[.$value >= threshold, , drop = FALSE]" ) %.>%

srcfn(

"tapply(.$value, .$group, 'mean')" ) %.>%

pkgfn(

"sort",

arg\_name = "x",

args = list(decreasing = TRUE))

cat(format(pipeline))

## UnaryFnList(

## SrcFunction{ .[.$value >= threshold, , drop = FALSE] }(.=., ),

## SrcFunction{ tapply(.$value, .$group, 'mean') }(.=., ),

## base::sort(x=., decreasing))

We used two wrapr abstractions to capture the steps for re-use (something built in to rquery, and now also supplied by wrapr). The abstractions are:

* SRCFN() which wraps arbitrary quoted code as a function object.
* PKGFN () which wraps a package qualified function name as a function object (“base” being the default package).

This sort of pipeline can be applied to data using pipe notation:

d %.>% pipeline

## b a

## 2.0 1.5

The above pipeline has one key inconvenience and one key weakness:

* For the srcfn() steps we had to place the source code in quotes, which defeats any sort of syntax highlighting and auto-completing in our R integrated development environment (IDE).
* The above pipeline has a reference to the value of threshold in our current environment, this means the pipeline is not sufficiently self-contained to serialize and share.

We can quickly address both of these issues with the wrapr :: ge() (“quote expression”) function. It uses base::substitute() to quote its arguments, and the IDE doesn’t know the contents are quoted and thus can help us with syntax highlighting and auto-completion. Also we are using base::bquote() .()-style escaping to bind in the value of threshold.

pipeline <-

srcfn(

qe( .[.$value >= .(threshold), , drop = FALSE] )) %.>%

srcfn(

qe( tapply(.$value, .$group, 'mean') )) %.>%

pkgfn(

"sort",

arg\_name = "x",

args = list(decreasing = TRUE))

cat(format(pipeline))

## UnaryFnList(

## SrcFunction{ .[.$value >= 0, , drop = FALSE] }(.=., ),

## SrcFunction{ tapply(.$value, .$group, "mean") }(.=., ),

## base::sort(x=., decreasing))

d %.>% pipeline

## b a

## 2.0 1.5

Notice this pipeline works as before, but no longer refers to the external value threshold. This pipeline can be saved and shared.

Another recommended way to bind in values is with the args-argument, which is a named list of values that are expected to be available with a srcfn() is evaluated, or additional named arguments that will be applied to a pkgfn().

In this notation the pipeline is written as follows.

pipeline <-

srcfn(

qe( .[.$value >= threshold, , drop = FALSE] ),

args = list('threshold' = threshold)) %.>%

srcfn(

qe( tapply(.$value, .$group, 'mean') )) %.>%

pkgfn(

"sort",

arg\_name = "x",

args = list(decreasing = TRUE))

cat(format(pipeline))

## UnaryFnList(

## SrcFunction{ .[.$value >= threshold, , drop = FALSE] }(.=., threshold),

## SrcFunction{ tapply(.$value, .$group, "mean") }(.=., ),

## base::sort(x=., decreasing))

d %.>% pipeline

## b a

## 2.0 1.5

We can save this pipeline.

saveRDS(pipeline, "wrapr\_pipeline.RDS")

And simulate using it in a fresh environment (i.e. simulate sharing it).

# simulate a fresh environment

rm(list = setdiff(ls(), "d"))

library("wrapr")

pipeline <- readRDS('wrapr\_pipeline.RDS')

cat(format(pipeline))

## UnaryFnList(

## SrcFunction{ .[.$value >= threshold, , drop = FALSE] }(.=., threshold),

## SrcFunction{ tapply(.$value, .$group, "mean") }(.=., ),

## base::sort(x=., decreasing))

d %.>% pipeline

## b a

## 2.0 1.5

**Conclusion**

And that is some of the power of wrapr piping, rquery/rqdatatable, and wrapr function objects. Essentially wrapr function objects are a reference application of the S3/S4 piping abilities

The technique is very convenient when each of the steps is a substantial (such as non-trivial data preparation and model application steps).

The above techniques can make reproducing and sharing methods much easier.

# clean up after example

unlink("rquery\_optree.RDS")

unlink("wrapr\_pipeline.RDS")