**Introduction**

[rquery](https://github.com/WinVector/rquery) is a data wrangling system designed to express complex data manipulation as a series of simple data transforms. This is in the spirit of R’s base::transform(), or dplyr’s dplyr::mutate() and uses a pipe in the style popularized in R with magrittr. The operators themselves follow the selections in Codd’s relational algebra, with the addition of the traditional SQL “window functions.”

In transform formulations data manipulation is written as transformations that produce new data.frames, instead of as alterations of a primary data structure (as is the case with data.table). Transform system *can* use more space and time than in-place methods. However, in our opinion, transform systems have a number of pedagogical advantages.

In rquery’s case the primary set of data operators is as follows:

* drop\_columns
* select\_columns
* rename\_columns
* select\_rows
* order\_rows
* extend
* project
* natural\_join
* convert\_records (supplied by the cdata package).

These operations break into a small number of themes:

* Simple column operations (selecting and re-naming columns).
* Simple row operations (selecting and re-ordering rows).
* Creating new columns or replacing columns with new calculated values.
* Aggregating or summarizing data.
* Combining results between two data.frames.
* General conversion of record layouts (supplied by the cdata package).

The point is: Codd worked out that a great number of data transformations can be decomposed into a small number of the above steps. rquery supplies a high performance implementation of these methods that scales from in-memory scale up through big data scale (to just about anything that supplies a sufficiently powerful SQL interface, such as PostgreSQL, Apache Spark, or Google BigQuery).

We will work through simple examples/demonstrations of the rquery data manipulation operators.

**rquery operators**

**Simple column operations (selecting and re-naming columns)**

The simple column operations are as follows.

* drop\_columns
* select\_columns
* rename\_columns

These operations are easy to demonstrate.

We set up some simple data.

d <- data.frame(

x = c(1, 1, 2),

y = c(5, 4, 3),

z = c(6, 7, 8)

)

knitr::kable(d)

| **x** | **y** | **z** |
| --- | --- | --- |
| 1 | 5 | 6 |
| 1 | 4 | 7 |
| 2 | 3 | 8 |

For example: drop\_columns works as follows. drop\_columns creates a new data.frame without certain columns.

library(rquery)

drop\_columns(d, c('y', 'z'))

## x

## 1: 1

## 2: 1

## 3: 2

In all cases the first argument of a rquery operator is either the data to be processed, or an earlier rquery pipeline to be extended. We will take about composing rquery operations after we work through examples of all of the basic operations.

We can write the above in piped notation (using the [wrapr pipe](https://journal.r-project.org/archive/2018/RJ-2018-042/index.html) in this case):

d %.>%

drop\_columns(., c('y', 'z')) %.>%

knitr::kable(.)

| **x** |
| --- |
| 1 |
| 1 |
| 2 |

Notice the first argument is an explicit “dot” in [wrapr pipe notation](https://journal.r-project.org/archive/2018/RJ-2018-042/index.html).

select\_columns’s action is also obvious from example.

d %.>%

select\_columns(., c('x', 'y')) %.>%

knitr::kable(.)

| **x** | **y** |
| --- | --- |
| 1 | 5 |
| 1 | 4 |
| 2 | 3 |

**Simple row operations (selecting and re-ordering rows)**

The simple row operations are:

* select\_rows
* order\_rows

select\_rows keeps the set of rows that meet a given predicate expression.

d %.>%

select\_rows(., x == 1) %.>%

knitr::kable(.)

| **x** | **y** | **z** |
| --- | --- | --- |
| 1 | 5 | 6 |
| 1 | 4 | 7 |

order\_rows re-orders rows by a selection of column names (and allows reverse ordering by naming which columns to reverse in the optional reverse argument). Multiple columns can be selected in the order, each column breaking ties in the earlier comparisons.

d %.>%

order\_rows(.,

c('x', 'y'),

reverse = 'x') %.>%

knitr::kable(.)

| **x** | **y** | **z** |
| --- | --- | --- |
| 2 | 3 | 8 |
| 1 | 4 | 7 |
| 1 | 5 | 6 |

General rquery operations do not depend on row-order and are not guaranteed to preserve row-order, so if you do want to order rows you should make it the last step of your pipeline.

**Creating new columns or replacing columns with new calculated values**

The important create or replace column operation is:

* extend

extend accepts arbitrary expressions to create new columns (or replace existing ones). For example:

d %.>%

extend(., zzz := y / x) %.>%

knitr::kable(.)

| **x** | **y** | **z** | **zzz** |
| --- | --- | --- | --- |
| 1 | 5 | 6 | 5.0 |
| 1 | 4 | 7 | 4.0 |
| 2 | 3 | 8 | 1.5 |

We can use = or := for column assignment. In these examples we will use := to keep column assignment clearly distinguishable from argument binding.

extend allows for very powerful per-group operations akin to what [SQL](https://en.wikipedia.org/wiki/SQL) calls [“window functions”](https://en.wikipedia.org/wiki/SQL_window_function). When the optional partitionby argument is set to a vector of column names then aggregate calculations can be performed per-group. For example.

shift <- data.table::shift

d %.>%

extend(.,

max\_y := max(y),

shift\_z := shift(z),

row\_number := row\_number(),

cumsum\_z := cumsum(z),

partitionby = 'x',

orderby = c('y', 'z')) %.>%

knitr::kable(.)

| **x** | **y** | **z** | **max\_y** | **shift\_z** | **row\_number** | **cumsum\_z** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 4 | 7 | 5 | NA | 1 | 7 |
| 1 | 5 | 6 | 5 | 7 | 2 | 13 |
| 2 | 3 | 8 | 3 | NA | 1 | 8 |

Notice the aggregates were performed per-partition (a set of rows with matching partition key values, specified by partitionby) and in the order determined by the orderby argument (without the orderby argument order is not guaranteed, so always set orderby for windowed operations that depend on row order!).

**Aggregating or summarizing data**

The main aggregation method for rquery is:

* project

project performs per-group calculations, and returns only the grouping columns (specified by groupby) and derived aggregates. For example:

d %.>%

project(.,

max\_y := max(y),

count := n(),

groupby = 'x') %.>%

knitr::kable(.)

| **x** | **max\_y** | **count** |
| --- | --- | --- |
| 1 | 5 | 2 |
| 2 | 3 | 1 |

Notice we only get one row for each unique combination of the grouping variables. We can also aggregate into a single row by not specifying any groupby columns.

d %.>%

project(.,

max\_y := max(y),

count := n()) %.>%

knitr::kable(.)

| **max\_y** | **count** |
| --- | --- |
| 5 | 3 |

**Combining results between two data.frames**

To combine multiple tables in rquery one uses what we call the natural\_join operator. In the rquery natural\_join, rows are matched by column keys and any two columns with the same name are *coalesced* (meaning the first table with a non-missing values supplies the answer). This is easiest to demonstrate with an example.

Let’s set up new example tables.

d\_left <- data.frame(

k = c('a', 'a', 'b'),

x = c(1, NA, 3),

y = c(1, NA, NA),

stringsAsFactors = FALSE

)

knitr::kable(d\_left)

| **k** | **x** | **y** |
| --- | --- | --- |
| a | 1 | 1 |
| a | NA | NA |
| b | 3 | NA |

d\_right <- data.frame(

k = c('a', 'b', 'q'),

y = c(10, 20, 30),

stringsAsFactors = FALSE

)

knitr::kable(d\_right)

| **k** | **y** |
| --- | --- |
| a | 10 |
| b | 20 |
| q | 30 |

To perform a join we specify which set of columns our our row-matching conditions (using the by argument) and what type of join we want (using the jointype argument). For example we can use jointype = 'LEFT' to augment our d\_left table with additional values from d\_right.

natural\_join(d\_left, d\_right,

by = 'k',

jointype = 'LEFT') %.>%

knitr::kable(.)

| **k** | **x** | **y** |
| --- | --- | --- |
| a | 1 | 1 |
| a | NA | 10 |
| b | 3 | 20 |

In a left-join (as above) if the right-table has unique keys then we get a table with the same structure as the left-table- but with more information per row. This is a very useful type of join in data science projects. Notice columns with matching names are coalesced into each other, which we interpret as “take the value from the left table, unless it is missing.”

**General conversion of record layouts**

**Composing operations**

We could, of course, perform complicated data manipulation by sequencing rquery operations. For example to select one row with minimal y per-x group we could work in steps as follows.

. <- d

. <- extend(.,

row\_number := row\_number(),

partitionby = 'x',

orderby = c('y', 'z'))

. <- select\_rows(.,

row\_number == 1)

. <- drop\_columns(.,

"row\_number")

knitr::kable(.)

| **x** | **y** | **z** |
| --- | --- | --- |
| 1 | 4 | 7 |
| 2 | 3 | 8 |

The above discipline has the advantage that it is easy to debug, as we can run line by line and inspect intermediate values.

d ->.;

extend(.,

row\_number := row\_number(),

partitionby = 'x',

orderby = c('y', 'z')) ->.;

select\_rows(.,

row\_number == 1) ->.;

drop\_columns(.,

"row\_number") ->.;

knitr::kable(.)

| **x** | **y** | **z** |
| --- | --- | --- |
| 1 | 4 | 7 |
| 2 | 3 | 8 |

Or we can use the [wrapr pipe](https://journal.r-project.org/archive/2018/RJ-2018-042/index.html) on the data, which we call “immediate mode”

d %.>%

extend(.,

row\_number := row\_number(),

partitionby = 'x',

orderby = c('y', 'z')) %.>%

select\_rows(.,

row\_number == 1) %.>%

drop\_columns(.,

"row\_number") %.>%

knitr::kable(.)

| **x** | **y** | **z** |
| --- | --- | --- |
| 1 | 4 | 7 |
| 2 | 3 | 8 |

rquery operators can also act on rquery pipelines instead of acting on data. We can write our operations as follows:

ops <- local\_td(d) %.>%

extend(.,

row\_number := row\_number(),

partitionby = 'x',

orderby = c('y', 'z')) %.>%

select\_rows(.,

row\_number == 1) %.>%

drop\_columns(.,

"row\_number")

cat(format(ops))

## mk\_td("d", c(

## "x",

## "y",

## "z")) %.>%

## extend(.,

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

## partitionby = c('x'),

## orderby = c('y', 'z'),

## reverse = c()) %.>%

## select\_rows(.,

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

## drop\_columns(.,

## c('row\_number'))

And we can re-use this pipeline, both on local data and to generate SQL to be run in remote databases. Applying this operator pipeline to our data.frame d is performed as follows.

d %.>%

ops %.>%

knitr::kable(.)

| **x** | **y** | **z** |
| --- | --- | --- |
| 1 | 4 | 7 |
| 2 | 3 | 8 |

What we are trying to illustrate above: there is a continuum of notations possible between:

* Working over values with explicit intermediate variables.
* Working over values with a pipeline.
* Working over operators with a pipeline.

**Conclusion**

rquery supplies a very teachable grammar of data manipulation based on Codd’s relational algebra and experience with pipelined data transforms (such as base::transform(), dplyr, and data.table).

For in-memory situations rquery uses data.table as the implementation provider (through the small adapter package rqdatatable) and is routinely faster than any other R data manipulation system *except* data.table itself.

For bigger than memory situations rquery can translate to any sufficiently powerful SQL dialect, allowing rquery pipelines to be executed on PostgreSQL, Apache Spark, or Google BigQuery.

In addition the data\_algebra Python package supplies a nearly identical system for working with data in Python.