Big Data Transforms

John Mount, Win-Vector LLC

11/11/2017

As part of our consulting practice Win-Vector LLC has been helping a few clients stand-up advanced analytics and machine learning stacks using `R` and substantial data stores (such as relational database variants such as `PostgreSQL` or big data systems such as `Spark`).

Often we come to a point where we or a partner realize: "the design would be a whole lot easier if we could phrase it in terms of higher order data operators."

The `R` package `DBI` gives us direct access to `SQL` and the package `dplyr` gives us access to a transform grammar that can either be executed or translated into `SQL`.

But, as we point out in the `replyr` `README`: moving from in-memory `R` to large data systems is always a bit of a shock as you lose a lot of your higher order data operators or transformations. Missing operators include:

- union (binding by rows many data frames into a single data frame).
- split (splitting a single data frame into many data frames).
- pivot (moving row values into columns).
- un-pivot (moving column values to rows).

I can repeat this. If you are an `R` user used to using one of `dplyr::bind_rows()`, `base::split()`, `tidyr::spread()`, or `tidyr::gather()`: you will find these functions do not work on remote data sources, but have replacement implementations in the `replyr` and `cdata` packages.

For example:

```
library("RPostgreSQL")
```

```
## Loading required package: DBI
```

```
suppressPackageStartupMessages(library("dplyr"))
isSpark <- FALSE

# # Can work with PostgreSQL
# my_db <- DBI::dbConnect(dbDriver("PostgreSQL"),
#                         host = 'localhost',
#                         port = 5432,
#                         user = 'postgres',
#                         password = 'pg')

# Can work with Sparklyr
my_db <-  sparklyr::spark_connect(version='2.2.0',
                                  master = "local")
```

```
## Warning in yaml.load(readLines(con),
## error.label = error.label, ...): R
## expressions in yaml.load will not be auto-
## evaluated by default in the near future

## Warning in yaml.load(readLines(con),
## error.label = error.label, ...): R
## expressions in yaml.load will not be auto-
## evaluated by default in the near future

## Warning in yaml.load(readLines(con),
## error.label = error.label, ...): R
## expressions in yaml.load will not be auto-
## evaluated by default in the near future
```

```
isSpark <- TRUE
```

```
d <- dplyr::copy_to(my_db, data.frame(x =  c(1,5),
                                      group = c('g1', 'g2'),
                                      stringsAsFactors = FALSE),
                    'd')
knitr::kable(d)
```

x	group
1	g1
5	g2

```
# show dplyr::bind_rows() fails.
dplyr::bind_rows(list(d, d))
```

```
## Error in bind_rows_(x, .id): Argument 1 must be a data frame or a named atomic vector, not a tbl_spa
```

The `replyr` and `cdata` packages supply `R` accessible implementations of these missing operators for large data systems such as `PostgreSQL` and `Spark`.

For example:

```
# using the development version of replyr https://github.com/WinVector/replyr
library("replyr")
```

```
## Loading required package: seplyr
```

```
## Loading required package: wrapr
```

```
## Loading required package: cdata
```

```
packageVersion("replyr")
```

```
## [1] '0.9.1'
```

```
# binding rows
dB <- replyr_bind_rows(list(d, d))
knitr::kable(dB)
```

x	group
1	g1
5	g2
1	g1
5	g2

```
# splitting frames
replyr_split(dB, 'group')
```

```
## $g2
## # Source:
## #   table<replyr_gapply_ju4f1zuvur2l0g2ryikf_0000000001>
## #   [?? x 2]
## # Database: spark_connection
##       x group
##   <dbl> <chr>
## 1  5.00 g2
## 2  5.00 g2
##
## $g1
## # Source:
## #   table<replyr_gapply_ju4f1zuvur2l0g2ryikf_0000000003>
## #   [?? x 2]
## # Database: spark_connection
##       x group
##   <dbl> <chr>
## 1  1.00 g1
## 2  1.00 g1
```

```
# pivoting
pivotControl <-
  cdata::build_pivot_control_q('d',
                               columnToTakeKeysFrom = 'group',
                               columnToTakeValuesFrom = 'x',
                               sep = '_',
                               my_db = my_db)
```

```
dWname <-
  cdata::blocks_to_rowrecs_q(keyColumns = NULL,
                             controlTable = pivotControl,
                             tallTable = 'd',
                             my_db = my_db, strict = FALSE)
dW <- dplyr::tbl(my_db, dWname)
knitr::kable(dW)
```

group_g2	group_g1
5	1

```
# un-pivoting
unpivotControl <-
  cdata::build_unpivot_control(nameForNewKeyColumn = 'group',
                               nameForNewValueColumn = 'x',
                               columnsToTakeFrom = colnames(dW))
dXname <-
  cdata::rowrecs_to_blocks_q(controlTable = unpivotControl,
                             wideTable = dWname,
                             my_db = my_db)
dX <- dplyr::tbl(my_db, dXname)
knitr::kable(dX)
```

group	x
group_g2	5
group_g1	1

The point is: using the `replyr` and `cdata` packages you *can* design in terms of higher-order data transforms, even when working with big data in `R`. Designs in terms of these operators tend to be succinct, powerful, performant, and maintainable.

To master the terms `rowrecs_to_blocks` and `blocks_to_rowrecs` I suggest trying the following two articles:

- Theory of coordinatized data.
- Fluid data transforms.

```
if(isSpark) {
  status <- sparklyr::spark_disconnect(my_db)
} else {
  status <- DBI::dbDisconnect(my_db)
}
my_db <- NULL
```