Fluid Data

John Mount, Win-Vector LLC

November 11, 2017

Introduction

The `cdata` R package provides a powerful extension of the "fluid data" (or "coordinatized data") concept (please see here for some notes) that goes way beyond the concepts of pivot/un-pivot.

The fluid data concept is:

1) Data cells have coordinates, and the dependence of these coordinates on a given data representation (a table or map) is an inessential detail to be abstracted out.
2) There may not be one "preferred" shape (or frame of reference) for data: you have to anticipate changing data shape many times to adapt to the tasks and tools (data relativity).

`cdata` supplies two general operators for fluid data work at database scale (and `Spark` big data scale):

1) `moveValuesToRows*()`: operators centered around `SQL cross-join` semantics. un-pivot, `tidyr::gather()`, and `cdata::unpivotValuesToRows()` are special cases of this general operator.
2) `moveValuesToColumns*()`: an operator centered around `SQL group by` semantics. pivot, `tidyr::spread()`, `cdata::pivotValuesToColumns()`, transpose, and one-hot-encoding are special cases of this general operator.

Because these operators are powerful, they are fairly general, and at first hard to mentally model (especially if you insist on think of them in only in terms of more a specialized operator such as pivot, instead of more general relational concepts such as "cross join" and "group by"). These operators are thin wrappers populating and enforcing a few invariants over a large `SQL` statement. That does not mean that these operators are trivial, they are thin because `SQL` is powerful and we have a good abstraction.

Due to the very detailed and explicit controls used in these operators- they are very comprehensible once studied. We will follow-up later with additional training material to make quicker comprehension available to more readers. This document is limiting itself to being a mere concise statement of and demonstration of the operators.

Data coordinate notation theory

We are going to introduce a explicit, dense, and powerful data coordinate notation.

Consider the following table that we call a "control table":

```
suppressPackageStartupMessages(library("cdata"))
packageVersion("cdata")
```

```
## [1] '0.5.1'
```

```
suppressPackageStartupMessages(library("dplyr"))
options(width = 160)
tng <- cdata::makeTempNameGenerator('fdexample')

controlTable <- dplyr::tribble(~group, ~col1, ~col2,
                               'aa',  'c1',  'c2',
                               'bb',  'c3',  'c4')
knitr::kable(controlTable)
```

group	col1	col2
aa	c1	c2
bb	c3	c4

Control tables partially specify a change of data shape or change of data cell coordinates.

The specification is interpreted as follows:

The region `controlTable[ , 2:ncol(controlTable)]` specifies partial coordinates of data cells in another table. In our example these partial coordinates are "`c1`", "`c2`", "`c3`", and "`c4`" treated as column names. For example if our data is:

```
dat1 <- dplyr::tribble(
  ~ID,          ~c1,          ~c2,          ~c3,          ~c4,
  'id1', 'val_id1_c1', 'val_id1_c2', 'val_id1_c3', 'val_id1_c4',
  'id2', 'val_id2_c1', 'val_id2_c2', 'val_id2_c3', 'val_id2_c4',
  'id3', 'val_id3_c1', 'val_id3_c2', 'val_id3_c3', 'val_id3_c4' )
knitr::kable(dat1)
```

ID	c1	c2	c3	c4
id1	val_id1_c1	val_id1_c2	val_id1_c3	val_id1_c4
id2	val_id2_c1	val_id2_c2	val_id2_c3	val_id2_c4
id3	val_id3_c1	val_id3_c2	val_id3_c3	val_id3_c4

Then each data cell in `dat1` (excluding the key-columns, in this case "`ID`") is named by the row-id (stored in the `ID` column) plus the column-name ("`c1`", "`c2`", "`c3`", and "`c4`"). Knowing `ID` plus the column name unique identifies the data-caring cell in table `dat1`.

However, there is an alternate cell naming available from the `controlTable` notation. Each name in the region `controlTable[ , 2:ncol(controlTable)]` is itself uniquely named by the `group` entry and column name of the control table itself. This means we have the following correspondence from the partial names "`c1`", "`c2`", "`c3`", and "`c4`" to a new set of partial names:

```
namePairings <- expand.grid(seq_len(nrow(controlTable)),
                     2:ncol(controlTable))
colnames(namePairings) <- c("controlI", "controlJ")
namePairings$coords_style1 <-
  vapply(seq_len(nrow(namePairings)),
         function(ii) {
           as.character(paste("column:",
                              controlTable[namePairings$controlI[[ii]],
                                           namePairings$controlJ[[ii]]]))
         },
         character(1))
namePairings$coords_style2 <-
  vapply(seq_len(nrow(namePairings)),
         function(ii) {
           paste("group:",
                 controlTable$group[[namePairings$controlI[[ii]]]],
                 ", column:",
                 colnames(controlTable)[[namePairings$controlJ[[ii]]]])
         },
         character(1))
as.matrix(namePairings[ , c("coords_style1", "coords_style2")])
```

```
##      coords_style1 coords_style2
## [1,] "column: c1"  "group: aa , column: col1"
## [2,] "column: c3"  "group: bb , column: col1"
## [3,] "column: c2"  "group: aa , column: col2"
## [4,] "column: c4"  "group: bb , column: col2"
```

The idea is the control table is a very succinct description of the pairing of the `namePairings$coords_style1` cell partial coordinates and the `namePairings$coords_style2` cell partial coordinates. As we have said the `namePairings$coords_style1` cell partial coordinates become full cell coordinates for the data cells in `dat1` when combined with the `dat1 ID` column. The `namePairings$coords_style2` are part of a natural naming for the data cells in the following table:

```
my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
dat1db <- dplyr::copy_to(my_db, dat1, 'dat1db')
dat2 <- cdata::moveValuesToRowsN(wideTable = 'dat1db',
                                 controlTable = controlTable,
                                 my_db = my_db,
                                 columnsToCopy = "ID",
                                 tempNameGenerator = tng) %>%
  dplyr::tbl(my_db, .) %>%
  arrange(ID, group)
knitr::kable(dat2)
```

ID	group	col1	col2
id1	aa	val_id1_c1	val_id1_c2
id1	bb	val_id1_c3	val_id1_c4
id2	aa	val_id2_c1	val_id2_c2
id2	bb	val_id2_c3	val_id2_c4
id3	aa	val_id3_c1	val_id3_c2
id3	bb	val_id3_c3	val_id3_c4

For `dat2` the composite row-key `(ID, group)` plus the column name (one of `col1` or `col2`) gives us the positions of the data carrying cells.

So essentially the two readings of `controlTable` are a succinct representation of the explicit pairing of data cell coordinates shown in the `namePairings` table.

The Operators

In terms of the above notation/theory our two operators `moveValuesToRows*()` and `moveValuesToColumns*()` are (in principle) easy to describe:

- `moveValuesToRows*()` reshapes data from style 1 to style 2
- `moveValuesToColumns*()` reshapes data from style 2 to style 1.

The above is certainly succinct, but carries a lot of information and allows for a lot of different possible applications. Many important applications are derived from how these two operators interact with row-operations and column-operations.

We give simple examples of each of the operators below.

moveValuesToRows()*

```
wideTableName <- 'dat'
d <- dplyr::copy_to(my_db,
```

```
      dplyr::tribble(
        ~ID,          ~c1,          ~c2,          ~c3,          ~c4,
      'id1', 'val_id1_c1', 'val_id1_c2', 'val_id1_c3', 'val_id1_c4',
      'id2', 'val_id2_c1', 'val_id2_c2', 'val_id2_c3', 'val_id2_c4',
      'id3', 'val_id3_c1', 'val_id3_c2', 'val_id3_c3', 'val_id3_c4' ),
            wideTableName, overwrite = TRUE, temporary=TRUE)
controlTable <- dplyr::tribble(~group, ~col1, ~col2,
                               'aa',  'c1',  'c2',
                               'bb',  'c3',  'c4')
columnsToCopy <- 'ID'
cdata::moveValuesToRowsN(wideTable = wideTableName,
                         controlTable =  controlTable,
                         my_db = my_db,
                         columnsToCopy = columnsToCopy,
                         tempNameGenerator = tng) %>%
  dplyr::tbl(my_db, .) %>%
  arrange(ID, group) %>%
  knitr::kable()
```

ID	group	col1	col2
id1	aa	val_id1_c1	val_id1_c2
id1	bb	val_id1_c3	val_id1_c4
id2	aa	val_id2_c1	val_id2_c2
id2	bb	val_id2_c3	val_id2_c4
id3	aa	val_id3_c1	val_id3_c2
id3	bb	val_id3_c3	val_id3_c4

moveValuesToColumns()*

```
tallTableName <- 'dat'
d <- dplyr::copy_to(my_db,
  dplyr::tribble(
   ~ID,   ~group, ~col1,              ~col2,
   "id1", "aa",   "val_id1_gaa_col1", "val_id1_gaa_col2",
   "id1", "bb",   "val_id1_gbb_col1", "val_id1_gbb_col2",
   "id2", "aa",   "val_id2_gaa_col1", "val_id2_gaa_col2",
   "id2", "bb",   "val_id2_gbb_col1", "val_id2_gbb_col2",
   "id3", "aa",   "val_id3_gaa_col1", "val_id3_gaa_col2",
   "id3", "bb",   "val_id3_gbb_col1", "val_id3_gbb_col2" ),
        tallTableName,
        overwrite = TRUE, temporary=TRUE)
controlTable <- dplyr::tribble(~group, ~col1, ~col2,
                               'aa',  'c1',  'c2',
```

```
                                'bb',  'c3',  'c4')
keyColumns <- 'ID'
cdata::moveValuesToColumnsN(tallTable = tallTableName,
                            controlTable = controlTable,
                            keyColumns = keyColumns,
                            my_db = my_db,
                            tempNameGenerator = tng) %>%
  dplyr::tbl(my_db, .) %>%
  arrange(ID) %>%
  knitr::kable()
```

ID	c1	c2	c3	c4
id1	val_id1_gaa_col1	val_id1_gaa_col2	val_id1_gbb_col1	val_id1_gbb_col2
id2	val_id2_gaa_col1	val_id2_gaa_col2	val_id2_gbb_col1	val_id2_gbb_col2
id3	val_id3_gaa_col1	val_id3_gaa_col2	val_id3_gbb_col1	val_id3_gbb_col2

Pivot/Un-Pivot

Pivot and un-pivot (or `tidyr::spread()` and `tidyr::gather()`) are special cases of the `moveValuesToColumns*()` and `moveValuesToRows*()` operators. Pivot/un-pivot are the cases where the control table has two columns.

Pivot

```
d <- data.frame(
  index = c(1, 2, 3, 1, 2, 3),
  meastype = c('meas1','meas1','meas1','meas2','meas2','meas2'),
  meas = c('m1_1', 'm1_2', 'm1_3', 'm2_1', 'm2_2', 'm2_3'),
  stringsAsFactors = FALSE)
knitr::kable(d)
```

index	meastype	meas
1	meas1	m1_1
2	meas1	m1_2
3	meas1	m1_3
1	meas2	m2_1
2	meas2	m2_2
3	meas2	m2_3

```
# the cdata::pivotValuesToColumns version
# equivalent to tidyr::spread(d, 'meastype', 'meas')
```

```
cdata::pivotValuesToColumns(d,
                            columnToTakeKeysFrom = 'meastype',
                            columnToTakeValuesFrom = 'meas',
                            rowKeyColumns= 'index',
                            sep= '_') %>%
  arrange(index) %>%
  knitr::kable()
```

index	meastype_meas1	meastype_meas2
1	m1_1	m2_1
2	m1_2	m2_2
3	m1_3	m2_3

```
# the cdata::moveValuesToColumns*() version
dtall <- dplyr::copy_to(my_db, d, "dtall")
controlTable <- cdata::buildPivotControlTableN(tableName = "dtall",
                                               columnToTakeKeysFrom = 'meastype',
                                               columnToTakeValuesFrom = 'meas',
                                               my_db = my_db,
                                               sep = "_")
knitr::kable(controlTable)
```

meastype	meas
meas1	meastype_meas1
meas2	meastype_meas2

```
moveValuesToColumnsN(tallTable = "dtall",
                     controlTable = controlTable,
                     keyColumns = "index",
                     my_db = my_db,
                     tempNameGenerator = tng) %>%
  dplyr::tbl(my_db, .) %>%
  arrange(index) %>%
  knitr::kable()
```

index	meastype_meas1	meastype_meas2
1	m1_1	m2_1
2	m1_2	m2_2
3	m1_3	m2_3

Un-Pivot

```
d <- data.frame(
  index = c(1, 2, 3),
  info = c('a', 'b', 'c'),
  meas1 = c('m1_1', 'm1_2', 'm1_3'),
  meas2 = c('2.1', '2.2', '2.3'),
  stringsAsFactors = FALSE)
knitr::kable(d)
```

index	info	meas1	meas2
1	a	m1_1	2.1
2	b	m1_2	2.2
3	c	m1_3	2.3

```
# the cdata::unpivotValuesToRows() version
# equivalent to tidyr::gather(d, 'meastype', 'meas', c('meas1','meas2'))
cdata::unpivotValuesToRows(d,
                           nameForNewKeyColumn= 'meastype',
                           nameForNewValueColumn= 'meas',
                           columnsToTakeFrom= c('meas1','meas2')) %>%
  arrange(index, info) %>%
  knitr::kable()
```

index	info	meastype	meas
1	a	meas1	m1_1
1	a	meas2	2.1
2	b	meas1	m1_2
2	b	meas2	2.2
3	c	meas1	m1_3
3	c	meas2	2.3

```
# the cdata::cdata::unpivotValuesToRows() version
dwide <- dplyr::copy_to(my_db, d, "dwide")
controlTable <- buildUnPivotControlTable(nameForNewKeyColumn= 'meastype',
                                         nameForNewValueColumn= 'meas',
                                         columnsToTakeFrom= c('meas1','meas2'))
knitr::kable(controlTable)
```

meastype	meas
meas1	meas1

meastype	meas
meas2	meas2

```
keyColumns = c('index', 'info')
moveValuesToRowsN(wideTable = "dwide",
                  controlTable = controlTable,
                  my_db = my_db,
                  columnsToCopy = keyColumns,
                  tempNameGenerator = tng) %>%
  dplyr::tbl(my_db, .) %>%
  arrange(index, info) %>%
  knitr::kable()
```

index	info	meastype	meas
1	a	meas1	m1_1
1	a	meas2	2.1
2	b	meas1	m1_2
2	b	meas2	2.2
3	c	meas1	m1_3
3	c	meas2	2.3

Additional Interesting Applications

Interesting applications of `cdata::moveValuesToRows*()` and `cdata::moveValuesToColumns*()` include situations where `tidyr` is not available (databases and `Spark`) and also when the data transformation is not obviously a single pivot or un-pivot.

Row-parallel dispatch

A particularly interesting application is converting many column operations into a single operation using a row-parallel dispatch.

Suppose we had the following data in the following format in our system of record (but with many more column groups and columns):

```
purchaseDat <- dplyr::copy_to(my_db, dplyr::tribble(
  ~ID, ~Q1purchases, ~Q2purchases, ~Q1rebates, ~Q2rebates,
    1,           20,           10,          5,          3,
    2,            5,            6,         10,         12),
  'purchaseDat')
knitr::kable(purchaseDat)
```

ID	Q1purchases	Q2purchases	Q1rebates	Q2rebates
1	20	10	5	3
2	5	6	10	12

Common tasks might include totaling columns and computing rates between columns. However, sometimes that is best done in a row-oriented representation (though outside systems may need column oriented, or more denormalized results).

With fluid data the task is easy:

```
controlTable <- dplyr::tribble(
  ~group, ~purchases,    ~rebates,
  "Q1",   "Q1purchases", "Q1rebates",
  "Q2",   "Q2purchases", "Q2rebates")
knitr::kable(controlTable)
```

group	purchases	rebates
Q1	Q1purchases	Q1rebates
Q2	Q2purchases	Q2rebates

```
purchasesTall <- moveValuesToRowsN(wideTable = "purchaseDat",
                                   columnsToCopy = "ID",
                                   controlTable = controlTable,
                                   my_db = my_db,
                                   tempNameGenerator = tng) %>%
  dplyr::tbl(my_db, .)
knitr::kable(purchasesTall)
```

ID	group	purchases	rebates
1	Q1	20	5
1	Q2	10	3
2	Q1	5	10
2	Q2	6	12

```
# perform the calculation in one easy step
calc <- purchasesTall %>%
  mutate(purchasesPerRebate = purchases/rebates) %>%
  compute(name = "purchasesTallC")
knitr::kable(calc)
```

ID	group	purchases	rebates	purchasesPerRebate
1	Q1	20	5	4.000000
1	Q2	10	3	3.333333
2	Q1	5	10	0.500000
2	Q2	6	12	0.500000

```
# move what we want back
controlTable <- controlTable %>%
  mutate(purchasesPerRebate =
           paste0(group, "purchasesPerRebate"))
knitr::kable(controlTable)
```

group	purchases	rebates	purchasesPerRebate
Q1	Q1purchases	Q1rebates	Q1purchasesPerRebate
Q2	Q2purchases	Q2rebates	Q2purchasesPerRebate

```
# notice the step back is not a single
# pivot or un-pivot
# due to the larger controlTable
# (especially if there were more quarters)
result <- moveValuesToColumnsN(tallTable = "purchasesTallC",
                               keyColumns = "ID",
                               controlTable = controlTable,
                               my_db = my_db,
                               tempNameGenerator = tng) %>%
  dplyr::tbl(my_db, .)
knitr::kable(result)
```

ID	Q1purchases	Q1rebates	Q1purchasesPerRebate	Q2purchases	Q2rebates	Q2purchasesPerRebate
1	20	5	4.0	10	3	3.333333
2	5	10	0.5	6	12	0.500000

The point is: the above can work on a large number of rows and columns (especially on a system such as `Spark` where row operations are performed in parallel).

The above work pattern is particularly powerful on big data systems when the tall table is built up in pieces by appending data (so only the pivot style step is required).

One-hot encoding

Adding indicators or dummy variables (by one-hot encoding, or other methods) are essentially special cases of the pivot flavor of `cdata::moveValuesToColumns*()`.

Transpose

Transpose is a special case of these operators. In fact the key-columns behave like group specifiers, meaning we can transpose many similarly structured tables at once.

group_by/aggregate

Many operations that look like a complicated pivot in column format are in fact a simple row operation followed a group_by/aggregate (and optional format conversion).

Some fun

The structure of the control table is so similar to the data expected by `moveValuesToColumns*()` that you can actually send the control table through `moveValuesToColumns*()` to illustrate the kernel of the transformation.

```
controlTable <- dplyr::tribble(~group, ~col1, ~col2,
                               'aa',  'c1',  'c2',
                               'bb',  'c3',  'c4')
tallTableName <- 'dc'
d <- dplyr::copy_to(my_db, controlTable, tallTableName)
keyColumns <- NULL
wideTableName <- moveValuesToColumnsN(tallTable = tallTableName,
                          controlTable = controlTable,
                          keyColumns = keyColumns,
                          my_db = my_db)
dw <- dplyr::tbl(my_db, wideTableName)
knitr::kable(dw)
```

c1	c2	c3	c4
c1	c2	c3	c4

The transformed table is essentially an example row of the wide-form.

And we can, of course, map back.

This "everything maps to a row" form means the control table is

essentially a graphical representation of the desired data transform. It also helps make clear that just about *any* even more general shape to shape transform can be achieved by a `moveValuesToColumnsN()` followed by a `moveValuesToRowsN()`.[1]

[1] Or by the a `moveValuesToRowsN()` followed by a `moveValuesToColumnsN()`. One direction is storing all intermediate values in a single denormalized column, the other is storing in many RDF-triple like rows.

```
moveValuesToRowsN(wideTable = wideTableName,
                  controlTable = controlTable,
                  my_db = my_db) %>%
  dplyr::tbl(my_db, .) %>%
  arrange(group) %>%
  knitr::kable()
```

group	col1	col2
aa	c1	c2
bb	c3	c4

Conclusion

`cdata::moveValuesToRows*()` and `cdata::moveValuesToColumns*()` represent two very general "fluid data" or "coordinatized data" operators that have database scale (via `DBI/dbplyr/dplyr`) and big data scale implementations (via `Sparklyr`). Some very powerful data transformations can be translated into the above explicit control table terminology. The extra details explicitly managed in the control table notation makes for clear calling interfaces.

```
for(ti in tng(dumpList = TRUE)) {
  dplyr::db_drop_table(my_db, ti)
}
DBI::dbDisconnect(my_db)
```