# Package StQT

*Luis Sanguiao Sande*

*13 de abril de 2016*

## Introduction

The StQT package is an attempt to standardize the representation of data transformations, in the same sense that StQ standardizes the representation of data. In fact, it stands for StQ Transformation. But, what do we understand by data transformation? In this context, it means a process that calculates a new data set from an old data set. For example, an imputation, a summary or even a forecast are transformations. But a data merge **is not** a transformation since we are using a second data set (except for a self data merge which will be explained later). If we would happen to need a transformation that uses information from two data sets, we would have to join them into a single data set before.

The main goals of this package are:

- To make clear the steps of a transformation, fixing a few simple rules, which we can use to express any transformation as a sequence of simple steps. We should think that each rule is a function with some instructions about how you use it on the data.
- To add an abstraction layer, which allow us to work with transformations without knowing how they work in the inside. With this package, once we have built a library of transformations (more or less basic), we should be able to combine them into new more complicated transformations.

It is important to focus in the first objective when you write the rules of a transformation. The rules should be flexible enough to represent any transformation in a understandable way, but as usual there is nothing that prevents an unclear codification.

To achieve the second main goal, we are going to implement several tools.

## The representation

It is strongly inspired in the syntax of operator := from data.table package. The main idea is to apply a set of rules according to the sentence `data[<domain>,<output>:=<fun>(<input>),by=<by>]` where each <> surrounded item is an element of the rule. As long as function definition is included in the object, it is obvious that we can represent this way almost everything. Almost, because the number of rows of data.table data, would not be changed. The consequence was to allow a direct assignation in `<output>` field which when present, will create new rows. For a more detailed view of the syntax of the rules read next section.

Note that if we remove `<domain>` and `<by>` and put `<fun>` and `<input>` as the only elements of the rule we still can get any transformation (even with one rule). But this rules representation would be useless in that case, since you have a function (an R function) that does all the work. And anyone who wants to know what the rule does, would have to examine carefully the function. So, generally speaking: keep the functions as simple as possible and try to use the other fields of the rule to achieve that. And be conscious that a lower number of rules **is not** necessarily better.

So, a transformation object has two slots: a data.frame with the rules and a list of the functions called by the rules. The functions are inside the object because that way the object has all the information needed to make the transformation. You don't need to load the functions into the workspace or check dependencies, you just load the transformation an apply it.

## Detailed syntax

There are three types of rules in StQT:

- Horizontal: These are by far the more common rules and they are applied as we had said with a command like `data[<domain>,<output>:=<fun>(<input>),by=<by>]`. This creates new or override `<output>` columns with one element per row applying the usual recycling. Next, an elementary example.

```
require(data.table)
require(StQT)
dt <- data.table(Stratum = c("1","2","3"), SD = c(1.54,1.12,1.88))
dt
```

```
##    Stratum   SD
## 1:       1 1.54
## 2:       2 1.12
## 3:       3 1.88
```

```
T1 <- NewStQT(data.frame(
        output = "VAR",
        fun = "*",
        input = "SD,SD",
        stringsAsFactors = FALSE))
TApply(dt,T1)[]
```

```
##    Stratum   SD    VAR
## 1:       1 1.54 2.3716
## 2:       2 1.12 1.2544
## 3:       3 1.88 3.5344
```

We had a table with strata identified and its standard deviation. The transformation binds a new column with the variance of the correspondent strata. We could have defined a function returning the square of its argument. But we don't need to do that, because we can use R primitive function `*` calculating VAR as SD $\times$ SD.

- Vertical: Similar to horizontal rules, but output includes some assignments (one or more). For each `<by>` group an additional row is created which copies the variables in `<by>`, sets the assigned variables and calculates the unassigned in a similar way to an horizontal rule. Let's see an easy example.

```
require(data.table)
require(StQT)
dt <- data.table(Region = c("1","2","3"), GDP = c(1.38,0.94,1.23))
dt
```

```
##    Region  GDP
## 1:      1 1.38
## 2:      2 0.94
## 3:      3 1.23
```

```
T1 <- NewStQT(data.frame(
        output = "Region='Total',GDP",
        fun = "sum",
        input = "GDP",
        stringsAsFactors = FALSE))
TApply(dt,T1)
```

```
##    Region  GDP
## 1:      1 1.38
## 2:      2 0.94
## 3:      3 1.23
## 4:  Total 3.55
```

In this case `<by>` is empty, so we have only one group (the whole table) and in consequence we only bind one new row.

In the other extreme situation we have one new column per row:

```
dt <- data.table(Region = c("1","2","3"), GDP = c(1.38,0.94,1.23))
dt
```

```
##    Region  GDP
## 1:      1 1.38
## 2:      2 0.94
## 3:      3 1.23
```

```
T1 <- NewStQT(data.frame(
        output = "Copy=TRUE,GDP",
        fun = "sum",
        input = "GDP",
        by = "Region",
        stringsAsFactors = FALSE))
TApply(dt,T1)
```

```
##    Region  GDP Copy
## 1:      1 1.38   NA
## 2:      2 0.94   NA
## 3:      3 1.23   NA
## 4:      1 1.38 TRUE
## 5:      2 0.94 TRUE
## 6:      3 1.23 TRUE
```

- Internal: These are not real functions but *special* codes in `<fun>` field that trigger concrete behaviors different to usual horizontal or vertical rules. They are *FunDelVar*, *FunDelRow* and *FunAutoLink* so far. Except for *FunDelRow*, we could write functions to get the same behavior (at least when applied to data.table objects), but they are anyways useful because they implement common tasks in a standard way which helps to make understandable the rules.

*FunDelVar* removes the variables specified in output, *FunDelRow* removes the rows in `<domain>` and *FunAutoLink* copies the `<input>` variables to the `<output>` variables, linking rows according to the equality specified in domain. The equality must be like `a == b & c == d & ...`, and be careful: in this case the equality is not commutative! When you put `a==b` you are linking each row with other row that has in variable *b* the same value than the original row has in variable *a*. So, if you swap *a* and *b*, you are making a different linkage. Now, let's see some examples:

```
dt <- data.table(Unit = 1:10, Value = rnorm(10), Next = c(2:10,NA))
T1 <- NewStQT(data.frame(
        domain = c("Next==Unit","","Value < 0"),
        output = c("Value_next","Value_next",""),
        fun = c("FunAutoLink","FunDelVar","FunDelRow"),
        input = c("Value","",""),
        stringsAsFactors = FALSE))
dt
```

```
##     Unit       Value Next
##  1:    1  0.01408321    2
##  2:    2  0.77049118    3
##  3:    3 -1.04071411    4
##  4:    4  0.04011127    5
##  5:    5  1.00544674    6
##  6:    6 -0.07785795    7
##  7:    7  1.34017642    8
##  8:    8  0.94152489    9
##  9:    9  0.47242680   10
## 10:   10  0.14233300   NA
```

```
TApply(dt,T1[1])
dt
```

```
##     Unit       Value Next  Value_next
##  1:   10  0.14233300   NA          NA
##  2:    1  0.01408321    2  0.77049118
##  3:    2  0.77049118    3 -1.04071411
##  4:    3 -1.04071411    4  0.04011127
##  5:    4  0.04011127    5  1.00544674
##  6:    5  1.00544674    6 -0.07785795
##  7:    6 -0.07785795    7  1.34017642
##  8:    7  1.34017642    8  0.94152489
##  9:    8  0.94152489    9  0.47242680
## 10:    9  0.47242680   10  0.14233300
```

```
TApply(dt,T1[2])
dt
```

```
##     Unit       Value Next
##  1:   10  0.14233300   NA
##  2:    1  0.01408321    2
##  3:    2  0.77049118    3
##  4:    3 -1.04071411    4
##  5:    4  0.04011127    5
##  6:    5  1.00544674    6
##  7:    6 -0.07785795    7
##  8:    7  1.34017642    8
##  9:    8  0.94152489    9
## 10:    9  0.47242680   10
```

```r
TApply(dt,T1[3])
```

```
##    Unit      Value Next
## 1:   10 0.14233300   NA
## 2:    1 0.01408321    2
## 3:    2 0.77049118    3
## 4:    4 0.04011127    5
## 5:    5 1.00544674    6
## 6:    7 1.34017642    8
## 7:    8 0.94152489    9
## 8:    9 0.47242680   10
```

```r
dt
```

```
##     Unit       Value Next
##  1:   10  0.14233300   NA
##  2:    1  0.01408321    2
##  3:    2  0.77049118    3
##  4:    3 -1.04071411    4
##  5:    4  0.04011127    5
##  6:    5  1.00544674    6
##  7:    6 -0.07785795    7
##  8:    7  1.34017642    8
##  9:    8  0.94152489    9
## 10:    9  0.47242680   10
```

The examples also show how to use the operator `[`, that applies a subset of the rules. Note that some rules modify the original data, but some others do not. *TApply* uses data.table methods by reference when possible, but sometimes they are unavailable (row insertion and deletion). I strongly recommend to assign the old table to the new one `dt <- TApply(dt, T1)` to avoid confusion. If you don't want to loose information just don't remove the data.

In the first rule, we are storing the value of the next element according to *Next* field in the new variable *Value_next*. In the second one, we are removing the new column so we recover the original data.table. Finally we remove the rows where *Value* is negative, but as seen *dt* remains unchanged.

There is also an `<order>` field which if not empty, orders the data by the variables specified.

Note that `<key>` field is ignored unless we are transforming an StQ object (see next section).

## How does a transformation work on StQ objects

We are assuming that the reader is familiarized with StQ objects and its internal representation. First of all, we have to ask ourselves what a variable is. When you dcast an StQ object, you get a lot of columns with long `"_"` separated names. For us, those **are not** variables. For us, a variable is an IDDD-sort element of the data slots from a DD object. Moreover it would be useful to think in it as a vector with a component for each qualifiers combination.

But we have to make some concessions to the qualifiers, because they are the "dimensions" of our vector variable. So we have to allow them in the rules, but we should avoid them when possible in `<input>` and `<output>` field, specially `<input>`. In `<by>` and `<domain>` they are perfectly acceptable. Let's say we have a variable by unit and region, and we want the total by unit. Then we only have to sum the variable by unit and that's it. In this example, we have had to use the unit qualifier in the `<by>` field, and the rule seems quite clear. By the way, we have to specify that the resulting variable has unit as only qualifier (and not unit *and* region). This is what `<key>` field is intended for. We know that R sum function returns a single value, and since we are grouping by unit, the `<key>` should be unit. But if we substitute sum by another function, and we only have its closure and arguments, it would be very difficult to find an automatic way to decide if it returns a single value. So we specify `<key>` as unit.

A second question is what do we do with more than one variable. Let's see. If we have two or more variables whose qualifiers are exactly the same, we only have to put them as columns and we have a data.table where to apply the rule as usual. It is natural to associate values of both variables when they have the same key. When their qualifiers are different, we look for a variable whose qualifiers are the union of all the qualifiers of all the variables. We are going to name this variable as main variable, and we are going to use it to link them all. If such variable does not exist, that means that there are more than one maximal variables (in the sense that its qualifiers are not included in the other qualifiers), and there is not an unique way to link them. In this case the rule is considered no valid, and an error is shown. If we need to link this way, we have to use auxiliary variables.

## Other methods