5

RevoScaleR Package

RevoScaleR package is part of the Microsoft R Server and R Services. With rapid development and constant upgrades, this chapter will cover version 8.X and version 9.X, latter available also with SQL Server 2017. Changes and upgrades in version 9.X are not to be overlooked and will be covered as well. The outline of this chapter is following:

* Limitations of R challenged
* Scalable and distributive computational environment
* Functions for data preparation
* Functions for descriptive statistics
* Functions for statistical tests and sampling
* Functions for predictive modeling

Primarily, this R package is designed to be handled in ecosystems, where clients would be connecting to Microsoft R Server in order to have R code executed against much more powerful server, which would presumably hold whole datasets, not just a smaller portion, on which people working on clients would be dealing with.

# Limitations of R challenged

Prior to SQL Server 2016 (and 2017) BI and data scientists would have the OLAP cubes, DMX language and all super awesome and cool Microsoft algorithms available within Sql Server Analysis services (SSAS). But, with rapid changes and bigger market demands, the need for integration of a open-source product (whether R or Python or Perl Data Language or any other) was practically already there. And the next logical step was to integrate it with on. Microsoft sought for solution and ended up acquiring Revolution analytics, that have put the on the track again. Revolution R has addressed major issues concerning R language.

Microsoft addressed the R limitations. Many of these limitations were aimed in faster data exploration and parallel programming techniques in R. In addition to this, also MKL computations have been enchacned, therefore making matrix-wise calculations even faster, also scalar calculation and also calculation resulting in cartesian-product.

Limitations were addressed ana alo solved:

* Communications overhead, particularly an issue with fine-grained parallelism consisting of a very large number of relatively small tasks;
* Load balance, where the computing resources aren't contributing equally to the problem;
* Impacts from use of RAM and virtual memory, such as cache misses and page faults;
* Network effects, such as latency and bandwidth, that impact performance and communication overhead;
* Interprocess conflicts and thread scheduling;
* Data access and other I/O considerations.

# Scalable and distributive computational environment

RevoScaleR package has following functions available, which will be covered in detailed throughout the chapter.

To get the list of all the ScaleR functions, following T-SQL can be used:

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'require(RevoScaleR)

OutputDataSet <- data.frame(ls("package:RevoScaleR"))'

WITH RESULT SETS

(( Functions NVARCHAR(200) ))

You will be returned the table in SSMS with all relevant rx functions that can be used with RevoScaleR package.

Based on the list of these functions, a simpler and better overview of the functions can be prepared:

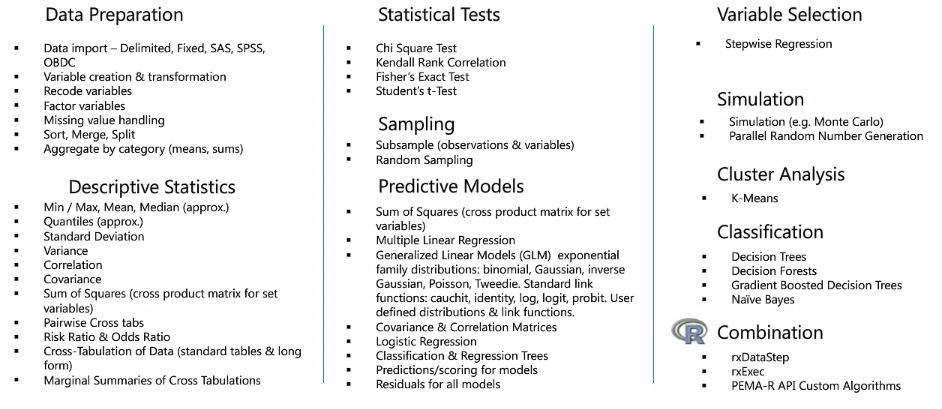


Figure 1: List of RevoScaleR functions (Courtesy of Microsoft)

**Functions for data preparation**

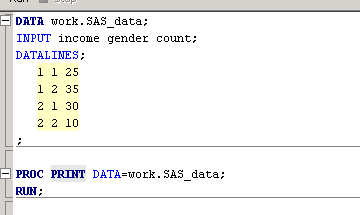
**Data Import from SAS, SPSS, ODBC**

Importing the data into R or SQL Server tables is not the main focus of RevoScaleR library, but once this is on the list, let’s briefly look into it.

**Importing SAS data**

SAS is among popular programs for data analysis if not the most popular for statistical analysis, data mining and machine learning. Therefore, let us create a simple SAS file and read is using ScaleR function.

With the following SAS code (code is available along with the book), one can very easy create a sample dataset:



Now let’s assume that our SAS data is stored in the file sas\_data.sas7bdat as the code suggest in PROC DATA statement.

With the following R Code, we can extract and import this dataset into R data.frame:

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

library(RevoScaleR)

SampleSASFile <- file.path(rxGetOption("sampleDataDir"), "sas\_data.sas7bdat")

#import into Dataframe

OutputDataSet <- rxImport(SampleSASFile)

'

WITH RESULT SETS

((

income INT

,gender INT

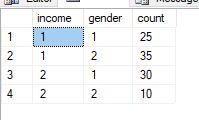
,[count] INT

))

Make sure that your sampleDataDir holds the data sample or if you specify some other path like:

SampleSASFile <- file.path(("C:\\Users\\TomazK\\Documents\\CH05"), "sas\_data.sas7bdat")

That you have granted access to this working folder. In both ways, you should get results presented as a table, read from the SAS file like:



Another way of importing SAS file by using RxSasData directly (in this case from R)

SampleSASFile <- file.path(("C:\\Users\\tomazK\\CH05"), "sas\_data.sas7bdat")

sasDS <- RxSasData(SampleSASFile, stringsAsFactors = TRUE,

colClasses = c(income = "integer", gender= "integer", count="integer"),

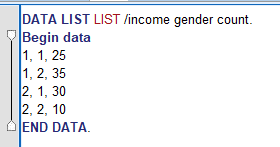
rowsPerRead = 1000)

rxHistogram( ~F(gender)|F(income), data = sasDS)

And you can easily generate a histogram from the SAS data file.

**Importing SPSS data**

With SPSS, the procedure is similar. Following SPSS syntax (syntax is included with this chapter) generates the sample dataset which is stored on your local machine.



And getting data into R Services using SPSS sav file, that is generated from the above SPSS syntax is relative same as with SAS file:

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

library(RevoScaleR)

SampleSPSSFile <- file.path(rxGetOption("sampleDataDir"), "spss\_data.sav")

#import into Dataframe

OutputDataSet <- rxImport(SampleSPSSFile)

'

WITH RESULT SETS

((

income INT

,gender INT

,[count] INT

))

In addition to this, RevoScaleR has a special function to read directly the SPSS file, called RxSpssData. Following R code can accomplish the same result as the above T-SQL code:

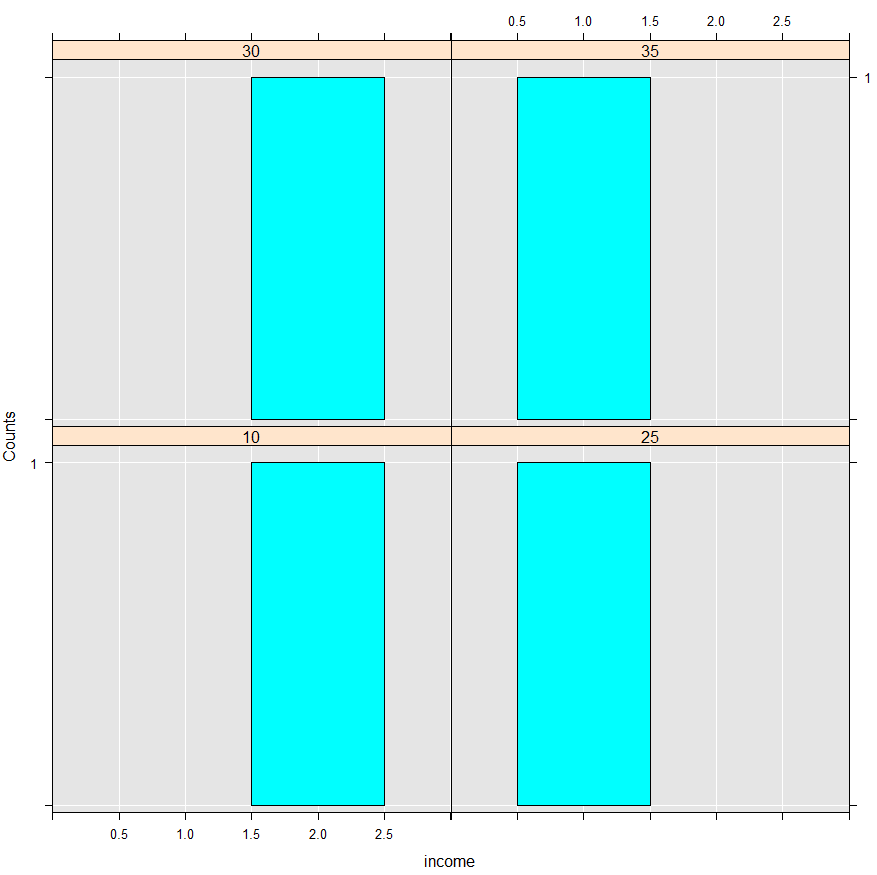
SampleSPSSFile <- file.path(("C:\\Users\\tomazK\\CH05"), "spss\_data.sav")

spssDS <- RxSpssData(SampleSPSSFile, stringsAsFactors = TRUE,

colClasses = c(income = "integer", gender= "integer", count="integer"),rowsPerRead = 1000)

rxHistogram( ~F(income)|F(count), data = spssDS)

And RevoScaleR histogram can be used directly with the SPSS datasource, generating a simple histogram:



**Importing data using ODBC**

Using ODBC driver extends the accessibility to almost any kind of database, for which you can obtain the driver and have common RDBM model.

RevoScaleR Package extends the list of ODBS drivers also to support systems on Linux and other systems. Using ODBC you can connect to MySQL, Oracle, PostgreSQL, SQL Server on Linux, Cloudera and Teradata (which in this case is much better to use RxTeradata function).

Following is the example that we will use ODBC driver to get data from another SQL server instance, both using RxOdbcData function and RxSqlServerData, since they are interchangeable.

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

library(RevoScaleR)

sConnectStr <- "Driver={ODBC Driver 13 for SQL Server};Server=SICN-KASTRUN;Database=AdventureWorks;Trusted\_Connection=Yes"

sQuery = "SELECT TOP 10 BusinessEntityID,[Name],SalesPersonID FROM [Sales].[Store] ORDER BY BusinessEntityID ASC"

sDS <-RxOdbcData(sqlQuery=sQuery, connectionString=sConnectStr)

OutputDataSet <- data.frame(rxImport(sDS))

'

WITH RESULT SETS

((

BusinessEntityID INT

,[Name] NVARCHAR(50)

,SalesPersonID INT

));

Which would be same as running this on the same server:

USE AdventureWorks;

GO

SELECT

TOP 10

BusinessEntityID

,[Name]

,SalesPersonID

FROM [Sales].[Store]

ORDER BY BusinessEntityID ASC

In the case of using RxOdbcData, you should also check the credentials and you might want to check also which user you are using to run the script. You can also create new login and user and use to check and to execute the script.

EXECUTE AS USER = 'MSSQLSERVER01'

GO

-- YOUR CODE

REVERT;

GO

**Variable creation and data transformation**

Variable creation and data transformation are two of the processes when defining data munging and data wrangling tasks. These tasks are important for proper data preparation and makes future task when analyzing data, easier.

These functions are available as followings:

* Variable creation and recoding
* Data transformation
* Handling missing values
* Sorting, merging and splitting data sets
* Aggregate by category (means, sums) – similar to T-SQL aggregations and windows functions

This part will cover some of the following functions, mainly focusing on data transformation, handling missing values and splitting data sets.

RxDataSource, rxDataStep, rxDataStepXdf, RxFileSystem, rxFindFileInPath, rxFindPackage, rxFisherTest, RxForeachDoPar, rxGetInfo, rxGetInfoXdf, rxGetJobInfo, rxGetJobInfo, rxGetOption, rxGetVarInfo, rxGetVarNames, rxImport, rxImportToXdf, rxIsOpen, rxOdbcData, rxOptions, rxOpen, rxQuantile, rxReadXdf, rxResultsDF, rxSetFileSystem, rxSetInfo, rxSetInfoXdf, rxSort, rxSetVarInfoXdf, rxSetVarInfo, rxMarginals, rxMerge, rxMergeXdf

When using In-database R Service (or In-database Machine learning service, counting in also Python for SQL server 2017), one should keep in mind where and how to do any kind of data transformation, data wrangling, as well as sorting and/or merging. After running many of the performance and speed tests, it became very clear that many of the munging and wrangling tasks should be done in-database, before sending the dataset to be executed by sp\_execute\_external\_script. This set of functions is the only set, where the computation context should be considered as very important one. All other functions for statistical tests, descriptive statistics and predictive statistics are can easily be used with external procedure, without compromising the decrease in performance or time.

Starting with rxDataStep function, it gives us many opportunities to extract and generated XDF file, also using in-database R:

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

df\_sql <- InputDataSet

df\_sql4 <- data.frame(df\_sql)

outfile <- file.path(rxGetOption("sampleDataDir"), "df\_sql4.xdf")

rxDataStep(inData = df\_sql4, outFile = outfile, overwrite = TRUE)'

,@input\_data\_1 = N'

SELECT

BusinessEntityID

,[Name]

,SalesPersonID

FROM [Sales].[Store]'

This will generate the df\_sql4.xdf file on your sample data directory. If you are interested to where this folder is pointing to, you can do the following:

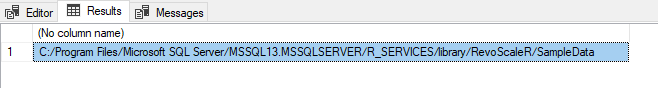
EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

OutputDataSet <- data.frame(path = file.path(rxGetOption("sampleDataDir")))'

It will be something as:



And make sure that you have access granted for the user, executing rxDataStep code, because the code will be creating a physical XDF file on the destined location.

Variable creation and recoding

Using rxGetVarInfo will expose the information about the data.frame to sp\_execute\_external\_script output. Immediately, it is obvious that, some of those functions were never designed for presenting the output to data.frame, but were designed only for exploring the dataset. Some of these functions - same applies for the rxGetVarInfo – will give a nice output in R environment, but will be hard to manipulate to fit in data frame for outputting in SQL Server database.

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

library(RevoScaleR)

df\_sql <- InputDataSet

var\_info <- rxGetVarInfo(df\_sql)

OutputDataSet <- data.frame(unlist(var\_info))'

,@input\_data\_1 = N'

SELECT

BusinessEntityID

,[Name]

,SalesPersonID

FROM [Sales].[Store]'

Note that, we are using unlist function, that unlists the set of lists in a single vector. Just to compare the outputs, we can run the same script in R environment:

library(RevoScaleR)

sConnectStr <- "Driver={ODBC Driver 13 for SQL Server};Server=TOMAZK\\MSSQLSERVER2017;Database=AdventureWorks;Trusted\_Connection=Yes"

sQuery = "SELECT BusinessEntityID,[Name],SalesPersonID FROM [Sales].[Store] ORDER BY BusinessEntityID ASC"

sDS <-RxOdbcData(sqlQuery=sQuery, connectionString=sConnectStr)

df\_sql <- data.frame(rxImport(sDS))

Now running the “rxGetVarInfo(df\_sql)” will give you slightly different export:

> var\_info <- rxGetVarInfo(df\_sql)

> var\_info

Var 1: BusinessEntityID, Type: integer, Low/High: (292, 2051)

Var 2: Name, Type: character

Var 3: SalesPersonID, Type: integer, Low/High: (275, 290)

And after unlisting with unlist() function, we get the same information, written slightly different:

> df <- data.frame(unlist(var\_info))

> df

unlist.var\_info.

BusinessEntityID.varType integer

BusinessEntityID.storage int32

BusinessEntityID.low 292

BusinessEntityID.high 2051

Name.varType character

Name.storage string

SalesPersonID.varType integer

SalesPersonID.storage int32

SalesPersonID.low 275

SalesPersonID.high 290

This will give you the idea that some of these functions for variable creation and recoding were meant more for R data engineer as for T-SQL data engineers.

Function rxGetInfo() will get you the size of your dataset and number of observations / variables:

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

library(RevoScaleR)

df\_sql <- InputDataSet

var\_info <- rxGetInfo(df\_sql)

OutputDataSet <- data.frame(unlist(var\_info))'

,@input\_data\_1 = N'

SELECT

BusinessEntityID

,[Name]

,SalesPersonID

FROM [Sales].[Store]'

Same logic applies, if you run this R environment, you will get neater display of information:

> rxGetInfo(df\_sql)

Data frame: df\_sql

Number of observations: 701

Number of variables: 3

Adding some additional parameters to this function also yield richer output, such as:

> rxGetInfo(df\_sql, getVarInfo = TRUE)

Data frame: df\_sql

Number of observations: 701

Number of variables: 3

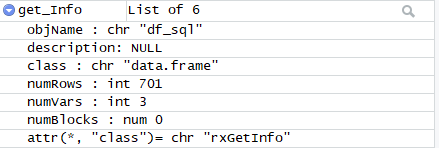
Variable information:

Var 1: BusinessEntityID, Type: integer, Low/High: (292, 2051)

Var 2: Name, Type: character

Var 3: SalesPersonID, Type: integer, Low/High: (275, 290)

Same as rxGetVarInfo, also rxGetInfo, will create a list of elements. rxGetVarInfo will generate a list of lists, where number of tuples equals the number of variables, and rxGetInfo will generate list of 6 elements, where each list will hold information about the object:



Knowing this, the above T-SQL executions can be slightly altered, so that the relevant information is displayed in more readable format, by presenting elements (tuples) to the result set:

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

library(RevoScaleR)

df\_sql <- InputDataSet

get\_Info <- rxGetInfo(df\_sql)

Object\_names <- c("Object Name", "Number of Rows", "Number of Variables")

Object\_values <- c(get\_Info$objName, get\_Info$numRows, get\_Info$numVars)

OutputDataSet <- data.frame(Object\_names, Object\_values)'

,@input\_data\_1 = N'

SELECT

BusinessEntityID

,[Name]

,SalesPersonID

FROM [Sales].[Store]'

WITH RESULT SETS

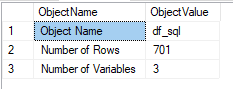
((

ObjectName NVARCHAR(100)

,ObjectValue NVARCHAR(MAX)

));

With returned results in SQL Server Management Studio:



This looks very neat and spending some extra effort will for sure give much better formatted results, that will be easier to read as well as much more informative.

In this example, you have seen also how to create new variable, this comes especially handy also, when cleaning the data or when recoding / bucketing data.

Let’s suppose that you want to recode the values of the exists variable in the dataset and create a new one. It can be done using a standard R code like:

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

df\_sql <- InputDataSet

#first create an empyt variable

df\_sql$BusinessType <- NA

df\_sql$BusinessType[df\_sql$BusinessEntityID<=1000] <- "Car Business"

df\_sql$BusinessType[df\_sql$BusinessEntityID>1000] <- "Food Business"

OutputDataSet <- df\_sql'

,@input\_data\_1 = N'

SELECT

BusinessEntityID

,[Name]

,SalesPersonID

FROM [Sales].[Store]'

WITH RESULT SETS

((

BusinessEntityID INT

,[Name] NVARCHAR(MAX)

,SalesPersonID INT

,TypeOfBusiness NVARCHAR(MAX)

));

Or by using function rxDataStep() and transformFunc parameter with additional function for creating new variable by transforming old values:

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

library(RevoScaleR)

df\_sql <- InputDataSet

df\_sql$BusinessEntityID\_2 <- NA

myXformFunc <- function(dataList) {

#dataList$BussEnt <- 100 \* dataList$BusinessEntityID

if (dataList$BusinessEntityID<=1000){dataList$BussEnt <- "Car Business"} else {dataList$BussEnt <- "Food Business"}

return (dataList)

}

df\_sql <- rxDataStep(inData = df\_sql, transformFunc = myXformFunc)

OutputDataSet <- df\_sql'

,@input\_data\_1 = N'

SELECT

BusinessEntityID

,[Name]

,SalesPersonID

FROM [Sales].[Store]'

WITH RESULT SETS

((

BusinessEntityID INT

,[Name] NVARCHAR(MAX)

,SalesPersonID INT

,TypeOfBusiness NVARCHAR(MAX)

));

Function rxDataStep() is very powerful function mainly for data selection, sub setting, data transformation and creation of new variables for desired dataset.

Dataset subsetting

Subsetting the data is also relatively straight forward using rxDataStep() function:

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

library(RevoScaleR)

df\_sql <- InputDataSet

df\_sql\_subset <- rxDataStep(inData = df\_sql, varsToKeep = NULL, rowSelection = (BusinessEntityID<=1000))

OutputDataSet <- df\_sql\_subset'

,@input\_data\_1 = N'

SELECT

BusinessEntityID

,[Name]

,SalesPersonID

FROM [Sales].[Store]'

WITH RESULT SETS

((

BusinessEntityID INT

,[Name] NVARCHAR(MAX)

,SalesPersonID INT

));

Please keep in mind that subsetting operations using R code might bring unnecessary memory and I/O costs, especially, when pumping the whole datasets into R, instead of subsetting the data prior. In the example above the using rowSelection parameter in rxDataStep can easily be replaced with WHERE clause in the @input\_data\_1 argument. So bear in mind and always avoid unnecessary traffic.

Dataset merging

Function rxMerge() merges two datasets at one time. Dataset must be a dataframe (or XDF format) and operates similarly as JOIN clause in T-SQL (rxMerge() should not be mixed or confused with T-SQL MERGE statement). Two datasets are merged based on one or more variables using the argument matchVars and in addition, when using the local compute context (which we are using in the next sample), the sorting of the data needs to be defined as well, since data.frames – as collection of vectors – in R are not presorted or do not hold any sorts whatsoever. So if there is not any presorting done, the argument autoSort must be set to true.

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

library(RevoScaleR)

df\_sql <- InputDataSet

someExtraData <- data.frame(BusinessEntityID = 1:1200, department = rep(c("a", "b", "c", "d"), 25), Eff\_score = rnorm(100))

df\_sql\_merged <- rxMerge(inData1 = df\_sql, inData2 = someExtraData, overwrite = TRUE, matchVars = "BusinessEntityID", type = "left" ,autoSort = TRUE)

OutputDataSet <- df\_sql\_merged'

,@input\_data\_1 = N'

SELECT

BusinessEntityID

,[Name]

,SalesPersonID

FROM [Sales].[Store]'

WITH RESULT SETS

((

BusinessEntityID INT

,[Name] NVARCHAR(MAX)

,SalesPersonID INT

,Department CHAR(1)

,Department\_score FLOAT

));

This T-SQL code creates a left join on both datasets. Where dataframe 2 (called: someExtraData) is created on the fly, but it can be any other dataframe read from a XDF file or any manually inserted dataset, that will be joined in R runtime. Also pay the attention to which is the first and which is the second data frame in combination to which type of join you are using. The example above specifies:

inData1 = df\_sql, inData2 = someExtraData, type = "left"

but if the order of the data frames would be changed

inData1 = someExtraData , inData2 = df\_sql, type = "left"

and the output would be different.

**Functions for descriptive statistics**

Descriptive statistics gives insights into understanding data. These are summary statistics that describes a given dataset using summarizing features and measures, such as central tendency, measure of spread (or variability). Central tendency includes calculation of mean, median, mode, whereas measures of variability include range, quartiles, minimum and maximum value, variance and standard deviation, as well as skewness and kurtosis.

These statistics are covered my rx- functions in RevoScaleR packages, meaning that one can use all the computational advantages of the package by calling: rxSummary, rxCrossTabs, rxMarginals, rxQuantile, rxCube, rxHistogram, without worrying either about the performance, out of memory exceptions and which R package holds the right function.

We will be using the [Sales].[vPersonDemographics] view in the AdventureWorks database, that will neatly show the usability of these functions.

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

library(RevoScaleR)

df\_sql <- InputDataSet

summary <- rxSummary(~ TotalChildren, df\_sql, summaryStats = c( "Mean", "StdDev", "Min", "Max","Sum","ValidObs", "MissingObs"))

OutputDataSet <- summary$sDataFrame'

,@input\_data\_1 = N'

SELECT \* FROM [Sales].[vPersonDemographics] WHERE [DateFirstPurchase] IS NOT NULL'

WITH RESULT SETS

((

VariableName NVARCHAR(MAX)

,"Mean" NVARCHAR(100)

,"StdDev" NVARCHAR(100)

,"Min" NVARCHAR(100)

,"Max" NVARCHAR(100)

,"Sum" NVARCHAR(100)

,"ValidObs" NVARCHAR(100)

,"MissingObs" NVARCHAR(100)

));

With one line or R code, one can get some summary statistics. I prefer using the argument summaryStats to list the statistics, but please note, that order of the statistics does not mean the same order in the output. In addition, using the element summary$sDataFrame sDataFrame as a result from rxSummary, will automatically generate the data frame that will contain all the summaries for numeric variables.

The result of the T-SQL query is following:



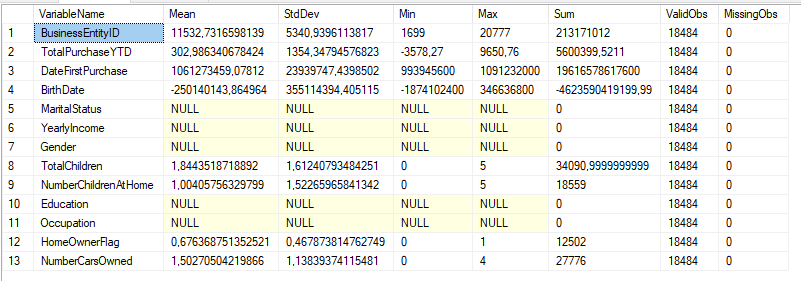
Function rxSummary() also holds formula, where one can specify which variables the function will take into calculation of descriptive statistics. In our case, we have used:

rxSummary(~ TotalChildren, ...

only the TotalChildren variable. But let’s assume, we want to get descriptives for all the variables, we simply write:

rxSummary(~., ....

and we will get statistics for all the variables. Like in the print-screen:



Note that only integers (continuous) type of variables will be taken into consideration, whereas variables as MaritalStatus, Education, Occupation,… will be presented as NULL, since these variables are treated as categorical variables in R.

For this we will need to specify first of all the factor variable and based on that, we will be able to run statistics.

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

library(RevoScaleR)

df\_sql <- InputDataSet

df\_sql\_r <- rxFactors(inData = df\_sql, sortLevels = TRUE,

factorInfo = list(MS = list(levels = c("M","S"), otherLevel=NULL, varName="MaritalStatus")))

summary <- rxSummary(~ MS, df\_sql\_r)

OutputDataSet <- data.frame(summary$categorical)'

,@input\_data\_1 = N'

SELECT \* FROM [Sales].[vPersonDemographics] WHERE [DateFirstPurchase] IS NOT NULL'

WITH RESULT SETS

((

MS NVARCHAR(MAX)

,"Counts" INT

));

This function will give the simple counts for the factor MaritalStatus:

MS Counts

M 10011

S 8473

Same logic can be applied to all other categorical variables. Formula in rxSummary() function also gives users the ability to combine different variables. For example, instead of using:

rxSummary(~ TotalChildren, ...

also this can be used:

rxSummary(NumberCarsOwned ~ TotalChildren, ...

This will calculate the observed statistics for both variables together:

Name Mean StdDev Min Max Sum ValidObs MissObs

NumberCarsOwned:TotalChildren 3.258656 4.473517 0 20 60233 18484 0

Same can be calculated also for categorical variables. These variables need to be recoded into factors first and later the same summary statistics can be calculated.

rxSummary(~ TotalChildren:F(MS), df\_sql\_r, ...

With complete T-SQL:

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

library(RevoScaleR)

df\_sql <- InputDataSet

df\_sql\_r <- rxFactors(inData = df\_sql, sortLevels = TRUE,factorInfo = list(MS = list(levels = c("M","S"), otherLevel=NULL, varName="MaritalStatus")))

summary <- rxSummary(~F(MS):TotalChildren, df\_sql\_r, summaryStats = c( "Mean", "StdDev", "Min", "Max", "ValidObs", "MissingObs", "Sum"), categorical=c("MS"))

OutputDataSet <- data.frame(summary$categorical)'

,@input\_data\_1 = N'

SELECT \* FROM [Sales].[vPersonDemographics] WHERE [DateFirstPurchase] IS NOT NULL'

WITH RESULT SETS

((

Category NVARCHAR(MAX)

,"MS" NVARCHAR(MAX)

,"Means" FLOAT

,"StDev" FLOAT

,"Min" INT

,"Max" INT

,"Sum" INT

,"ValidObs" INT

));

And results per each factor level:

Name Mean StdDev Min Max Sum ValidObs MissingObs

TotalChildren:F\_MS 1.844352 1.612408 0 5 34091 18484 0

Statistics by category (2 categories):

Category F\_MS Means StdDev Min Max Sum ValidObs

TotalChildren for F(MS)=M M 2.080412 1.583326 0 5 20827 10011

TotalChildren for F(MS)=S S 1.565443 1.601977 0 5 13264 8473

Quantile and deciles are also very usefull to see the data distribution and RevoScaleR packages provides function rxQuantile. Using T-SQL, the result set can be returned as following:

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

library(RevoScaleR)

df\_sql <- InputDataSet

quan <- rxQuantile(data = df\_sql, varName = "TotalChildren")

quan <- data.frame(quan)

values <- c("0%","25%","50%","75%","100%")

OutputDataSet <- data.frame(values,quan)'

,@input\_data\_1 = N'

SELECT \* FROM [Sales].[vPersonDemographics] WHERE [DateFirstPurchase] IS NOT NULL'

WITH RESULT SETS

((

Quartile NVARCHAR(100)

,QValue FLOAT

));

With result returning:

0% 25% 50% 75% 100%

0 0 2 3 5

We can also modify the and calculate deciles with a slight change to rxQuantile() function:

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

library(RevoScaleR)

df\_sql <- InputDataSet

dec <- rxQuantile(data = df\_sql, varName = "TotalChildren", probs = seq(from = 0, to = 1, by = .1))

dec <- data.frame(dec)

values <- c("0%","10%","20%","30%","40%","50%","60%","70%","80%","90%","100%")

OutputDataSet <- data.frame(values,dec)'

,@input\_data\_1 = N'

SELECT \* FROM [Sales].[vPersonDemographics] WHERE [DateFirstPurchase] IS NOT NULL'

WITH RESULT SETS

((

Decile NVARCHAR(100)

,DValue FLOAT

));

Calculating crosstabulations – the relationship between two (or more) variables – we will use two functions: rxCrossTabs and rxMargins. Crosstabulations are usually expressed in contingency table or any other [*n]\*[m]*  table format; which really depends on the number of levels, each variable will have.

We will take our two variables NumberCarsOwned and TotalChildren to explore the rxCrossTabs.

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

library(RevoScaleR)

df\_sql <- InputDataSet

crosstab <- rxCrossTabs(N(NumberCarsOwned) ~ F(TotalChildren), df\_sql, means=FALSE) #means=TRUE

children <- c(0,1,2,3,4,5)

OutputDataSet <- data.frame(crosstab$sums, children)'

,@input\_data\_1 = N'

SELECT \* FROM [Sales].[vPersonDemographics] WHERE [DateFirstPurchase] IS NOT NULL'

WITH RESULT SETS

((

NumberOfCarsOwnedSUM INT

,NumberOfChildren INT

));

Calculating crosstabulations using rxCrossTabs can give you two type of statistics: count of observations and mean of observations, given the category of intersect. This is manipulated using argument means=TRUE or means=FALSE. The function operates in a way, that will need the dependent variable(s) and independent variables(s) and in our example, the information can be retrieved from results as:

Cross Tabulation Results for: N(NumberCarsOwned) ~ F(TotalChildren)

Data: df\_sql

Dependent variable(s): N(NumberCarsOwned)

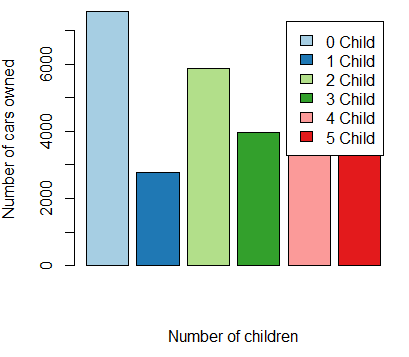
Number of valid observations: 18484

Number of missing observations: 0

Statistic: sums

In order to have the crosstabulation successfully calculated, the independent variables must be presented as factors, in this case variable TotalChildren has a F() function wrapped, denoting a factor conversion in the runtime.

This can be visualized using a standard barplot in base package or R:



Using following code:

library(RColorBrewer)

barplot(OutputDataSet$V1, xlab = "Number of children",ylab = "Number of cars owned",

legend.text = c("0 Child","1 Child","2 Child","3 Child","4 Child","5 Child"), col=brewer.pal(6, "Paired"))

Using variables that are categorical, there is no need for explicit convertion:

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

library(RevoScaleR)

df\_sql <- InputDataSet

crosstab <- rxCrossTabs(NumberCarsOwned ~ MaritalStatus, df\_sql, means=FALSE)

status <- c("M","S")

OutputDataSet <- data.frame(crosstab$sums, status)'

,@input\_data\_1 = N'

SELECT \* FROM [Sales].[vPersonDemographics] WHERE [DateFirstPurchase] IS NOT NULL'

WITH RESULT SETS

((

NumberOfCarsOwnedSUM INT

,MaritalStatus NVARCHAR(100)

));

But also transforms argument can be used to recode, recalculate or somehow transform any of the variables. Marginal statistics from the contingency tables deriving from rxCrossTabs, can be called using rxMarginals functions, that is simply wrapped around the rxCrossTabs.

Marginal statistics will give you sum or counts or means for each of the totals per row or per column for desired variable:

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

library(RevoScaleR)

df\_sql <- InputDataSet

mar <- rxMarginals(rxCrossTabs(NumberCarsOwned ~ F(TotalChildren), data=df\_sql, margin=TRUE, mean=FALSE))

OutputDataSet <- data.frame(mar$NumberCarsOwned$grand)'

,@input\_data\_1 = N'

SELECT \* FROM [Sales].[vPersonDemographics] WHERE [DateFirstPurchase] IS NOT NULL'

WITH RESULT SETS

((

GrandTotal INT

));

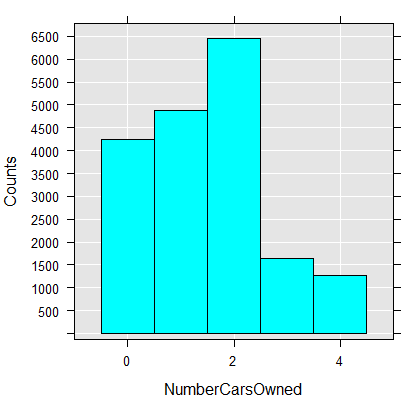
And the result is:

> mar$NumberCarsOwned$grand

[1] 27776

Exploring the data can also be done using the graphs and RevoScaleR comes packed with Line and bar plot, both designed to tackle large datasets.

Simplistic preview of one of the variables:

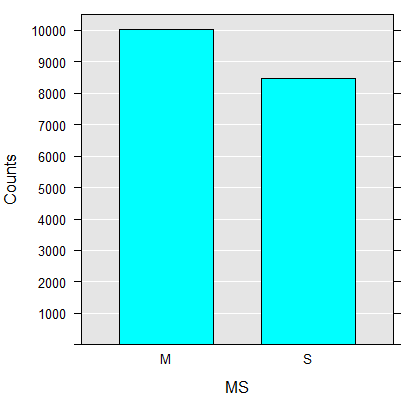


With following line of R code:

rxHistogram(~NumberCarsOwned, data=df\_sql)

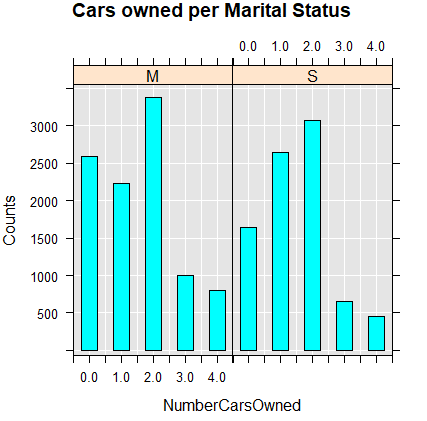
or already and previously converted to factor MaritalStatus:

rxHistogram(~F(MS), data=df\_sql\_r)



Also variables can be combined like:

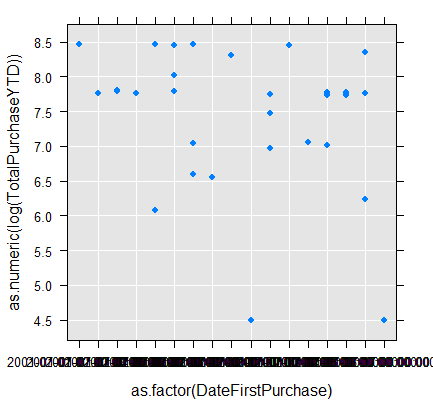
rxHistogram(~ NumberCarsOwned | F(MS), title="Cars owned per Marital Status", numBreaks=10, data = df\_sql\_r)

****

And instead of bar plot we can use also Line plot, but this time with different variables:

rxLinePlot(as.numeric(log(TotalPurchaseYTD)) ~ as.factor(DateFirstPurchase), data = df\_sql\_r, rowSelection=

DateFirstPurchase >= "2001-07-01 00:00:00.000" & DateFirstPurchase <= "2001-07-17 00:00:00.000", type="p")



So at the end we can combine all four graphs by using par() function arranging the 2 columns, each having 2 graphs:

# combined

h1 <- rxHistogram(~NumberCarsOwned, data=df\_sql)

h2 <- rxHistogram(~F(MS), data=df\_sql\_r)

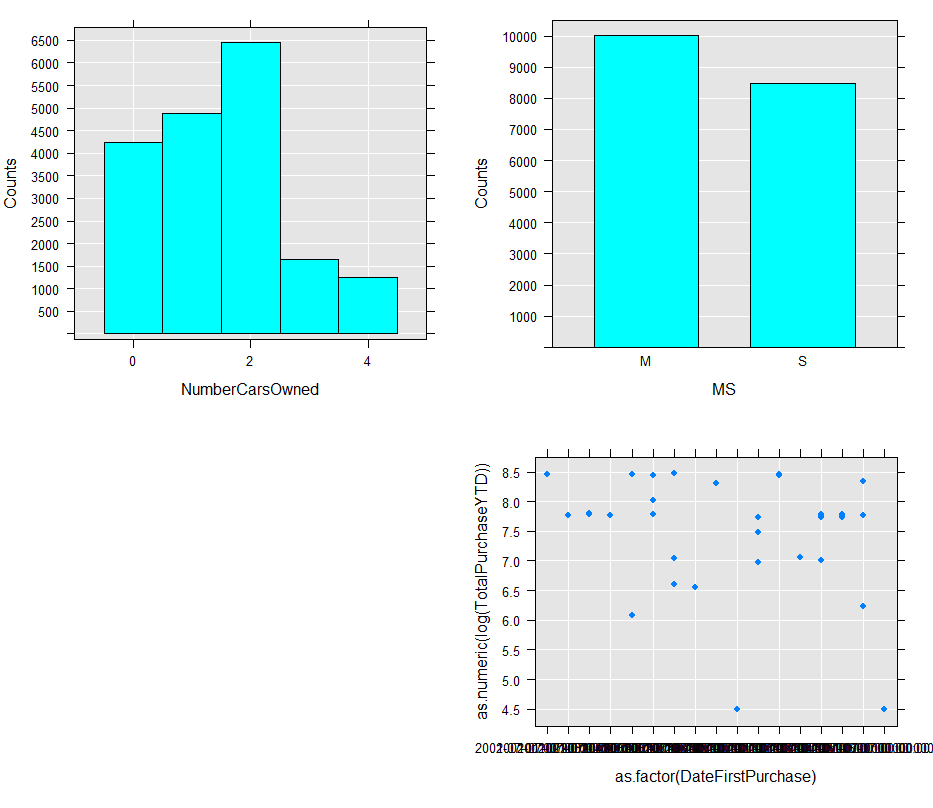
p1 <- rxLinePlot(as.numeric(log(TotalPurchaseYTD)) ~ as.factor(DateFirstPurchase), data = df\_sql\_r, rowSelection=

DateFirstPurchase >= "2001-07-01 00:00:00.000" & DateFirstPurchase <= "2001-07-17 00:00:00.000", type="p")

print(h1, position = c(0, 0.5, 0.5, 1), more = TRUE)

print(h2, position = c(0.5, 0.5, 1, 1), more = TRUE)

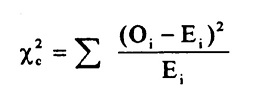
print(p1, position = c(0.5, 0.05, 1, 0.5))

****

**Functions for statistical tests and sampling**

Statistical tests are important for determining the correlation between two (or more) variables and what is their direction of correlation (positive, neutral or negative). Statistically speaking, the correlation is a measure of the strength of the association between two variables and their direction. RevoScaleR package supports calculation of Chi-square, Fischer and Kendall rank correlation. Based on the type of variables, one can distinguish between Kendall, Spearman or Pearson correlation coefficient.

For Chi-Square test, we will be using rxChiSquareTest() function, that uses contingency table and see if two variables are related. A small chi-square test statistic mean that observed data fits your expected data very well, denoting there is a correlation, respectively. Formula for calculating chi-square is:



Prior to calculating this statistical independence test, we must have data in the xCrossTab or xCube format. Therefore, the T-SQL query will need to generate the crosstabulations first in order to calculate the chi-square coefficient.

Chi-Square generated on two categorical variables:

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

library(RevoScaleR)

df\_sql <- InputDataSet

df\_sql\_r <- rxFactors(inData = df\_sql, sortLevels = TRUE,factorInfo = list(MS = list(levels = c("M","S"), otherLevel=NULL, varName="MaritalStatus")))

df\_sql\_r$Occupation <- as.factor(df\_sql\_r$Occupation)

df\_sql\_r$MS <- df\_sql\_r$MS

testData <- data.frame(Occupation = df\_sql\_r$Occupation, Status=df\_sql\_r$MS)

d <- rxCrossTabs(~Occupation:Status, testData, returnXtabs = TRUE)

chi\_q <- rxChiSquaredTest(d)

#results

xs <- chi\_q$''X-squared''

p <- chi\_q$''p-value''

OutputDataSet <- data.frame(xs,p)'

,@input\_data\_1 = N'

SELECT \* FROM [Sales].[vPersonDemographics] WHERE [DateFirstPurchase] IS NOT NULL'

WITH RESULT SETS

((

Chi\_square\_value NVARCHAR(100)

,Stat\_significance NVARCHAR(100)

));

With results returned:

Chi-squared test of independence between Occupation and Status

X-squared df p-value

588.2861 4 5.312913e-126

With Kendall tau one can calculate the correlation between the ranks and the result of the above correlation using the R code:

rxKendallCor(d, type = "b")

With the results:

estimate 1 p-value

-0.05179647 0

HA: two.sided

Same principle can be used in T-SQL query:

EXEC sp\_execute\_external\_script

@language = N'R'

,@script = N'

library(RevoScaleR)

df\_sql <- InputDataSet

df\_sql\_r <- rxFactors(inData = df\_sql, factorInfo = list(MS = list(levels = c("M","S"), otherLevel=NULL, varName="MaritalStatus")))

df\_sql\_r$Occupation <- as.factor(df\_sql\_r$Occupation)

df\_sql\_r$MS <- df\_sql\_r$MS

testData <- data.frame(Occupation = df\_sql\_r$Occupation, Status=df\_sql\_r$MS)

d <- rxCrossTabs(~Occupation:Status, testData, returnXtabs = TRUE)

ken <- rxKendallCor(d, type = "b")

k<- ken$`estimate 1`

p<- ken$`p-value`

#results

OutputDataSet <- data.frame(k,p)'

,@input\_data\_1 = N'

SELECT \* FROM [Sales].[vPersonDemographics] WHERE [DateFirstPurchase] IS NOT NULL'

WITH RESULT SETS

((

Kendall\_value NVARCHAR(100)

,Stat\_significance NVARCHAR(100)

));

Many other principles can be used for calculating the correlations among the variables. But this will be out of the scope of this book and therefore, we have focused only on the necessary one.

**Functions for predictive modeling**

In this part of the chapter we will briefly cover the predictive modeling, as the chapter 06 will detail more in details with this.

**Summary**

This chapter has covered the important functions (amongst many others) for data manipulation and data wrangling. These steps are absolutely and utterly most important for understanding the structure of the dataset, the content of the dataset and how the data are distributed; mainly understanding the frequencies, descriptive statistics and also some statistical sampling, as well as statistical correlations.

None of these steps must not be done prior to data cleaning and data merging. Cleaning the data is of highest importance, as outliers might bring the sensitive data (or any kind of data) to strange or false conclusions, as well it might sway the results in unwanted direction. So paying high importance to these steps by using the powerful rx- functions (or classes) should be the task of every data engineer, data wrangles as well as data scientist.