Microsoft

RevoScaleR Teradata Getting Started Guide

The correct bibliographic citation for this manual is as follows: Microsoft Corporation. 2016. *RevoScaleR Teradata Getting Started Guide*. Microsoft Corporation, Redmond, WA.

RevoScaleR Teradata Getting Started Guide

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
U.S.A.

Revised on December 29, 2015

Table of Contents

1 Overview

This guide is an introduction to high-performance 'big data' analytics for ***Teradata*** using ***RevoScaleR***, an R package included with ***Microsoft R Services***. ***Teradata Platforms*** running the ***Teradata Database*** provide high-performance, high-capacity data storage capabilities that are a great match for the ***RevoScaleR*** high-performance analytics.

There are three key components to running ***RevoScaleR*** high performance analytics:

- The name and arguments of the analysis function: What analysis do you want to perform?
- The specification of your data source(s): Where is your data and what are its characteristics? And, if you are creating new data, for example with scoring, where does the new data go?
- The specification for your compute context: Where do you want to perform the computations?

RevoScaleR provides a variety of data sources and compute contexts that can be used with the high performance analytics functions. This guide focuses on analyzing Teradata data in-database. That is, the data is located in a Teradata database and the computations are performed at the location of the data. We also consider a second use case: data is extracted from the Teradata database, taking advantage of the Teradata Parallel Transporter (TPT), and computations are performed on a computer alongside the Teradata Platform.

More detailed examples of using ***RevoScaleR*** can be found in the following guides included with ***Microsoft R Services***:

- *RevoScaleR Getting Started Guide* (RevoScaleR_Getting_Started.pdf)
- *RevoScaleR User's Guide* (RevoScaleR_Users_Guide.pdf)
- *RevoScaleR Distributed Computing Guide* (RevoScaleR_Distributed_Computing.pdf)
- *RevoScaleR ODBC Data Import Guide* (RevoScaleR_ODBC.pdf)

On Windows, the ***R Productivity Environment*** integrated development environment is included with ***Microsoft R Services***. It has its own *Getting Started Guide*.

For information on other distributed computing compute contexts, see:

- *RevoScaleR Hadoop Getting Started Guide* (RevoScaleR_Hadoop_Getting_Started.pdf)
- *RevoScaleR HPC Server Getting Started Guide* (RevoScaleR_HPC_Server_Getting_Started.pdf)
- *RevoScaleR LSF Cluster Getting Started Guide* (RevoScaleR_LSF_Cluster_Getting_Started.pdf)

2 Installation

The ***RevoScaleR*** package is installed as part of ***Microsoft R Services*** on Windows, Red Hat Enterprise Linux (RHEL) and SUSE Linux Enterprise Server (SLES). The package is automatically loaded when you start ***Microsoft R Services***. On your client machine, you will need:

- ***Microsoft R Services*** installed.
- Teradata ODBC drivers and the Teradata Parallel Transporter installed from the Teradata 14.10 client installer, with a high-speed connection (100 Mbps or above) to a Teradata Database running version 14.10

If you plan to do in-database computations in ***Teradata***, you will need ***Microsoft R Services*** installed on all of the nodes of the ***Teradata Platform***.

If you plan to perform large computations alongside ***Teradata***, ***Microsoft R Services*** should be installed on a server or powerful workstation with access to your ***Teradata Database*** with the following specification:

- Red Hat Enterprise Linux 5.x or 6.x, SLES 11, or a recent version of Windows (post-XP)
- At least 4 fast cores
- At least 8 GB but preferably 32 GB of RAM
- A fast hard drive with at least 100 GB available– enough disk space to temporarily store the data being analyzed if desired
- Teradata ODBC drivers and the Teradata Parallel Transporter installed from the Teradata 14.10 client installer, with a high-speed connection (100 Mbps or above) to a Teradata Database running version 14.10 [If you are using Teradata 14.0, the in-Teradata computation functionality will not be available.]

More detailed descriptions of the Teradata component requirements are provided in the *Microsoft R Services Client Installation Guide for Teradata*. Note that the R Productivity Environment client (an Integrated Development Environment for R) runs on Windows workstations only.

3 Setting Up the Sample Data

3.1 The Sample *ccFraud* Data

To follow the examples in this guide, you will need the following comma-delimited text data files to load into your Teradata database:

```
ccFraud.csv
ccFraudScore.csv
```

These data files each have 10 million rows, and should be put in tables called `ccFraud10` and `ccFraudScore10`, respectively. These data files are available online. We recommend that you create a RevoTestDB database to hold the data table.

3.2 Loading Your Data into the Teradata Database

You can use the 'fastload' Teradata command to load the data sets into your data base. You can find sample scripts for doing this in Appendix I, and online with the sample data files. You will need to edit the sample scripts to provide appropriate logon information for your site. Check with your system administrator to see of the data has already been loaded into your database. You will also see an example later in this guide of loading data into your Teradata data base from R using the `rxDataStep` function.

4 Using a Teradata Data Source and ComputeContext

The ***RevoScaleR*** package provides a framework for quickly writing start-to-finish, scalable R code for data analysis. Even if you are relatively new to R, you can get started with just a few basic functions. In this guide we'll be focusing on analyzing data that is located in a ***Teradata Database***. The first step is to create a *data source* R object that contains information about the data that will be analyzed.

4.1 Creating an RxTeradata Data Source

To create a Teradata data source for use in RevoScaleR, you will need basic information about your database connection. The connection string can contain information about your driver, your Teradata ID, your database name, your user ID, and your password. Modify the code below to specify the connection string appropriate to your setup:

```
tdConnString <- "DRIVER=Teradata;DBCNAME=machineNameOrIP;
      DATABASE=RevoTestDB;UID=myUserID;PWD=myPassword;"
```

Then we need an SQL query statement to identify the data we want to use. We'll start with all of the variables in the *ccFraud10* data, a simulated data set with 10 million observations. Modify the code below to specify the correct database name:

```
tdQuery <- "SELECT * FROM ccFraud10"
```

We use this information to create an RxTeradata data source object:

```
teradataDS <- RxTeradata(connectionString = tdConnString,
      sqlQuery = tdQuery, rowsPerRead = 50000)
```

Note that we have also specified `rowsPerRead`. This parameter is important for handling memory usage and efficient computations. Most of the ***RevoScaleR*** analysis functions process data in chunks and accumulate intermediate results, returning the final computations after all of the data has been read. The `rowsPerRead` parameter controls how many rows of data are read into each chunk for processing. If it is too large, you may encounter slow-downs because you don't have enough memory to efficiently process such a large chunk of data. On some systems, setting `rowsPerRead` to too small a value can also provide slower performance. You may want to experiment with this setting on your system.

4.2 Extracting Basic Information about Your Data

The Teradata data source can be used as the 'data' argument in RevoScaleR functions. For example, to get basic information about the data and variables, enter:

```
rxGetVarInfo(data = teradataDS)
```

You should see the following information:

```
Var 1: custID, Type: integer
Var 2: gender, Type: integer
Var 3: state, Type: integer
Var 4: cardholder, Type: integer
Var 5: balance, Type: integer
Var 6: numTrans, Type: integer
Var 7: numIntlTrans, Type: integer
Var 8: creditLine, Type: integer
Var 9: fraudRisk, Type: integer
```

4.3 Specifying Column Information in Your Data Source

All of the variables in our data set are stored as integers in the data base, but some of them actually represent categorical data – called factor variables in R. We can specify this information in the data source, and have them automatically converted to factors when analyzed or imported. The variable *state* represents the 50 states plus the District of Columbia. We can specify the state abbreviations as follows to be used as labels for that variable:

```
stateAbb <- c("AK", "AL", "AR", "AZ", "CA", "CO", "CT", "DC",
    "DE", "FL", "GA", "HI","IA", "ID", "IL", "IN", "KS", "KY", "LA",
    "MA", "MD", "ME", "MI", "MN", "MO", "MS", "MT", "NB", "NC", "ND",
    "NH", "NJ", "NM", "NV", "NY", "OH", "OK", "OR", "PA", "RI","SC",
    "SD", "TN", "TX", "UT", "VA", "VT", "WA", "WI", "WV", "WY")
```

Now we'll create a column information object specifying the mapping of the existing integer values to categorical levels in order to create factor variables for *gender*, *cardholder*, and *state*:

```
ccColInfo <- list(
    gender = list(
        type = "factor",
        levels = c("1", "2"),
        newLevels = c("Male", "Female")),
    cardholder = list(
          type = "factor",
          levels = c("1", "2"),
          newLevels = c("Principal", "Secondary")),
      state = list(
          type = "factor",
          levels = as.character(1:51),
          newLevels = stateAbb)
      )
```

Next we can recreate a Teradata data source, adding the column information:

```
teradataDS <- RxTeradata(connectionString = tdConnString,
    sqlQuery = tdQuery, colInfo = ccColInfo, rowsPerRead = 50000)
```

If we get the variable information for the new data source, we can see that the three variables specified in *colInfo* are now treated as factors:

```
rxGetVarInfo(data = teradataDS)

Var 1: custID, Type: integer
Var 2: gender
       2 factor levels: Male Female
Var 3: state
       51 factor levels: AK AL AR AZ CA ... VT WA WI WV WY
Var 4: cardholder
       2 factor levels: Principal Secondary
Var 5: balance, Type: integer
Var 6: numTrans, Type: integer
Var 7: numIntlTrans, Type: integer
Var 8: creditLine, Type: integer
Var 9: fraudRisk, Type: integer
```

Teradata has a limit of 30 bytes for table and column names, and sometimes creates additional tables with six-character suffixes, so you restrict your table and variable names to no more than 24 characters.

4.4 Creating an RxInTeradata Compute Context

Since we want to perform ***RevoScaleR*** analytic computations in-database, the next step is to create an `RxInTeradata` compute context. You will need basic information about your Teradata platform. Since the computations are tied to the database, a connection string is required for the `RxInTeradata` compute context. As when specifying an `RxTeradata` data source, the connection string can contain information about your driver, your Teradata ID, your database name, your user ID, and your password. If you have not done so already, modify the code below to specify the connection string appropriate to your setup:

```
tdConnString <- "DRIVER=Teradata;DBCNAME=machineNameOrIP;
      DATABASE=RevoTestDB;UID=myUserID;PWD=myPassword;"
```

Although the data source and compute context have overlapping information (and similar names), be sure to distinguish between them. The data source (`RxTeradata`) tells us where the data is; the compute context (`RxInTeradata`) tells us where the computations are to take place. Note that the compute context only determines where the ***RevoScaleR*** high-performance analytics computations take place; it does not affect other standard R computations that you are performing on the client machine.

The compute context also requires information about your shared directory, both locally and remotely, and the path where ***Microsoft R Services*** is installed on each of the nodes of the ***Teradata*** platform. Specify the appropriate information here:

```
tdShareDir <- paste("c:\\AllShare\\", Sys.getenv("USERNAME"), sep="")
tdRemoteShareDir <- "/tmp/revoJobs"
tdRevoPath <- "/usr/lib64/MRO-for-MRS-8.0/R-3.2.2/lib64/R"
```

Be sure that the `tdShareDir` specified exists on your client machine. To create the directory from R, you can run:

```
dir.create(tdShareDir, recursive = TRUE)
```

Last, we need to specify some information about how we want output handled. We'll request that our R session wait for the results, and not return console ouput from the in-database computations:

```
tdWait <- TRUE
tdConsoleOutput <- FALSE
```

We use all this information to create our `RxInTeradata` compute context object:

```
tdCompute <- RxInTeradata(
    connectionString = tdConnString,
    shareDir = tdShareDir,
    remoteShareDir = tdRemoteShareDir,
    revoPath = tdRevoPath,
    wait = tdWait,
    consoleOutput = tdConsoleOutput)
```

4.5 Getting Information about Your RxInTeradata Compute Context

You can retrieve basic information about your Teradata platform using the `rxGetNodeInfo` function. Your platform has at least one PE (Parsing Engine). When RevoScaleR analytics are submitted to the Teradata platform, it is the Parsing Engine that receives and distributes instructions. Each AMP(Access Module Processor) is connected to a disk, and has read/write access. When it receives an instruction from the PE, it retrieves its piece of the data. All of the AMPs work in parallel. The `rxGetNodeInfo` function will send a small task to each of the AMPS, and give a summary for each node. If you have a large platform, this may take some time:

```
rxGetNodeInfo(tdCompute)
```

4.6 Troubleshooting the RxInTeradata Compute Context

If you encounter difficulties while using the `RxInTeradata` context, you may find it convenient to turn on run-time tracing. You can do this by passing the arguments `traceEnabled` and `traceLevel` to the `RxInTeradata` constructor. Set the `traceLevel` to 7 to show all tracing information:

```
tdComputeTrace <- RxInTeradata(
    connectionString = tdConnString,
    shareDir = tdShareDir,
    remoteShareDir = tdRemoteShareDir,
    revoPath = tdRevoPath,
        wait = tdWait,
        consoleOutput = tdConsoleOutput,
        traceEnabled = TRUE,
        traceLevel = 7)
```

5 High-Performance In-Database Analytics in Teradata

5.1 Computing Summary Statistics in Teradata

Let’s start by computing summary statistics on our `teradataDS` data source, performing the computations in-database. The next step is to change the active compute context. When you run `rxSetComputeContext` with `tdCompute` as its argument, all subsequent RevoScaleR HPA computations will take place in the in-database in Teradata until the compute context is switched back to a local compute context.

```
# Set the compute context to compute in Teradata
rxSetComputeContext(tdCompute)
```

We can use the high-performance `rxSummary` function to compute summary statistics for several of the variables. The formula used is a standard R formula:

```
rxSummary(formula = ~gender + balance + numTrans + numIntlTrans +
      creditLine, data = teradataDS)
Call:
rxSummary(formula = ~gender + balance + numTrans + numIntlTrans +
    creditLine, data = teradataDS)

Summary Statistics Results for: ~gender + balance + numTrans +
    numIntlTrans + creditLine
Data: teradataDS (RxTeradata Data Source)
Number of valid observations: 1e+07

 Name         Mean        StdDev      Min Max   ValidObs MissingObs
 balance      4109.919919 3996.847310 0   41485 1e+07    0
 numTrans       28.935187   26.553781 0     100 1e+07    0
 numIntlTrans    4.047190    8.602970 0      60 1e+07    0
 creditLine      9.134469    9.641974 1      75 1e+07    0

Category Counts for gender
Number of categories: 2
Number of valid observations: 1e+07
Number of missing observations: 0

 gender Counts
 Male   6178231
 Female 3821769
```

You can see that the `ccFraud10` data set has 10 million observations with no missing data.

To set the compute context back to our client machine, enter:

```
# Set the compute context to compute locally
rxSetComputeContext ("local")
```

5.2 Refining the RxTeradata Data Source

The computed summary statistics provide useful information about our data that can be put in the data source for use in further computations. For example, RevoScaleR uses minimum and maximum values in computing histograms, and can very efficiently convert integer data to categorical factor data "on-the-fly" using the F function. You can include the specification of high and low values in an `RxTeradata` data source. We can add this information for `balance`, `numTrans, numIntlTrans`, and `creditLine` to the `colInfo` used to create the data source:

```
ccColInfo <- list(
    gender = list(
        type = "factor",
        levels = c("1", "2"),
        newLevels = c("Male", "Female")),
    cardholder = list(
          type = "factor",
          levels = c("1", "2"),
          newLevels = c("Principal", "Secondary")),
      state = list(
          type = "factor",
          levels = as.character(1:51),
          newLevels = stateAbb),
      balance = list(low = 0, high = 41485),
      numTrans = list(low = 0, high = 100),
      numIntlTrans = list(low = 0, high = 60),
      creditLine = list(low = 1, high = 75)
      )
```

Again, recreate a Teradata data source, adding the additional column information:

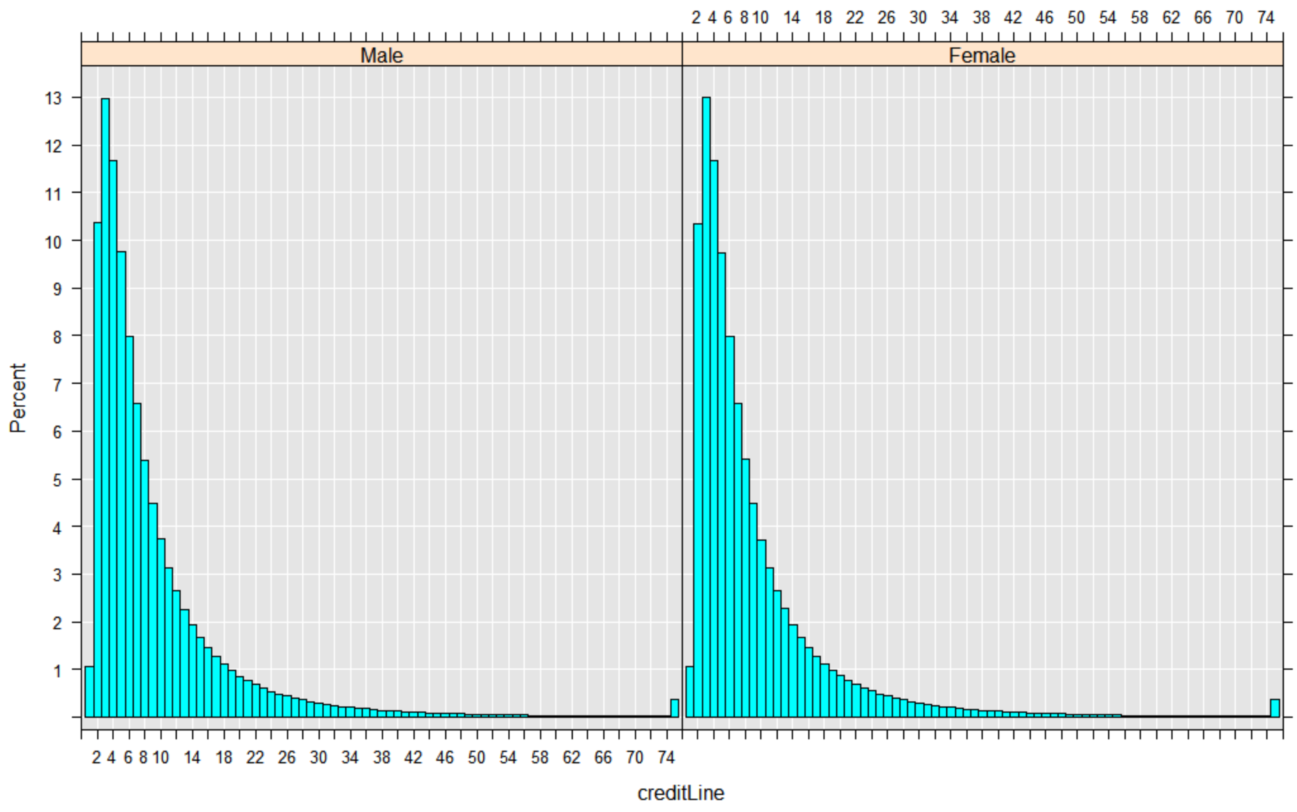
```
teradataDS <- RxTeradata(connectionString = tdConnString,
    sqlQuery = tdQuery, colInfo = ccColInfo, rowsPerRead = 50000)
```

5.3 Visualizing Your Data Using *rxHistogram* and *rxCube*

The `rxHistogram` function will show us the distribution of any of the variables in our data set. For example, let's look at *creditLine* by *gender*. First we'll set the compute context back to `tdCompute` so that all of our analytics computations will be performed in Teradata rather than alongside:`rxSetComputeContext(tdCompute)`

Next we'll call `rxHistogram`. Internally, `rxHistogram` will call the ***RevoScaleR*** `rxCube` analytics function, which will perform the required computations in-database in Teradata and return the results to your local workstation for plotting:

```
rxHistogram(~creditLine|gender, data = teradataDS,
      histType = "Percent")
```



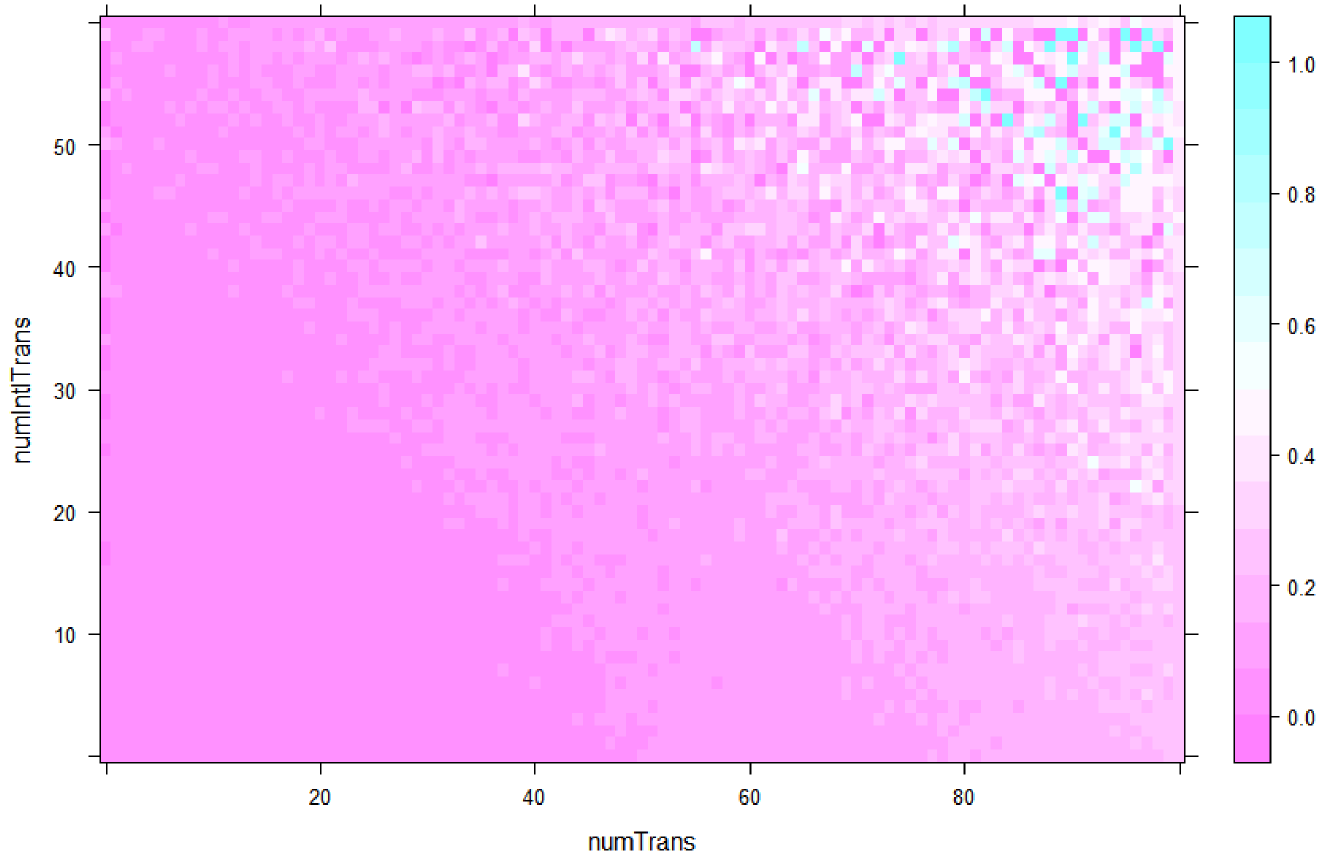
We can also call the `rxCube` function directly and use the results with one of many of R's plotting functions. For example, `rxCube` can compute group means, so we can compute the mean of *fraudRisk* for every combination of *numTrans* and *numIntlTrans*. We'll use the `F()` notation to have integer variables treated as categorical variables (with a level for each integer value). The low and high levels specified in `colInfo` will automatically be used.

```
cube1 <- rxCube(fraudRisk~F(numTrans):F(numIntlTrans),
      data = teradataDS)
```

The `rxResultsDF` function will convert the results of the `rxCube` function into a data frame that can easily be used in one of R's standard plotting functions. Here we'll create a heat map using the *levelplot* function from the `lattice` package:

```
cubePlot <- rxResultsDF(cube1)
levelplot(fraudRisk~numTrans*numIntlTrans, data = cubePlot)
```

We can see that the risk of fraud increases with both the number of transactions and the number of international transactions:



5.4 Analyzing Your Data with *rxLinMod*

Linear models are a work horse of predictive analytics. `RevoScaleR` provides a high performance, scalable algorithm. As a simple example, let's estimate a linear model with `balance` as the dependent variable and `gender` and `creditLine` as independent variables:

```
linModObj <- rxLinMod(balance ~ gender + creditLine,
      data = teradataDS)
```

As long as we have not changed the compute context, the computations will be performed in-database in Teradata. The function will return an object containing the model results to your local workstation. We can look at a summary of the results using the standard R `summary` function:

```
summary(linModObj)

Call:
rxLinMod(formula = balance ~ gender + creditLine, data = teradataDS)

Linear Regression Results for: balance ~ gender + creditLine
Data: teradataDS (RxTeradata Data Source)
```

```
Dependent variable(s): balance
Total independent variables: 4 (Including number dropped: 1)
Number of valid observations: 1e+07
Number of missing observations: 0

Coefficients: (1 not defined because of singularities)
               Estimate Std. Error  t value Pr(>|t|)
(Intercept)   3109.8508     2.2856 1360.647 2.22e-16 ***
gender=Male      3.2671     2.5091    1.302    0.193
gender=Female   Dropped    Dropped  Dropped  Dropped
creditLine     109.2620     0.1264  864.080 2.22e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3856 on 9999997 degrees of freedom
Multiple R-squared: 0.06948
Adjusted R-squared: 0.06948
F-statistic: 3.733e+05 on 2 and 9999997 DF,  p-value: < 2.2e-16
Condition number: 1.0001
```

5.5 Analyzing Your Data with *rxLogit*

Now, let's estimate a logistic regression on whether or not an individual is a fraud risk. We'll continue to use the same compute context and data source, and specify a large model – 60 independent variables, including the 3 dummy variables that are dropped. Note that in R (and RevoScaleR) every level of a categorical factor variable is automatically treated as a separate dummy variable:

```
logitObj <- rxLogit(fraudRisk ~ state + gender + cardholder + balance +
    numTrans + numIntlTrans + creditLine, data = teradataDS,
    dropFirst = TRUE)
summary(logitObj)
```

We get the following output:

```
Call:
rxLogit(formula = fraudRisk ~ state + gender + cardholder + balance +
    numTrans + numIntlTrans + creditLine, data = teradataDS,
    dropFirst = TRUE)

Logistic Regression Results for: fraudRisk ~ state + gender +
    cardholder + balance + numTrans + numIntlTrans + creditLine
Data: teradataDS (RxTeradata Data Source)
Dependent variable(s): fraudRisk
Total independent variables: 60 (Including number dropped: 3)
Number of valid observations: 1e+07
-2*LogLikelihood: 2101682.8022 (Residual deviance on 9999943 degrees of freedom)

Coefficients:
                     Estimate Std. Error  z value Pr(>|z|)
(Intercept)         -8.524e+00  3.640e-02 -234.163 2.22e-16 ***
state=AK               Dropped    Dropped  Dropped  Dropped
state=AL            -6.127e-01  3.840e-02  -15.956 2.22e-16 ***
```

```
state=AR             -1.116e+00  4.100e-02  -27.213 2.22e-16 ***
state=AZ             -1.932e-01  3.749e-02   -5.153 2.56e-07 ***
state=CA             -1.376e-01  3.591e-02   -3.831 0.000127 ***
state=CO             -3.314e-01  3.794e-02   -8.735 2.22e-16 ***
state=CT             -6.332e-01  3.933e-02  -16.101 2.22e-16 ***
state=DC             -7.717e-01  5.530e-02  -13.954 2.22e-16 ***
state=DE             -4.316e-02  4.619e-02   -0.934 0.350144
state=FL             -2.833e-01  3.626e-02   -7.814 2.22e-16 ***
state=GA              1.297e-02  3.676e-02    0.353 0.724215
state=HI             -6.222e-01  4.395e-02  -14.159 2.22e-16 ***
state=IA             -1.284e+00  4.093e-02  -31.386 2.22e-16 ***
state=ID             -1.127e+00  4.427e-02  -25.456 2.22e-16 ***
state=IL             -7.355e-01  3.681e-02  -19.978 2.22e-16 ***
state=IN             -1.124e+00  3.842e-02  -29.261 2.22e-16 ***
state=KS             -9.614e-01  4.125e-02  -23.307 2.22e-16 ***
state=KY             -8.121e-01  3.908e-02  -20.778 2.22e-16 ***
state=LA             -1.231e+00  3.950e-02  -31.173 2.22e-16 ***
state=MA             -8.426e-01  3.810e-02  -22.115 2.22e-16 ***
state=MD             -2.378e-01  3.752e-02   -6.338 2.33e-10 ***
state=ME             -7.322e-01  4.625e-02  -15.829 2.22e-16 ***
state=MI             -1.250e+00  3.763e-02  -33.218 2.22e-16 ***
state=MN             -1.166e+00  3.882e-02  -30.030 2.22e-16 ***
state=MO             -7.618e-01  3.802e-02  -20.036 2.22e-16 ***
state=MS             -1.090e+00  4.096e-02  -26.601 2.22e-16 ***
state=MT             -1.049e+00  5.116e-02  -20.499 2.22e-16 ***
state=NB             -6.324e-01  4.270e-02  -14.808 2.22e-16 ***
state=NC             -1.135e+00  3.754e-02  -30.222 2.22e-16 ***
state=ND             -9.667e-01  5.675e-02  -17.033 2.22e-16 ***
state=NH             -9.966e-01  4.778e-02  -20.860 2.22e-16 ***
state=NJ             -1.227e+00  3.775e-02  -32.488 2.22e-16 ***
state=NM             -5.825e-01  4.092e-02  -14.235 2.22e-16 ***
state=NV             -2.222e-01  3.968e-02   -5.601 2.13e-08 ***
state=NY             -1.235e+00  3.663e-02  -33.702 2.22e-16 ***
state=OH             -6.004e-01  3.687e-02  -16.285 2.22e-16 ***
state=OK             -1.043e+00  4.005e-02  -26.037 2.22e-16 ***
state=OR             -1.284e+00  4.056e-02  -31.649 2.22e-16 ***
state=PA             -5.802e-01  3.673e-02  -15.797 2.22e-16 ***
state=RI             -7.197e-01  4.920e-02  -14.627 2.22e-16 ***
state=SC             -7.759e-01  3.878e-02  -20.008 2.22e-16 ***
state=SD             -8.295e-01  5.541e-02  -14.970 2.22e-16 ***
state=TN             -1.258e+00  3.861e-02  -32.589 2.22e-16 ***
state=TX             -6.103e-01  3.618e-02  -16.870 2.22e-16 ***
state=UT             -5.573e-01  4.036e-02  -13.808 2.22e-16 ***
state=VA             -7.069e-01  3.749e-02  -18.857 2.22e-16 ***
state=VT             -1.251e+00  5.944e-02  -21.051 2.22e-16 ***
state=WA             -1.297e-01  3.743e-02   -3.465 0.000530 ***
state=WI             -9.027e-01  3.843e-02  -23.488 2.22e-16 ***
state=WV             -5.708e-01  4.245e-02  -13.448 2.22e-16 ***
state=WY             -1.153e+00  5.769e-02  -19.987 2.22e-16 ***
gender=Male             Dropped    Dropped  Dropped  Dropped
gender=Female         6.131e-01  3.754e-03  163.306 2.22e-16 ***
cardholder=Principal    Dropped    Dropped  Dropped  Dropped
cardholder=Secondary  4.785e-01  9.847e-03   48.588 2.22e-16 ***
balance               3.828e-04  4.660e-07  821.534 2.22e-16 ***
```

```
numTrans              4.747e-02  6.649e-05  713.894 2.22e-16 ***
numIntlTrans          3.021e-02  1.776e-04  170.097 2.22e-16 ***
creditLine            9.491e-02  1.416e-04  670.512 2.22e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Condition number of final variance-covariance matrix: 2922.044
Number of iterations: 8
```

5.6 Scoring a Data Set with Your Model

We can use the estimated logistic regression model to create scores for another data set with the same independent variables. We will need to specify two new data sources: the new input data set to be scored, and a new table in the Teradata Database for the results.

```
tdQuery <- "SELECT * FROM ccFraudScore10"
teradataScoreDS <- RxTeradata(connectionString = tdConnString,
    sqlQuery = tdQuery, colInfo = ccColInfo, rowsPerRead = 50000)


teradataOutDS <- RxTeradata(table = "ccScoreOutput",
    connectionString = tdConnString, rowsPerRead = 50000 )
```

Now we set our compute context. We'll also make sure that the output table doesn't exist by making a call to *rxTeradataDropTable*.

```
rxSetComputeContext(tdCompute)
if (rxTeradataTableExists("ccScoreOutput"))
    rxTeradataDropTable("ccScoreOutput")
```

Now we can use the *rxPredict* function to score. We will set *writeModelVars* to TRUE to have all of the variables used in the estimation included in the new table. The new variable containing the scores will be named *ccFraudLogitScore*. We have a choice of having our predictions calculated on the scale of the response variable or the underlying 'link' function. Here we choose the 'link' function, so that the predictions will be on a logistic scale.

```
rxPredict(modelObject = logitObj,
      data = teradataScoreDS,
      outData = teradataOutDS,
      predVarNames = "ccFraudLogitScore",
      type = "link",
      writeModelVars = TRUE,
      overwrite = TRUE)
```

To add additional variables to our output predictions, use the *extraVarsToWrite* argument. For example, we can include the variable *custID* from our scoring data table in our table of predictions as follows :

```
if (rxTeradataTableExists("ccScoreOutput"))
    rxTeradataDropTable("ccScoreOutput")
```

```
rxPredict(modelObject = logitObj,
      data = teradataScoreDS,
      outData = teradataOutDS,
      predVarNames = "ccFraudLogitScore",
      type = "link",
      writeModelVars = TRUE,
      extraVarsToWrite = "custID",
      overwrite = TRUE)
```

After the new table has been created, we can compute and display a histogram of the 10 million predicted scores. The computations will be faster if we pre-specify the low and high values. We can get this information from the data base using a special data source with *rxImport*. (Note that the *RxOdbcData* data source type can be used with a Teradata data base. It is typically faster for small queries.)

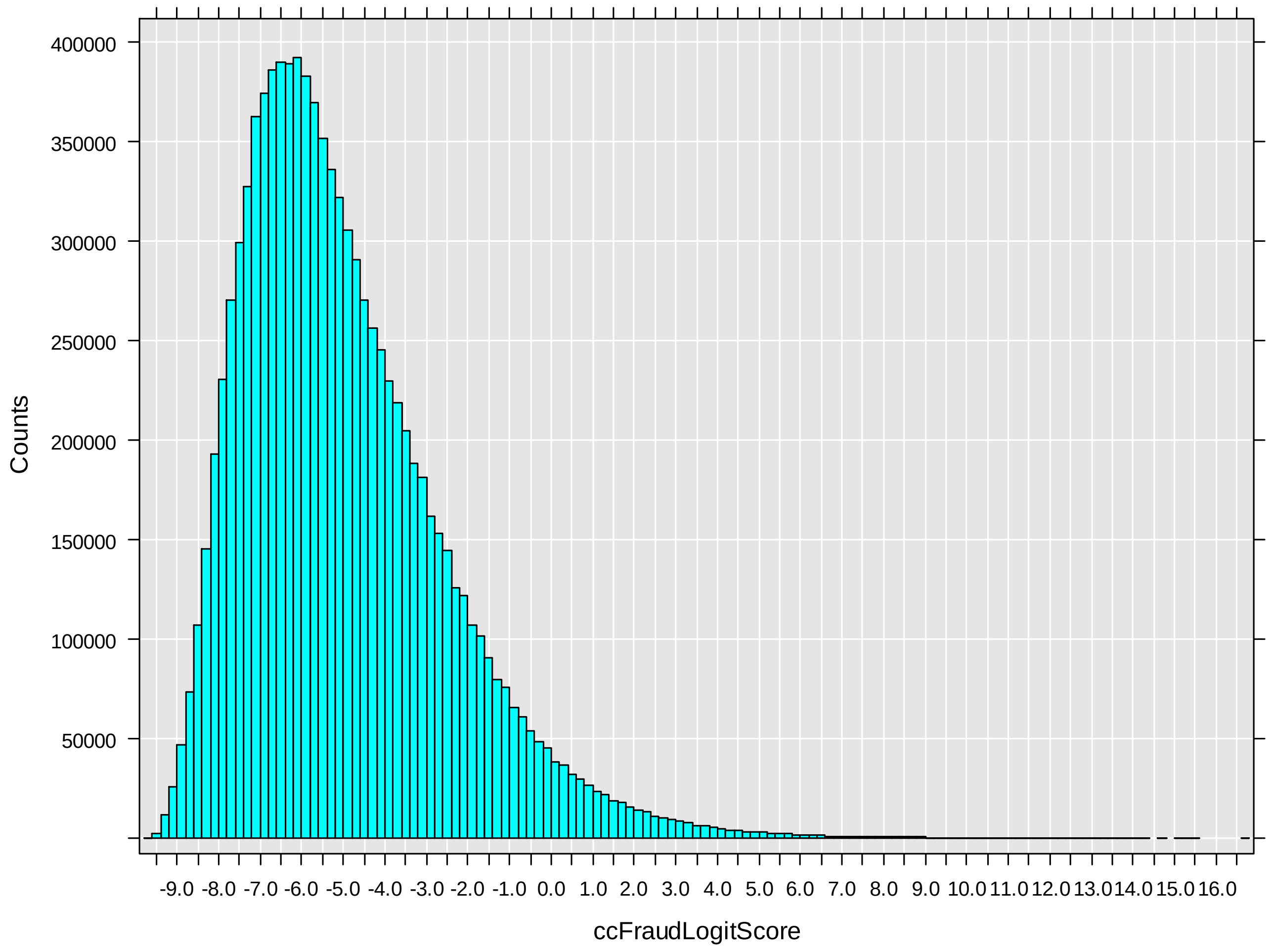
```
tdMinMax <- RxOdbcData(sqlQuery = paste(
      "SELECT MIN(ccFraudLogitScore),",
      "MAX(ccFraudLogitScore) FROM ccScoreOutput"),
      connectionString = tdConnString)
minMaxVals <- rxImport(tdMinMax)
minMaxVals <- as.vector(unlist(minMaxVals))
```

Now we'll create our data source:

```
teradataScoreDS <- RxTeradata(sqlQuery =
      "Select ccFraudLogitScore FROM ccScoreOutput",
      connectionString = tdConnString, rowsPerRead = 50000,
      colInfo = list(ccFraudLogitScore = list(
            low = floor(minMaxVals[1]),
            high = ceiling(minMaxVals[2]))))
```

Then we compute and display the histogram:

```
rxSetComputeContext(tdCompute)
rxHistogram(~ccFraudLogitScore, data = teradataScoreDS)
```


400000
350000
300000
250000
200000
150000
100000
50000
Counts
-9.0 -8.0 -7.0 -6.0 -5.0 -4.0 -3.0 -2.0 -1.0 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0 13.0 14.0 15.0 16.0
ccFraudLogitScore

6 Using rxDataStep and rxImport

6.1 Using rxDataStep to Transform Variables

The `rxDataStep` function will process data a chunk at a time, reading from one data source and writing to another. In this example, we'll use a function in another R package. The `boot` package is 'recommended' package that is included with every distribution of R, but is not loaded automatically on start-up. It contains a function `inv.logit` that computes the inverse of a logit; that is, converts a logit back to a probability on the [0,1] scale. (Note that we could have gotten predictions in this scale by setting `type="response"` in our call to `rxPredict`.) We'd like all of the variables in our `ccScoreOutput` table to be put in the new table, in addition to the newly created variable. So we specify our input and output data sources as follows:

```
teradataScoreDS <- RxTeradata(
    sqlQuery =  "Select * FROM ccScoreOutput",
    connectionString = tdConnString, rowsPerRead = 50000 )

teradataOutDS2 <- RxTeradata(table = " ccScoreOutput2",
      connectionString = tdConnString, rowsPerRead = 50000)

rxSetComputeContext(tdCompute)
if (rxTeradataTableExists("ccScoreOutput2"))
    rxTeradataDropTable("ccScoreOutput2")
```

Now we call the `rxDataStep` function, specifying the transforms we want in a list. We also specifying the additional R packages that are needed to perform the transformations. Note that these packages must be pre-installed on the nodes of your Teradata platform.

```
rxDataStep(inData = teradataScoreDS, outFile = teradataOutDS2,
      transforms = list(ccFraudProb = inv.logit(ccFraudLogitScore)),
      transformPackages = "boot", overwrite = TRUE)
```

We can now look at basic variable information for the new data set.

```
rxGetVarInfo(teradataOutDS2)
Var 1: ccFraudLogitScore, Type: numeric
Var 2: custID, Type: integer
Var 3: state, Type: character
Var 4: gender, Type: character
Var 5: cardholder, Type: character
Var 6: balance, Type: integer
Var 7: numTrans, Type: integer
Var 8: numIntlTrans, Type: integer
Var 9: creditLine, Type: integer
Var 10: ccFraudProb, Type: numeric
```

Notice that factor variables are written to the data base as character data. To use them as factors in subsequent analysis, use `colInfo` to specify the levels.

6.2 Using *rxImport* to Extract a Subsample into a Data Frame in Memory

If we want to examine high risk individuals in more detail, we can extract information about them into a data frame in memory, order them, and print information about those with the highest risk. Since we will be computing on our local computer, we can reset the compute context to local. We'll use the `sqlQuery` argument for the Teradata data source to specify the observations to select so that only the data of interest is extracted from the data base.

```
rxSetComputeContext("local")

teradataProbDS <- RxTeradata(
      sqlQuery = paste(
      "Select * FROM ccScoreOutput2",
      "WHERE (ccFraudProb > .99)"),
      connectionString = tdConnString)
highRisk <- rxImport(teradataProbDS)
```

Now we have the high risk observations in a data frame in memory. We can use any R functions to manipulate the data frame:

```
orderedHighRisk <- highRisk[order(-highRisk$ccFraudProb),]
row.names(orderedHighRisk) <- NULL  # Reset row numbers
head(orderedHighRisk)

  ccFraudLogitScore   custID state gender cardholder balance numTrans
1          16.60087 10957409    WA Female  Principal   39987       84
2          15.57692  6754290    FL   Male  Principal   31099       90
3          15.31091  5304850    AL   Male  Principal   33113      100
4          15.08932 10529850    WV   Male  Principal   32177      100
5          14.79001  3040086    CA Female  Principal   32562       66
6          14.22977  4422001    OK   Male  Secondary   33476      100
  numIntlTrans creditLine ccFraudProb
1            1         56   0.9999999
2           36         75   0.9999998
3            0         74   0.9999998
4            0         75   0.9999997
5            4         75   0.9999996
6            2         60   0.9999993
```

6.3 Using rxDataStep to Create a Teradata Table

When we are working in a local compute context, we have access to both local data files and the Teradata database via TPT. So, we can use rxDataStep to take a data set on our local system, perform any desired transformations, and create a new Teradata table. For example, one of the sample .xdf files shipped with RevoScaleR is AirlineDemoSmall.xdf.

```
rxSetComputeContext("local")
xdfAirDemo <- RxXdfData(file.path(rxGetOption("sampleDataDir"),
    "AirlineDemoSmall.xdf"))
rxGetVarInfo(xdfAirDemo)

Var 1: ArrDelay, Type: integer, Low/High: (-86, 1490)
Var 2: CRSDepTime, Type: numeric, Storage: float32, Low/High: (0.0167, 23.9833)
Var 3: DayOfWeek
       7 factor levels: Monday Tuesday Wednesday Thursday Friday Saturday Sunday
```

Let's put this data into a Teradata table, storing `DayOfWeek` as an integer with values from 1 to 7.

```
teradataAirDemo <- RxTeradata(table = " AirDemoSmallTest",
       connectionString = tdConnString)
if (rxTeradataTableExists("AirDemoSmallTest",
     connectionString = tdConnString))
     rxTeradataDropTable("AirDemoSmallTest",
     connectionString = tdConnString)

rxDataStep(inData = xdfAirDemo, outFile = teradataAirDemo,
       transforms = list(
             DayOfWeek = as.integer(DayOfWeek),
             rowNum = .rxStartRow : (.rxStartRow + .rxNumRows - 1)
       ), overwrite = TRUE
)
```

Now we can set our compute context back to in-Teradata, and look at summary statistics of the new table:

```
rxSetComputeContext(tdCompute)
teradataAirDemo <- RxTeradata(sqlQuery =
       "SELECT * FROM AirDemoSmallTest",
       connectionString = tdConnString,
       rowsPerRead = 50000,
       colInfo = list(DayOfWeek = list(type = "factor",
             levels = as.character(1:7))))

rxSummary(~., data = teradataAirDemo)
```

6.4 Performing Your Own 'Chunking' Analysis

The `rxDataStep` function also allows us to write our own 'chunking' analysis. Reading the data in chunks on multiple AMPS in Teradata, we can process each chunk of data using the R language, and write out summary results for each chunk into a common Teradata data source. Let's look at an example using the `table` function in R, which computes a contingency table. (If you actually have data sets to tabulate, use the `rxCrossTabs` or `rxCube` functions built into ***RevoScaleR***; this example is meant for instructional purposes only.)

The first step is to write a function to process each chunk of data. The data will automatically be fed into the function as a rectangular list of data columns. The function must also return a rectangular list of data columns (which a data frame is). In the example below, we'll be summarizing the input data and returning a data frame with a single row.

```
ProcessChunk <- function( dataList)
{
        # Convert the input list to a data frame and
        # call the 'table' function to compute the
        # contingency table
          chunkTable <- table(as.data.frame(dataList))

          # Convert table output to data frame with single row
          varNames <- names(chunkTable)
          varValues <- as.vector(chunkTable)
          dim(varValues) <- c(1, length(varNames))
          chunkDF <- as.data.frame(varValues)
          names(chunkDF) <- varNames

          # Return the data frame, which has a single row
          return( chunkDF )
    }
```

Next we'll set our active compute context to compute in-database, and setup our data source:

```
    rxSetComputeContext( tdCompute )
    tdQuery <-
    "select DayOfWeek from AirDemoSmallTest"
    inDataSource <- RxTeradata(sqlQuery = tdQuery,
          rowsPerRead = 50000,
          colInfo = list(DayOfWeek = list(type = "factor",
                levels = as.character(1:7))))
```

Now setup a data source to hold the intermediate results. Again, we'll "drop" the table if it exists.

```
    iroDataSource = RxTeradata(table = "iroResults",
          connectionString = tdConnString)
    if (rxTeradataTableExists(table = "iroResults",
          connectionString = tdConnString))
    {
        rxTeradataDropTable( table = "iroResults",
          connectionString = tdConnString)
    }
```

Next we'll call `rxDataStep` with our `ProcessChunk` function as the transformation function to use.

```
    rxDataStep( inData = inDataSource, outFile = iroDataSource,
                transformFunc = ProcessChunk,
                overwrite = TRUE)
```

To see our intermediate results for all of the chunks, we can import the data base table into memory:

```
    iroResults <- rxImport(iroDataSource)
    iroResults
```

To compute our final results, in this example we can just sum the columns:

```
finalResults <- colSums(iroResults)
finalResults
```

To remove the intermediate results table:

```
rxTeradataDropTable( table = " iroResults",
      connectionString = tdConnString)
```

7 Using rxDataStep for By-Group Analyses

When running in Teradata, it is also possible to control the data that is provided in each 'chunk' processed by `rxDataStep`. This allows us to perform by-group analyses, estimating a separate model for each group of observations defined by one or more categorical variables.

7.1 Creating a Simulated Data Set for By-Group Analysis

For example, let's create a simulated data set that has 100,000 observations and three variables: `SKU`, `INCOME`, and `SALES`. Our objective will be to estimate a model of `SALES` on `INCOME` for each `SKU`. To make it easy to interpret our results, we'll set the underlying coefficient for `INCOME` equal to the `SKU` number.

```
set.seed(10)
numObs <- 100000
testData <- data.frame(
      SKU = as.integer(runif(n = numObs, min = 1, max = 11)),
      INCOME = as.integer(runif(n = numObs, min = 10, max = 21)))
# SKU is also underlying coefficient
testData$SALES <- as.integer(testData$INCOME * testData$SKU +
      25*rnorm(n = numObs))
```

Next, we'll upload this data frame into a table in our Teradata database, removing any existing table by that name first:

```
rxSetComputeContext("local")
if (rxTeradataTableExists("sku_sales_100k",
      connectionString = tdConnString))
{
      rxTeradataDropTable("sku_sales_100k",
            connectionString = tdConnString)
}

teradataDS <- RxTeradata(table = "sku_sales_100k",
      connectionString = tdConnString)
rxDataStep(inData = testData, outFile = teradataDS)
```

7.2 A Transformation Function for By-Group Analysis

Next, let's consider the analysis we want to perform for each group. As in the previous section, we will write a transformation function that operates on a chunk of data, then writes out the results for that chunk into an "intermediate results" table. In this case we estimate a linear model on the chunk of data, and put the components of the model we will need for scoring into a string. We will also include the SKU number and the estimated slope in our results table. Note that we are assuming, at this point, that we will have all of the data for a single SKU in the chunk of data being processed by the transformation function.

```
EstimateModel <- function(dataList)
{
      # Convert the input list to a data frame
      chunkDF <- as.data.frame(dataList)
```

```
        # Estimate a linear model on this chunk
        # Don't include the model frame in the model object
        model <- lm( SALES~INCOME, chunkDF, model=FALSE )

        # Print statements can be useful for debugging
        # print( model$coefficients )

        # Remove unneeded parts of model object
        model$residuals <- NULL
        model$effects <- NULL
        model$fitted.values <- NULL
        model$assign <- NULL
        model$qr[[1]] <- NULL
        attr(model$terms, ".Environment") <- NULL

        # Convert the model to a character string
        modelString <-
          rawToChar(serialize(model, connect=NULL, ascii=TRUE))

        # Pad to a fixed length
        modelString <- paste( modelString,
          paste(rep("X", 2000 - nchar(modelString)), collapse = ""),
          sep="")

        # Create the entry for the results table
        resultDF <- data.frame(
          SKU = chunkDF[1,]$SKU, # Assumes SKU's all the same
          COEF = model$coefficients[2], # Slope
          MODEL = modelString )

        return( resultDF )
    }
```

7.3 Setting Up an Input Data Source for By-Group Analysis

The next step is to setup a data source that controls how the data is passed into the transformation function. To do so, we specify the `tableOpClause` when creating an `RxTeradata` data source. The `HASH BY`, `PARTITION BY`, `PARTITION_WITH_VIEW BY`, and `LOCAL ORDER BY` clauses can be used to organize the data. Note that this data source parameter only affects `rxDataStep` running in an `RxInTeradata` compute context. (Note: when using Teradata software with versions before 14.10.01.07-1 or 14.10.02d, replace `PARTITION` with `PARTITION-WITH-VIEW`. However, working with these versions is unsupported.)

```
tdQuery <- "SELECT * FROM sku_sales_100k"
partitionKeyword <- "PARTITION"
partitionClause <- paste( partitionKeyword, "BY sku_sales_100k.SKU" )
inDataSource <- RxTeradata(sqlQuery = tdQuery,
    tableOpClause = partitionClause,
    rowsPerRead = 100000)
```

By using the partition clause, we will have one computing resource for each SKU so that the transformation function will be operating on a single SKU of data. (For more details on using the partition clause, see Teradata's *SQL Data Manipulation Language* manual.) For modeling, we need all rows for a given model (a single SKU) to fit in one read, so `rowsPerRead` is set high.

7.4 Estimating By-Group Linear Models

Before running the analysis we need to setup the output results data source:

```
resultsDataSource = RxTeradata(table = "models",
    connectionString = tdConnString )
```

Since the `lm` function is called inside of our transformation function, we need to make sure that it is available wherever the computations are taking place. The `lm` function is contained within the `stats` package, so we specify that in the `transformPackages` RevoScaleR option:

```
rxOptions(transformPackages = c("stats"))
```

We want to make sure that our compute context is set appropriately, since the `tableOpClause` only has an effect when running in Teradata:

```
# Make sure compute context is in-database
rxSetComputeContext(tdCompute)
```

Now we can run use rxDataStep to process the data by SKU:

```
rxDataStep( inData = inDataSource, outFile = resultsDataSource,
    transformFunc = EstimateModel, reportProgress = 0,
    overwrite = TRUE )
```

We can take a quick look at our estimated slopes by bringing the data from our results table into a data frame in memory.

```
lmResults <- rxImport(resultsDataSource)
lmResults[order(lmResults$SKU),c("SKU", "COEF")]

   SKU     COEF
7    1 1.001980
8    2 2.099449
4    3 3.005241
6    4 3.979653
2    5 4.980316
1    6 5.864830
5    7 6.940838
9    8 8.151135
3    9 9.012148
10  10 9.925830
```

7.5 Scoring Using the Estimated By-Group Linear Models

We can use a similar process to score the data, using a transformation function that will only be processing data from a single SKU:

```
ScoreChunk <- function(dataList)
{
    chunkDF <- as.data.frame(dataList)

    # Extract the model string for the first observation
    # and convert back to model object
    # All observations in this chunk have the same SKU,
    # so they all share the same model
    modelString <- as.character(chunkDF[1,]$MODEL)
    model <- unserialize( charToRaw( modelString ) )

    resultDF <- data.frame(
        SKU = chunkDF$SKU,
        INCOME = chunkDF$INCOME,
        # Use lm's predict method
        PREDICTED_SALES = predict(model, chunkDF) )

    return( resultDF )
 }
```

Next, we set up an SQL query that will join our original simulated table with our intermediate results table, matching the SKU's:

```
predictQuery <- paste("SELECT sku_sales_100k.SKU,",
      "sku_sales_100k.INCOME, models.MODEL",
      "FROM sku_sales_100k JOIN models ON",
      "sku_sales_100k.SKU = models.SKU")
```

For scoring, we do not need all rows for a given model to fit in one read. As long as we compute in-database, all of the data in each chunk will belong to the same SKU.

```
inDataSource <- RxTeradata(sqlQuery = predictQuery,
      connectionString = tdConnString,
      tableOpClause = partitionClause, rowsPerRead = 10000)
```

We also need to setup our new output data source:

```
scoresDataSource = RxTeradata(table = "scores",
    connectionString = tdConnString)
```

Now we set the compute context and perform the scoring:

```
rxSetComputeContext(tdCompute)
rxDataStep( inData = inDataSource, outFile = scoresDataSource,
    transformFunc = ScoreChunk, reportProgress = 0,
    overwrite = TRUE )
```

Last, we can use an SQL query in a Teradata data source to extract summary results by SKU:

```
predSum <- rxImport( RxTeradata(
      sqlQuery = paste("SELECT SKU, INCOME, MIN(PREDICTED_SALES),",
                   "MAX(PREDICTED_SALES), COUNT(PREDICTED_SALES)",
                   "FROM scores GROUP BY SKU, INCOME",
                   "ORDER BY SKU, INCOME"),
      connectionString = tdConnString ) )
```

We'll display the first 15 rows of the results. We would expect that `PREDICTED_SALES` should be roughly `SKU*INCOME`.

```
options(width = 120) # Set display width
predSum[1:15,]
```

```
   SKU INCOME Minimum(PREDICTED_SALES) Maximum(PREDICTED_SALES) Count(PREDICTED_SALES)
1    1     10                 9.579956                 9.579956                    983
2    1     11                10.581936                10.581936                    911
3    1     12                11.583916                11.583916                    971
4    1     13                12.585896                12.585896                    895
5    1     14                13.587876                13.587876                    879
6    1     15                14.589856                14.589856                    891
7    1     16                15.591836                15.591836                    931
8    1     17                16.593815                16.593815                    944
9    1     18                17.595795                17.595795                    918
10   1     19                18.597775                18.597775                    889
11   1     20                19.599755                19.599755                    924
12   2     10                19.295443                19.295443                    939
13   2     11                21.394892                21.394892                    932
14   2     12                23.494341                23.494341                    894
15   2     13                25.593790                25.593790                    911
```

When done with the analysis, we can remove the created tables from the database and then reset the compute context to the local context.

```
rxTeradataDropTable("models")
rxTeradataDropTable("scores")
rxSetComputeContext("local")
```

8 Performing Simulations In-Database

There may be occasions when it is useful to perform computations on in-memory data in parallel on a Teradata platform. A simple example is shown here, simulating many games of craps. The game of craps is a familiar casino game that consists of rolling a pair of dice. If you roll a 7 or 11 on your initial roll, you win. If you roll 2, 3, or 12, you lose. If you roll a 4, 5, 6, 8, 9, or 10, that number becomes your *point* and you continue rolling until you either roll your point again (in which case you win) or roll a 7, in which case you lose. The game is easily simulated in R using the following function:

```
playCraps <- function()
{
      result <- NULL
      point <- NULL
      count <- 1
      while (is.null(result))
      {
            roll <- sum(sample(6, 2, replace=TRUE))

            if (is.null(point))
            {
                  point <- roll
            }
            if (count == 1 && (roll == 7 || roll == 11))
            {
                  result <- "Win"
            }
            else if (count == 1 &&
               (roll == 2 || roll == 3 || roll == 12))
            {
                  result <- "Loss"
            }
            else if (count > 1 && roll == 7 )
            {
                  result <- "Loss"
            }
            else if (count > 1 && point == roll)
            {
                  result <- "Win"
            }
            else
            {
                  count <- count + 1
            }
      }
      result
}
```

We can run the function to simulate a single game:

```
playCraps()
```

Did you win or lose?

We will now simulate 1000 games of craps to help determine the probability of a win. The *rxExec* function allows computations to be distributed to each of the AMPS of your Teradata platform. If the *timesToRun* argument is specified, runs may be performed multiple times on each AMP. The *rxExec* function returns a list with one element for every run. Setting *RNGseed* to *"auto"* causes separate parallel random number substreams (using the "MT2203" generator) to be generated and used for each of the 1000 invocations of playCraps.

```
teradataExec <- rxExec(playCraps, timesToRun=1000, RNGseed="auto")
length(teradataExec)
```

To see a summary of the results, we can convert the returned list to a vector, and summarize the results using the table function.

```
table(unlist(teradataExec))
```

Your results should look something like this:

```
Loss  Win
 508  492
```

9 Analyzing Your Data Alongside Teradata

Although typically it is much faster to perform in-database analyses, it is sometimes convenient to extract your data from Teradata and analyze it alongside with an alternative, powerful compute engine.   To perform the computations on your local machine, we just need to change the compute context back to local.

```
# Set the compute context to compute locally
rxSetComputeContext ("local")
```

9.1 Perform an rxSummary Using a Local Compute Context

When extracting data from Teradata, it is often more performant to increase the number of rows extracted for each read. We can do this by increasing the `rowsPerRead` in the data source:

```
tdQuery <- "SELECT * FROM ccFraudScore10"
teradataDS1 <- RxTeradata(connectionString = tdConnString,
    sqlQuery = tdQuery, colInfo = ccColInfo, rowsPerRead = 500000)
```

Now we can call rxSummary using the new data source. Note that this will be slow if you have a slow connection to your database; the data is being transferred to you local computer for analysis.

```
rxSummary(formula = ~gender + balance + numTrans + numIntlTrans +
      creditLine, data = teradataDS1)
```

You should see the same results as you did when you performed the computations in-database.

9.2 Import of Data from a Teradata Database to a Local File

The `rxImport` function allows you to import data from a data source to an local “xdf” file, . This can be convenient if you want to repeatedly analyze a subset of your data initially stored in a Teradata Database. Let’s store the variables `gender`, `cardholder`, `state`, and `balance` for the states of California, Oregon, and Washington on our local computer. We’ll create a new Teradata data source object to use as the `inData` argument for `rxImport`. First, let’s specify the variables and selection we want to read in the SQL query. (Make sure there are no hidden characters such as line feeds or tabs in the query.) We can use the `stateAbb` vector created earlier in this guide to identify the correct levels to include.

```
statesToKeep <- sapply(c("CA", "OR", "WA"), grep, stateAbb)
statesToKeep

importQuery <- paste("SELECT gender,cardholder,balance,state",
      "FROM ccFraud10",
      "WHERE (state = 5 OR state = 38 OR state = 48)")
```

Next we'll create the `colInfo` to be used. In our new data set, we just want 3 factor levels for the 3 states being included. We can use `statesToKeep` to identify the correct levels to include.

```
importColInfo <- list(
    gender = list(
        type = "factor",
        levels = c("1", "2"),
        newLevels = c("Male", "Female")),
    cardholder = list(
          type = "factor",
          levels = c("1", "2"),
          newLevels = c("Principal", "Secondary")),
      state = list(
          type = "factor",
          levels = as.character(statesToKeep),
            newLevels = names(statesToKeep))
      )
```

Since we are importing to our local computer, we'll be sure the compute context is set to local. We will store the data in a file named `ccFraudSub.xdf` in our current working directory

```
rxSetComputeContext("local")
teradataImportDS <- RxTeradata(connectionString = tdConnString,
      sqlQuery = importQuery, colInfo = importColInfo)

localDS <- rxImport(inData = teradataImportDS,
      outFile = "ccFraudSub.xdf",
      overwrite = TRUE)
```

The *localDs* object returned from *rxImport* is a "light-weight" *RxXdfData* data source object representing the *ccFraud.xdf data* file stored locally on disk. It can be used in analysis functions in the same way as the Teradata data source.

For example,

```
rxGetVarInfo(data = localDS)

Var 1: gender
       2 factor levels: Male Female
Var 2: cardholder
       2 factor levels: Principal Secondary
Var 3: balance, Type: integer, Low/High: (0, 39987)
Var 4: state
       3 factor levels: CA OR WA

rxSummary(~gender + cardholder + balance + state, data = localDS)

Call:
rxSummary(formula = ~gender + cardholder + balance + state, data = localDS)

Summary Statistics Results for: ~gender + cardholder + balance + state
Data: localDS (RxXdfData Data Source)
```

```
File name: ccFraud.xdf
Number of valid observations: 1540887

 Name    Mean     StdDev   Min Max   ValidObs MissingObs
 balance 4115.967 4000.873 0   39987 1540887  0

Category Counts for gender
Number of categories: 2
Number of valid observations: 1540887
Number of missing observations: 0

 gender Counts
 Male   952908
 Female 587979

Category Counts for cardholder
Number of categories: 2
Number of valid observations: 1540887
Number of missing observations: 0

 cardholder Counts
 Principal  1494355
 Secondary    46532

Category Counts for state
Number of categories: 3
Number of valid observations: 1540887
Number of missing observations: 0

 state Counts
 CA    1216069
 OR     121846
 WA     202972
```

10 Managing Memory in In-Teradata Computations

Because a Teradata cluster node typically has many worker processes (AMPs) running constantly, it is important to limit how much memory a distributed analysis consumes. Microsoft provides stored procedures in the revoAnalytics_Zqht2 scheduler database that allow a database administrator to set the memory limits for master and worker processes working on a RevoScaleR job. By default, the master process memory limit is 2000 MB (approximately 2GB) and the worker process memory limit is 1000MB (approximately 1GB).

These limits may be customized by a database administrator using the SetMasterMemoryLimitMB() and SetWorkerMemoryLimitMB() stored procedures, defined in the revoAnalytics_Zqht2 database created on installation.

For example, to set the master memory limit to 3000MB:

```
call revoAnalytics_Zqht2.SetMasterMemoryLimitMB(3000);
```

To set the worker memory limit to 1500MB:

```
call revoAnalytics_Zqht2.SetWorkerMemoryLimitMB(1500);
```

The current memory limits can be viewed in the revoAnalytics_Zqht2.Properties table.

Note that memory limits that have been changed are in effect immediately, and no restart of the database is required.

11 Appendix I: Scripts for Loading Data into Teradata

Modify the following script to include your Log In information and data base name, then use the ‘fastload’ Teradata command to load the ccFraud.csv data. (Copies of both scripts in this appendix can be found online.)

```
sessions 2;
errlimit 25;
logon PUT YOUR LOGON INFO HERE: MachineNameOrIP/UserName,Password;

DROP TABLE RevoTestDB.ccFraud10;
DROP TABLE RevoTestDB.ccFraud10_error_1;
DROP TABLE RevoTestDB.ccFraud10_error_2;

CREATE TABLE RevoTestDB.ccFraud10 (
"custID" INT,
"gender" INT,
"state" INT,
"cardholder" INT,
"balance" INT,
"numTrans" INT,
"numIntlTrans" INT,
"creditLine" INT,
"fraudRisk" INT)
PRIMARY INDEX ("custID");

set record VARTEXT ",";

RECORD 2;

DEFINE
Col1 (VARCHAR(15)),
Col2 (VARCHAR(15)),
Col3 (VARCHAR(15)),
Col4 (VARCHAR(15)),
Col5 (VARCHAR(15)),
Col6 (VARCHAR(15)),
Col7 (VARCHAR(15)),
Col8 (VARCHAR(15)),
Col9 (VARCHAR(15))
FILE=ccFraud.csv;

SHOW;

begin loading RevoTestDB.ccFraud10 errorfiles
RevoTestDB.ccFraud10_error_1, RevoTestDB.ccFraud10_error_2;
insert into RevoTestDB.ccFraud10 (
:Col1,
:Col2,
:Col3,
:Col4,
:Col5,
:Col6,
:Col7,
:Col8,
:Col9);

END LOADING;

LOGOFF;
```

This similar script will load the ccFraudScore.csv data:

```
sessions 2;
errlimit 25;
logon PUT YOUR LOGON INFO HERE;

DROP TABLE RevoTestDB.ccFraudScore10;
DROP TABLE RevoTestDB.ccFraudScore_error_1;
DROP TABLE RevoTestDB.ccFraudScore_error_2;

CREATE TABLE RevoTestDB.ccFraudScore10 (
"custID" INT,
"gender" INT,
"state" INT,
"cardholder" INT,
"balance" INT,
"numTrans" INT,
"numIntlTrans" INT,
"creditLine" INT)
PRIMARY INDEX ("custID");

set record VARTEXT ",";

RECORD 2;

DEFINE
Col1 (VARCHAR(15)),
Col2 (VARCHAR(15)),
Col3 (VARCHAR(15)),
Col4 (VARCHAR(15)),
Col5 (VARCHAR(15)),
Col6 (VARCHAR(15)),
Col7 (VARCHAR(15)),
Col8 (VARCHAR(15))
FILE=ccFraudScore.csv;

SHOW;

begin loading RevoTestDB.ccFraudScore errorfiles
RevoTestDB.ccFraudScore_error_1, RevoTestDB.ccFraudScore_error_2;
insert into RevoTestDB.ccFraudScore (
:Col1,
:Col2,
:Col3,
:Col4,
:Col5,
:Col6,
:Col7,
:Col8);
```

```
END LOADING;

LOGOFF;
```

To run a *fastload* command, navigate to the directory containing the script from a command prompt and enter:

```
fastload < TheNameOFtheFastLoadScript.fastload
```