Microsoft

RevoScaleR Hadoop Spark Getting Started Guide

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

RevoScaleR Hadoop Spark Getting Started Guide

Microsoft
One Microsoft Way
Redmond, WA 98052
USA.

Revised on March 21, 2016

Contents

1 Overview

This guide is an introduction to using the ***RevoScaleR*** package in a distributed computing environment of using Spark on Hadoop. ***RevoScaleR*** provides functions for performing scalable and extremely high performance data management, analysis, and visualization. Hadoop provides a distributed file system and a framework for distributed computation. This guide focuses on using ***RevoScaleR***'s HPA 'big data' capabilities in the Hadoop environment with Spark.

While ***RevoScaleR*** makes use of the Hadoop Spark framework, you need have no experience or detailed knowledge of that framework to use ***RevoScaleR*** in Hadoop. All you need to know is basic information about connection to your Hadoop cluster. This guide will walk you through the rest.

The data manipulation and analysis functions in ***RevoScaleR*** are appropriate for small and large datasets, but are particularly useful in three common situations: 1) to analyze data sets that are too big to fit in memory and, 2) to perform computations distributed over several cores, processors, or nodes in a cluster, or 3) to create scalable data analysis routines that can be developed locally with smaller data sets, then deployed to larger data and/or a cluster of computers. These are ideal candidates for ***RevoScaleR*** because ***RevoScaleR*** is based on the concept of operating on chunks of data and using *updating algorithms.*

The RevoScaleR high performance analysis functions are portable. The same high performance functions work on a variety of computing platforms, including Windows and RHEL workstations and servers and distributed computing platforms including Hadoop. So, for example, you can do exploratory analysis on your laptop, then deploy the same analytics code on a Hadoop cluster. The underlying RevoScaleR code handles the distribution of the computations across cores and nodes, so you don't have to worry about it. For those interested in the underlying architecture, on Hadoop the RevoScaleR analysis functions go through the following steps.

- A master process is initiated to run the main thread of the algorithm.
- The master process initiates a Spark job to make a pass through the data.
- The mapper produces "intermediate results objects" for each task processing a chunk of data. These are combined using a combiner and then a reducer
- The master process examines the results. For iterative algorithms, it decides if another pass through the data is required. If so, it initiates another Spark job and repeats.
- When complete, the final results are computed and returned.

When running on Hadoop, the RevoScaleR analysis functions process data contained in the Hadoop Distributed File System (HDFS). HDFS data can also be accessed directly from RevoScaleR, without performing the computations within the Hadoop framework. An example of this is shown in Section 5.6 on *Using a Local Compute Context with HDFS Data*.

More detailed examples of using ***RevoScaleR*** can be found in the following provided with ***RevoScaleR***:

- *RevoScaleR Getting Started Guide* (RevoScaleR_Getting_Started.pdf)
- *RevoScaleR User's Guide* (RevoScaleR_Users_Guide.pdf)
- *RevoScaleR Hadoop MapReduce Getting Started Guide* (RevoScaleR_MapReduce_Getting_Started.pdf)
- *RevoScaleR Distributed Computing Guide* [RevoScaleR_Distributed_Computing.pdf; see this guide for HPC examples]
- *RevoScaleR ODBC Data Import Guide* (RevoScaleR_ODBC.pdf)

If you would like information on using other ***RevoScaleR*** distributed computing contexts, see:

- *RevoScaleR LSF Cluster Getting Started Guide* (RevoScaleR_LSF_Cluster_GettingStarted.pdf)
- *RevoScaleR HPC Server Getting Started Guide* (RevoScaleR_HPC_Server_Getting_Started.pdf)

2 Data Sources and Functions Supported in Hadoop

The ***RevoScaleR*** package provides a set of portable, scalable, distributable data analysis functions. To perform an analysis, the user specifies three distinct pieces of information: where the computations should take place (the compute context), the data to use (the data source), and what analysis to perform (the analysis function). The ***RevoScaleR*** package also provides a set of data manipulation functions that are typically available in a local compute context.

Of course, not all data source types are available on all compute contexts. For the Spark compute context used in this Guide, named *RxSpark*, two types of data sources can be used: a comma delimited text data source (see *RxTextData*) and an efficient XDF data file format (see *RxXdfData*). As noted later in this Guide, the XDF file format has been modified for Hadoop to store data in a composite set of files rather than a single file. Both of these data sources can be specified for use with the Hadoop Distributed File System (HDFS).

The RevoScaleR analysis functions currently supported on with the Spark compute context are:

- *rxSummary*: Basic summary statistics of data, including computations by group. (Writing by group computations to .xdf file not supported.)
- *rxQuantile*: Compute approximate quantiles
- *rxCrossTabs*: Formula-based cross-tabulation of data.
- *rxCube*: Alternative formula-based cross-tabulation returning 'cube' results. (Writing output to .xdf file not supported.)
- *rxLinMod*: Fits a linear model to data

- *rxCovCor*: Calculate the covariance, correlation, or sum of squares / cross-product matrix for a set of variables.
- *rxLogit*: Fits a logistic regression model to data.
- *rxGlm*: Fits a generalized linear model to data
- *rxKmeans*: Performs k-means clustering.
- *rxDTree*: Fits a classification or regression tree to data.
- *rxDForest*: Fits a classification or regression decision forest to data.
- *rxBTrees*: Fits a classification or regression decision forest to data using a stochastic gradient boosting algorithm.
- *rxPredict*: Calculates predictions for fitted models. Output must be an XDF data source.

High performance computing is supported in Hadoop using:

- *rxExec*: Run an arbitrary R function on nodes of a cluster

The Spark compute context also allows the following data manipulation functionality:

- *rxDataStep*: Transform and subset data. Output can be an XDF data source, a comma delimited text data source (EXPERIMENTAL), or a data frame in memory (assuming you have sufficient memory to hold the output data).
- *rxFactors*: Create or recode factor variables in a composite XDF file in HDFS. A new file must be written out.

The following ‘helper’ functions to get basic information about your data source:

- *rxGetInfo*
- *rxGetVarInfo*
- *rxGetVarNames*

The Spark compute context has a number of job-related functions particularly helpful when running non-waiting jobs:

- *rxGetJobStatus*: Get the status of a non-waiting distributed computing job.
- *rxGetJobResults*: Get the return object(s) of a non-waiting distributed computing job.
- *rxGetJobOutput*: Get the console output from a non-waiting distributed computing job.
- *rxGetJobs*: Get the available distributed computing job information objects.

RevoScaleR also provides some wrapper functions for accessing Hadoop/HDFS functionality via R:

- *rxHadoopCommand*: Allows you to run basic Hadoop commands.
- *rxHadoopVersion*: Returns just the version string from running the *hadoop version* command.

- *rxCopyFromClient*: Copy a file from a remote client to the Hadoop Distributed File System on the Hadoop cluster.
- *rxHadoopCopyFromLocal*: Wraps the Hadoop fs -copyFromLocal command.
- *rxHadoopCopyToLocal*: Wraps the Hadoop fs -copyToLocal command.
- *rxHadoopListFiles*: Wraps the Hadoop fs -ls or fs -lsr command.
- *rxHadoopRemove*: Wraps the Hadoop fs -rm command.
- *rxHadoopCopy*: Wraps the Hadoop fs -cp command.
- *rxHadoopMove*: Wraps the Hadoop fs -mv command.
- *rxHadoopMakeDir*: Wraps the Hadoop fs -mkdir command.
- *rxHadoopRemoveDir*: Wraps the Hadoop fs -rmr command.

3 Installation

Before you can use ***Microsoft R Services*** (which contains the ***RevoScaleR*** package) with Hadoop, you must have a Hadoop cluster configured. ***Microsoft R Services*** is currently supported on the following Hadoop distributions with the RHEL5 or RHEL6 operating systems:

- Cloudera CDH 5.0, 5.1, 5.2, 5.3, 5.4
- HortonWorks HDP 1.3.0, HDP 2.0.0, HDP 2.1.0, HDP 2.2.0, HDP 2.3.0
- MapR 3.0.2, MapR 3.0.3, MapR 3.1.0, MapR 3.1.1, MapR 4.0.1, MapR 4.0.2 (provided this version of MapR has been updated to mapr-patch-4.0.2.29870.GA30600; contact MapR to obtain the patch), MapR 4.1

If you have such a Hadoop cluster and would like to install ***Microsoft R Services***, follow the instructions in the *Microsoft R Services Hadoop Configuration Guide*, which is part of the Linux installer distribution.

4 Running the Examples in the Getting Started Guide

This guide makes extensive use of the *Airline 2012 On-Time Data Set,* a set of 12 comma-separated files containing information on flight arrival and departure details for all commercial flights within the USA, for the year 2012. This is a big data set with over six million observations, and is used in the examples in Section 6.

Section 5 uses the AirlineDemoSmall.csv file from the RevoScaleR SampleData directory. That tutorial section describes how to copy the file from the sample data directory into HDFS.

You can obtain these data sets online.

Note: RevoScaleR (and this manual) assumes the existence of directories "/var/RevoShare" and "/var/RevoShare/$USER" in the native file system and "/user/RevoShare" and "/user/RevoShare/$USER" in the Hadoop Distributed File System on the Hadoop cluster. Other writable directories may be substituted, but some examples may need to be modified. This manual also assumes the existence of a writable directory "/share" on the Hadoop Distributed File System. If this directory does not exist, or is not writable by ***Microsoft R Services*** users, another writable directory must be substituted in the examples.

5 A Tutorial Introduction to RevoScaleR in Hadoop Spark

This section contains a detailed introduction to the most important high performance analytics features of ***RevoScaleR*** using data stored on your Hadoop cluster. The following tasks are performed***:***

1. Starting Microsoft R Services
2. Specify the NameNode.
3. Create a compute context for Spark.
4. Copy a data set into the Hadoop Distributed File System.
5. Create a data source.
6. Summarize your data.
7. Fit a linear model to the data.

5.1 Starting Microsoft R Services

How you start ***Microsoft R Services*** depends on which operating system you are running. On Linux hosts (including nodes of your Hadoop cluster), you start ***Microsoft R Services*** by typing the following at the shell prompt:

```
Revo64
```

On Windows 7 and earlier, you start ***Microsoft R Services*** as follows:

1. From the Task Bar, click **Start**.
2. Click **All Programs**.
3. Click **Revolution R**.
4. Click **Enterprise 8.0**.
5. Click **Revolution R Enterprise 8.0 (64)**.

On Windows 8 and Windows Server 2012, you start ***Microsoft R Services*** as follows:

1. Move your mouse to the lower left corner of the Desktop until **Start** pops up.
2. Click **Start** to view the Start screen.
3. Locate the tile for **Revolution R Enterprise 8.0 (64)**.

On Windows 10, you start ***Microsoft R Services*** as follows:

1. From the Task Bar, click **Start**.
2. Click **All apps**.
3. Click **Revolution R**.
4. Click **Revolution R Enterprise 8.0 (64)**.

5.2 Creating a Compute Context for Hadoop Spark

A *compute context* specifies the computing resources to be used by ***RevoScaleR***'s distributable computing functions. In this manual, we focus on using the nodes of the Hadoop cluster (internally via Spark) as the computing resources. In defining your compute context, you may have to specify different parameters depending on whether you are running from a node of your cluster or from a client accessing the cluster remotely.

5.2.1 Defining a Compute Context on the Cluster

If you are running on one of the nodes of the Hadoop cluster (which may be an edge node), you can define a Hadoop Spark compute context that uses the default values:

```
myHadoopCluster <- RxSpark()
```

Note: The default settings include a specification of `/var/RevoShare/$USER` as the `shareDir` and `/user/RevoShare/$USER` as the `hdfsShareDir`—that is, the default locations for writing various files on the cluster's local file system and HDFS file system, respectively. These directories must both exist and be writable for your cluster jobs to succeed. You must either create these directories or specify suitable writable directories for these parameters. If you are working on a node of the cluster, the default specifications for the shared directories are:

```
myShareDir = paste( "/var/RevoShare", Sys.info()[["user"]],
      sep="/" )
myHdfsShareDir = paste( "/user/RevoShare", Sys.info()[["user"]],
      sep="/" )
```

You can have many compute context objects available for use; only one is active at any one time. You specify the active compute context using the `rxSetComputeContext` function:

```
rxSetComputeContext(myHadoopCluster)
```

5.2.2 Defining a Compute Context on a High-Availability Cluster

If you are running on a Hadoop cluster configured for high-availabilty, you must specify the node providing the name service using the `nameNode` argument to `RxSpark`, and also specify the Hadoop port with the `port` argument:

```
myHadoopCluster <- RxSpark
(nameNode = "my-name-service-server",
    port = 8020)
```

5.2.3 Using Microsoft R Services as a Hadoop Client

If you are running Microsoft R Services from Linux or from a Windows computer equipped with PuTTY *and/or* both the Cygwin shell and Cygwin OpenSSH packages, you can create a compute context that will run ***RevoScaleR*** functions from your local client in a distributed fashion on your Hadoop cluster. You use *RxSpark* to create the compute context, but use additional arguments to specify your user name, the file-sharing directory where you have read and write access, the publicly-facing host name or IP address of your Hadoop cluster's name node or an edge node that will run the master processes, and any additional switches to pass to the ssh command (such as the -i flag if you are using a pem or ppk file for authentication, or -p to specify a non-standard ssh port number). For example:

```
mySshUsername <- "user1"
#public facing cluster IP address
mySshHostname <- "12.345.678.90"
mySshSwitches <- "-i /home/yourName/user1.pem" #See NOTE below
myShareDir <- paste("/var/RevoShare", mySshUsername, sep ="/")
myHdfsShareDir <- paste("/user/RevoShare",mySshUsername, sep="/")

myHadoopCluster <- RxSpark(
    hdfsShareDir = myHdfsShareDir,
    shareDir     = myShareDir,
    sshUsername  = mySshUsername,
    sshHostname  = mySshHostname,
    sshSwitches  = mySshSwitches)
```

NOTE if you are using a pem or ppk file for authentication the permissions of the file must be modified to ensure that only the owner has full read and write access (i.e. chmod go-rwx user1.pem).

If you are using PuTTY, you may incorporate the publicly facing host name and any authentication requirements into a PuTTY saved session, and use the name of that saved session as the sshHostname. For example:

```
mySshUsername <- "user1"
#name of PuTTY saved session
mySshHostname <- "myCluster"
myShareDir <- paste("/var/RevoShare", mySshUsername, sep ="/")
myHdfsShareDir <- paste("/user/RevoShare",mySshUsername, sep="/")

myHadoopCluster <- RxSpark(
    hdfsShareDir = myHdfsShareDir,
    shareDir     = myShareDir,
    sshUsername = mySshUsername,
    sshHostname = mySshHostname)
```

The above assumes that the directory containing the ssh and scp commands (Linux/Cygwin) or plink and pscp commands (PuTTY) is in your path (or that the Cygwin installer has stored its directory in the Windows registry). If not, you can specify the location of these files using the sshClientDir argument:

```
myHadoopCluster <- RxSpark(
    hdfsShareDir = myHdfsShareDir,
    shareDir     = myShareDir,
    sshUsername = mySshUsername,
    sshHostname = mySshHostname,
    sshClientDir = "C:\\Program Files (x86)\\PuTTY")
```

In some cases, you may find that environment variables needed by Hadoop are not set in the remote sessions run on the sshHostname computer. This may be because a different profile or startup script is being read on ssh login. You can specify the appropriate profile script by specifying the sshProfileScript argument to *RxSpark*, e.g., “/etc/profile” or “/home/<user>/.bash_profile”. This should be an absolute path:

```
myHadoopCluster <- RxSpark(
    hdfsShareDir = myHdfsShareDir,
    shareDir     = myShareDir,
    sshUsername  = mySshUsername,
    sshHostname  = mySshHostname,
    sshProfileScript = "/etc/profile",
    sshSwitches  = mySshSwitches)
```

Now that you have defined your compute context, make it the active compute context using the *rxSetComputeContext* function:

```
rxSetComputeContext(myHadoopCluster)
```

5.2.4 Using RxSpark Compute Context in Persistent Mode

By default, with *RxSpark* Compute Context, a new Spark application will be launched when a job starts and will be terminated when the job completes. To avoid the overhead of spark initialization on launching time, the *persistentRun* mode (experimental) is introduced. If set *persistentRun* as “TRUE”, the Spark application (and associated processes) will persist across jobs until *idleTimeout* or the *rxStopEngine* is called explicitly:

```
myHadoopCluster <- RxSpark(persistentRun = TRUE, idleTimeout =
600)
```

The *idleTimeout* is the number of seconds of being idle before system kills the Spark process.

If *persistentRun* mode is enabled, then the *RxSpark* compute context cannot be a Non-Wait Compute Context. See Non-Wait Compute Context in section 5.8.

5.3 Copying a Data File into the Hadoop Distributed File System

For our first explorations, we will work with one of RevoScaleR’s built-in data sets, *AirlineDemoSmall.csv*. This is part of the standard Microsoft R Services distribution. You can verify that it is on your local system as follows:

```
file.exists(system.file("SampleData/AirlineDemoSmall.csv",
    package="RevoScaleR"))
[1] TRUE
```

To use this file in our distributed computations, it must first be copied to HDFS. For our examples, we will make extensive use of the HDFS shared director, /share:

```
bigDataDirRoot <- "/share"  # HDFS location of the example data
```

First, check to see what directories and files are already in your shared file directory. You can use the `rxHadoopListFiles` function, which will automatically check your active compute context for information:

```
rxHadoopListFiles(bigDataDirRoot)
```

If the AirlineDemoSmall subdirectory does not exist and you have write permission, you can use the following functions to copy the data there:

```
source <-system.file("SampleData/AirlineDemoSmall.csv",
    package="RevoScaleR")
inputDir <- file.path(bigDataDirRoot,"AirlineDemoSmall")
rxHadoopMakeDir(inputDir)
rxHadoopCopyFromLocal(source, inputDir)
```

We can then verify that it exists as follows:

```
rxHadoopListFiles(inputDir)
```

5.4 Creating a Data Source

We will create a data source using this file, specifying that it is on the Hadoop Distributed File System. We first create a file system object that uses the default values:

```
hdfsFS <- RxHdfsFileSystem()
```

The input .csv file uses the letter `M` to represent missing values, rather than the default `NA`, so we specify this with the `missingValueString` argument. We will explicitly set the factor levels for `DayOfWeek` in the desired order using the `colInfo` argument:

```
colInfo <- list(DayOfWeek = list(type = "factor",
    levels = c("Monday", "Tuesday", "Wednesday", "Thursday",
    "Friday", "Saturday", "Sunday")))

airDS <- RxTextData(file = inputDir, missingValueString = "M",
    colInfo  = colInfo, fileSystem = hdfsFS)
```

5.5 Summarizing Your Data

Use the `rxSummary` function to obtain descriptive statistics for your data. The `rxSummary` function takes a formula as its first argument, and the name of the data set as the second.

```
adsSummary <- rxSummary(~ArrDelay+CRSDepTime+DayOfWeek,
    data = airDS)
adsSummary
```

Summary statistics will be computed on the variables in the formula, removing missing values for all rows in the included variables:

```
Call:
rxSummary(formula = ~ArrDelay + CRSDepTime + DayOfWeek, data = airDS)
Summary Statistics Results for: ~ArrDelay + CRSDepTime + DayOfWeek
Data: airDS (RxTextData Data Source)
File name: /share/AirlineDemoSmall
Number of valid observations: 6e+05

 Name       Mean     StdDev    Min        Max        ValidObs MissingObs
 ArrDelay   11.31794 40.688536 -86.000000 1490.00000 582628   17372
 CRSDepTime 13.48227  4.697566   0.016667   23.98333 600000       0

Category Counts for DayOfWeek
Number of categories: 7
Number of valid observations: 6e+05
Number of missing observations: 0

 DayOfWeek Counts
 Monday    97975
 Tuesday   77725
 Wednesday 78875
 Thursday  81304
 Friday    82987
 Saturday  86159
 Sunday    94975
```

Notice that the summary information shows cell counts for categorical variables, and appropriately does not provide summary statistics such as *Mean* and *StdDev*. Also notice that the *Call:* line will show the actual call you entered or the call provided by *summary*, so will appear differently in different circumstances.

You can also compute summary information by one or more categories by using interactions of a numeric variable with a factor variable. For example, to compute summary statistics on Arrival Delay by Day of Week:

```
rxSummary(~ArrDelay:DayOfWeek, data = airDS)
```

The output shows the summary statistics for *ArrDelay* for each day of the week:

```
Call:
rxSummary(formula = ~ArrDelay:DayOfWeek, data = airDS)

Summary Statistics Results for: ~ArrDelay:DayOfWeek
Data: airDS (RxTextData Data Source)
File name: /share/AirlineDemoSmall
Number of valid observations: 6e+05

 Name               Mean     StdDev   Min Max  ValidObs MissingObs
 ArrDelay:DayOfWeek 11.31794 40.68854 -86 1490 582628   17372

Statistics by category (7 categories):
 Category                          DayOfWeek Means     StdDev   Min Max ValidObs
 ArrDelay for DayOfWeek=Monday     Monday    12.025604 40.02463 -76 1017 95298
 ArrDelay for DayOfWeek=Tuesday    Tuesday   11.293808 43.66269 -70 1143 74011
 ArrDelay for DayOfWeek=Wednesday Wednesday 10.156539 39.58803 -81 1166 76786
 ArrDelay for DayOfWeek=Thursday   Thursday   8.658007 36.74724 -58 1053 79145
 ArrDelay for DayOfWeek=Friday     Friday    14.804335 41.79260 -78 1490 80142
 ArrDelay for DayOfWeek=Saturday   Saturday  11.875326 45.24540 -73 1370 83851
 ArrDelay for DayOfWeek=Sunday     Sunday    10.331806 37.33348 -86 1202 93395
```

5.6 Using a Local Compute Context with HDFS Data

At times it may be more efficient to perform smaller computations on the local node rather than using Spark. You can easily do this, accessing the same data from the HDFS file system. When working with the local compute context, you will need to specify the name of a specific data file. The same basic code is then used to run the analysis; simply change the compute context to a local context. (Note that this will not work if you are accessing the Hadoop cluster via a client.)

```
rxSetComputeContext("local")
inputFile <-
file.path(bigDataDirRoot,"AirlineDemoSmall/AirlineDemoSmall.csv")
airDSLocal <- RxTextData(file = inputFile,
    missingValueString = "M",
    colInfo  = colInfo, fileSystem = hdfsFS)
adsSummary <- rxSummary(~ArrDelay+CRSDepTime+DayOfWeek,
      data = airDSLocal)
adsSummary
```

The results are the same as doing the computations across the nodes with the Hadoop Spark compute context:

```
Call:
rxSummary(formula = ~ArrDelay + CRSDepTime + DayOfWeek, data = airDS)

Summary Statistics Results for: ~ArrDelay + CRSDepTime + DayOfWeek
Data: airDS (RxTextData Data Source)
File name: /share/AirlineDemoSmall/AirlineDemoSmall.csv
Number of valid observations: 6e+05

 Name       Mean     StdDev    Min        Max        ValidObs MissingObs
 ArrDelay   11.31794 40.688536 -86.000000 1490.00000 582628   17372
 CRSDepTime 13.48227  4.697566   0.016667   23.98333 600000       0

Category Counts for DayOfWeek
Number of categories: 7
Number of valid observations: 6e+05
Number of missing observations: 0

 DayOfWeek Counts
 Monday    97975
 Tuesday   77725
 Wednesday 78875
 Thursday  81304
 Friday    82987
 Saturday  86159
 Sunday    94975
```

Now set the compute context back to the Hadoop cluster for further analyses:

```
rxSetComputeContext(myHadoopCluster)
```

5.7 Fitting a Linear Model

Use the `rxLinMod` function to fit a linear model using your *airDS* data source. Use a single dependent variable, the factor `DayOfWeek`:

```
arrDelayLm1 <- rxLinMod(ArrDelay ~ DayOfWeek, data = airDS)
summary(arrDelayLm1)
```

The resulting output is:

```
Call:
rxLinMod(formula = ArrDelay ~ DayOfWeek, data = airDS)

Linear Regression Results for: ArrDelay ~ DayOfWeek
Data: airDS (RxTextData Data Source)
File name: /share/AirlineDemoSmall
Dependent variable(s): ArrDelay
Total independent variables: 8 (Including number dropped: 1)
Number of valid observations: 582628
Number of missing observations: 17372

Coefficients: (1 not defined because of singularities)
                    Estimate Std. Error t value Pr(>|t|)
(Intercept)          10.3318     0.1330  77.673 2.22e-16 ***
DayOfWeek=Monday      1.6938     0.1872   9.049 2.22e-16 ***
DayOfWeek=Tuesday     0.9620     0.2001   4.809 1.52e-06 ***
DayOfWeek=Wednesday  -0.1753     0.1980  -0.885    0.376
DayOfWeek=Thursday   -1.6738     0.1964  -8.522 2.22e-16 ***
DayOfWeek=Friday      4.4725     0.1957  22.850 2.22e-16 ***
DayOfWeek=Saturday    1.5435     0.1934   7.981 2.22e-16 ***
DayOfWeek=Sunday     Dropped    Dropped Dropped  Dropped
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 40.65 on 582621 degrees of freedom
Multiple R-squared: 0.001869
Adjusted R-squared: 0.001858
F-statistic: 181.8 on 6 and 582621 DF,  p-value: < 2.2e-16
Condition number: 10.5595
```

5.8 Creating a Non-Waiting Compute Context

When running all but the shortest analyses in Hadoop, it can be convenient to let Hadoop do its processing while returning control of your R session to you immediately.

You can do this by specifying `wait = FALSE` in your compute context definition. By using our existing compute context as the first argument, all of the other settings will be carried over to the new compute context:

```
myHadoopNoWaitCluster <- RxSpark(myHadoopCluster, wait = FALSE)
rxSetComputeContext(myHadoopNoWaitCluster)
```

Once you have set your compute context to non-waiting, distributed ***RevoScaleR*** functions return relatively quickly with a `jobInfo` object, which you can use to track the progress of your job, and, when the job is complete, obtain the results of the job. For example, we can re-run our linear model in the non-waiting case as follows:

```
job1 <- rxLinMod(ArrDelay ~ DayOfWeek, data = airDS)
rxGetJobStatus(job1)
```

Right after submission, the job status will typically return `"running"`. When the job status returns `"finished"`, you can obtain the results using `rxGetJobResults` as follows:

```
arrDelayLm1 <- rxGetJobResults(job1)
summary(arrDelayLm1)
```

You should always assign the `jobInfo` object so that you can easily track your work, but if you forget, the most recent `jobInfo` object is saved in the global environment as the object `rxgLastPendingJob`. (By default, after you've retrieved your job results, the results are removed from the cluster. To have your job results remain, set the `autoCleanup` argument to `FALSE` in `RxSpark`.)

If, after submitting a job in a non-waiting compute context, you decide you don't want to complete the job, you can cancel the job using the `rxCancelJob` function:

```
job2 <- rxLinMod(ArrDelay ~ DayOfWeek, data = airDS)
rxCancelJob(job2)
```

The `rxCancelJob` function returns `TRUE` if the job is successfully cancelled, `FALSE` otherwise.

For the remainder of this tutorial, we return to a waiting compute context:

```
rxSetComputeContext(myHadoopCluster)
```

6 Analyzing a Large Data Set with RevoScaleR

6.1 Getting Set Up to Use Your CSV Files

We will now move to examples using a more recent version of the airline data set. The airline on-time data has been gathered by the Department of Transportation since 1987.

The data through 2008 was used in the American Statistical Association Data Expo in 2009 (http://stat-computing.org/dataexpo/2009/). ASA describes the data set as follows:

The data consists of flight arrival and departure details for all commercial flights within the USA, from October 1987 to April 2008. This is a large dataset: there are nearly 120 million records in total, and takes up 1.6 gigabytes of space compressed and 12 gigabytes when uncompressed.

The airline on-time data set for 2012, consisting of 12 separate CSV files, is available online. We assume you have uploaded them to the /tmp directory on your name node (although any directory visible as a native file system directory from the name node will work.)

As before, our first step is to copy the data into HDFS. We specify the location of your Hadoop-stored data for the `airDataDir` variable:

```
airDataDir <- file.path(bigDataDirRoot,"/airOnTime12/CSV")
rxHadoopMakeDir(airDataDir)
rxHadoopCopyFromLocal("/tmp/airOT2012*.csv", airDataDir)
```

The original CSV files have rather unwieldy variable names, so we supply a colInfo list to make them more manageable (we won't use all of these variables in this manual, but you will use the data sources created in this manual as you continue to explore distributed computing in the *RevoScaleR Distributed Computing Guide* (RevoScaleR_Distributed_Computing.pdf):

```
airlineColInfo <- list(
     MONTH = list(newName = "Month", type = "integer"),
    DAY_OF_WEEK = list(newName = "DayOfWeek", type = "factor",
        levels = as.character(1:7),
        newLevels = c("Mon", "Tues", "Wed", "Thur", "Fri", "Sat",
                      "Sun")),
    UNIQUE_CARRIER = list(newName = "UniqueCarrier", type =
                            "factor"),
    ORIGIN = list(newName = "Origin", type = "factor"),
    DEST = list(newName = "Dest", type = "factor"),
    CRS_DEP_TIME = list(newName = "CRSDepTime", type = "integer"),
    DEP_TIME = list(newName = "DepTime", type = "integer"),
    DEP_DELAY = list(newName = "DepDelay", type = "integer"),
    DEP_DELAY_NEW = list(newName = "DepDelayMinutes", type =
                         "integer"),
    DEP_DEL15 = list(newName = "DepDel15", type = "logical"),
    DEP_DELAY_GROUP = list(newName = "DepDelayGroups", type =
                           "factor",
       levels = as.character(-2:12),
       newLevels = c("< -15", "-15 to -1","0 to 14", "15 to 29",
                    "30 to 44", "45 to 59", "60 to 74",
                    "75 to 89", "90 to 104", "105 to 119",
                    "120 to 134", "135 to 149", "150 to 164",
                    "165 to 179", ">= 180")),
    ARR_DELAY = list(newName = "ArrDelay", type = "integer"),
    ARR_DELAY_NEW = list(newName = "ArrDelayMinutes", type =
                         "integer"),
    ARR_DEL15 = list(newName = "ArrDel15", type = "logical"),
    AIR_TIME = list(newName = "AirTime", type =  "integer"),
    DISTANCE = list(newName = "Distance", type = "integer"),
    DISTANCE_GROUP = list(newName = "DistanceGroup", type =
                         "factor",
     levels = as.character(1:11),
     newLevels = c("< 250", "250-499", "500-749", "750-999",
         "1000-1249", "1250-1499", "1500-1749", "1750-1999",
         "2000-2249", "2250-2499", ">= 2500")))

varNames <- names(airlineColInfo)
```

We create a data source using these definitions as follows:

```
    hdfsFS <- RxHdfsFileSystem()
    bigAirDS <- RxTextData( airDataDir,
                            colInfo = airlineColInfo,
                            varsToKeep = varNames,
                            fileSystem = hdfsFS )
```

6.2 Estimating a Linear Model with a Big Data Set

We fit our first model to the large airline data much as we created the linear model for the AirlineDemoSmall data, and we time it to see how long it takes to fit the model on this large data set:

```
    system.time(
         delayArr <- rxLinMod(ArrDelay ~ DayOfWeek, data = bigAirDS,
              cube = TRUE)
    )
```

To see a summary of your results:

```
    print(
         summary(delayArr)
    )
```

You should see the following results:

```
    Call:
    rxLinMod(formula = ArrDelay ~ DayOfWeek, data = bigAirDS, cube = TRUE)

    Cube Linear Regression Results for: ArrDelay ~ DayOfWeek
    Data: bigAirDS (RxTextData Data Source)
    File name: /share/airOnTime12/CSV
    Dependent variable(s): ArrDelay
    Total independent variables: 7
    Number of valid observations: 6005381
    Number of missing observations: 91381

    Coefficients:
                  Estimate Std. Error t value Pr(>|t|)     | Counts
    DayOfWeek=Mon  3.54210    0.03736   94.80 2.22e-16 *** | 901592
```

```
DayOfWeek=Tues  1.80696    0.03835   47.12 2.22e-16 *** | 855805
DayOfWeek=Wed   2.19424    0.03807   57.64 2.22e-16 *** | 868505
DayOfWeek=Thur  4.65502    0.03757  123.90 2.22e-16 *** | 891674
DayOfWeek=Fri   5.64402    0.03747  150.62 2.22e-16 *** | 896495
DayOfWeek=Sat   0.91008    0.04144   21.96 2.22e-16 *** | 732944
DayOfWeek=Sun   2.82780    0.03829   73.84 2.22e-16 *** | 858366
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 35.48 on 6005374 degrees of freedom
Multiple R-squared: 0.001827 (as if intercept included)
Adjusted R-squared: 0.001826
F-statistic:  1832 on 6 and 6005374 DF,  p-value: < 2.2e-16
Condition number: 1
```

Notice that the results indicate we have processed all the data, six million observations, using all the .csv files in the specified directory. Notice also that because we specified `cube = TRUE`, we have an estimated coefficient for each day of the week (and not the intercept).

6.3 Importing Data As Composite XDF Files

As we have seen, you can analyze CSV files directly with RevoScaleR on Hadoop, but the analysis can be done more quickly if the data is stored in a more efficient format. The RevoScaleR .xdf format is extremely efficient, but is modified somewhat for HDFS so that individual files remain within a single HDFS block. (The HDFS block size varies from installation to installation but is typically either 64MB or 128MB.) When you use rxImport on Hadoop, you specify an RxTextData data source such as bigAirDS as the inData and an RxXdfData data source with fileSystem set to an HDFS file system as the outFile argument to create a set of *composite .xdf files*. The RxXdfData object can then be used as the data argument in subsequent RevoScaleR analyses.

For our airline data, we will define our RxXdfData object as follows (the second "<user>" should be replaced by your actual user name on the Hadoop cluster):

```
bigAirXdfName <- "/user/RevoShare/<user>/AirlineOnTime2012"
airData <- RxXdfData( bigAirXdfName,
                          fileSystem = hdfsFS )
```

We set a block size of 250000 rows and specify that we will read all the data by specifying `numRowsToRead = -1`:

```
blockSize <- 250000
numRowsToRead = -1
```

We then import the data using `rxImport`:

```
rxImport(inData = bigAirDS,
         outFile = airData,
         rowsPerRead = blockSize,
         overwrite = TRUE,
         numRows = numRowsToRead )
```

6.4 Estimating a Linear Model Using Composite XDF Files

Now we can re-estimate the same linear model, using the new, faster data source:

```
system.time(
     delayArr <- rxLinMod(ArrDelay ~ DayOfWeek, data = airData,
          cube = TRUE)
)
```

To see a summary of your results:

```
print(
     summary(delayArr)
)
```

You should see the following results (which should match the results we found for the CSV files above):

```
Call:
rxLinMod(formula = ArrDelay ~ DayOfWeek, data = airData, cube = TRUE)

Cube Linear Regression Results for: ArrDelay ~ DayOfWeek
Data: airData (RxXdfData Data Source)
File name: /user/RevoShare/v7alpha/AirlineOnTime2012
Dependent variable(s): ArrDelay
Total independent variables: 7
Number of valid observations: 6005381
Number of missing observations: 91381

Coefficients:
               Estimate Std. Error t value Pr(>|t|)     | Counts
DayOfWeek=Mon   3.54210    0.03736   94.80 2.22e-16 *** | 901592
DayOfWeek=Tues  1.80696    0.03835   47.12 2.22e-16 *** | 855805
DayOfWeek=Wed   2.19424    0.03807   57.64 2.22e-16 *** | 868505
DayOfWeek=Thur  4.65502    0.03757  123.90 2.22e-16 *** | 891674
DayOfWeek=Fri   5.64402    0.03747  150.62 2.22e-16 *** | 896495
DayOfWeek=Sat   0.91008    0.04144   21.96 2.22e-16 *** | 732944
DayOfWeek=Sun   2.82780    0.03829   73.84 2.22e-16 *** | 858366
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 35.48 on 6005374 degrees of freedom
Multiple R-squared: 0.001827 (as if intercept included)
Adjusted R-squared: 0.001826
F-statistic:  1832 on 6 and 6005374 DF,  p-value: < 2.2e-16
Condition number: 1
```

6.5 Handling Larger Linear Models

The data set contains a variable `UniqueCarrier` which contains airline codes for 15 carriers. Because the RevoScaleR Compute Engine handles factor variables so efficiently, we can do a linear regression looking at the Arrival Delay by Carrier. This will take a little longer, of course, than the previous analysis, because we are estimating 15 instead of 7 parameters.

```
delayCarrier <- rxLinMod(ArrDelay ~ UniqueCarrier,
     data = airData, cube = TRUE)
```

Next, sort the coefficient vector so that the we can see the airlines with the highest and lowest values.

```
dcCoef <- sort(coef(delayCarrier))
```

Next, see how the airlines rank in arrival delays:

```
print(
     dcCoef
)
```

The result is:

```
UniqueCarrier=AS UniqueCarrier=US UniqueCarrier=DL UniqueCarrier=FL
      -1.9022483       -1.2418484       -0.8013952       -0.2618747
UniqueCarrier=HA UniqueCarrier=YV UniqueCarrier=WN UniqueCarrier=MQ
       0.3143564        1.3189086        2.8824285        2.8843612
UniqueCarrier=OO UniqueCarrier=VX UniqueCarrier=B6 UniqueCarrier=UA
       3.9846305        4.0323447        4.8508215        5.0905867
UniqueCarrier=AA UniqueCarrier=EV UniqueCarrier=F9
       6.3980937        7.2934991        7.8788931
```

Notice that Alaska Airlines comes in as the best in on-time arrivals, while Frontier is at the bottom of the list. We can see the difference by subtracting the coefficients:

```
print(
sprintf("Frontier's additional delay compared with Alaska: %f",
     dcCoef["UniqueCarrier=F9"]-dcCoef["UniqueCarrier=AS"])
)
```

which results in:

```
[1] "Frontier's additional delay compared with Alaska: 9.781141"
```

6.6 A Large Data Logistic Regression

Our airline data includes a logical variable, ArrDel15, that tells whether a flight is 15 or more minutes late. We can use this as the response for a logistic regression to model the likelihood that a flight will be late given the day of the week:

```
logitObj <- rxLogit(ArrDel15 ~ DayOfWeek, data = airData)
logitObj
```

which results in:

```
Logistic Regression Results for: ArrDel15 ~ DayOfWeek
Data: airData (RxXdfData Data Source)
File name: AirOnTime2012.xdf
Dependent variable(s): ArrDel15
Total independent variables: 8 (Including number dropped: 1)
Number of valid observations: 6005381
Number of missing observations: 91381

Coefficients:
                  ArrDel15
(Intercept)    -1.61122563
DayOfWeek=Mon   0.04600367
DayOfWeek=Tues -0.08431814
DayOfWeek=Wed  -0.07319133
DayOfWeek=Thur  0.12585721
DayOfWeek=Fri   0.18987931
DayOfWeek=Sat  -0.13498591
DayOfWeek=Sun      Dropped
```

6.7 Prediction on Large Data

You can predict (or score) from a fitted model on Hadoop using *rxPredict*. In this example we will compute predicted values and their residuals for the logistic regression in the previous section. We can update the airData to include the predicted values and residuals by not specifying an *outData* argument, which is *NULL* by default.

```
rxPredict(modelObject=logitObj, data=airData,
      computeResiduals=TRUE, overwrite=TRUE, writeModelVars=TRUE)
```

By putting in a call to *rxGetVarInfo* we can see that two additional variables, *ArrDel15_Pred* and *ArrDel15_Resid* were added to the *airData* composite XDF.

If instead we wanted to have a separate output data set containing the predicted values and residuals, we could set the *outData* to an XDF data source. An *RxTextData* object can be used as the input data for *rxPredict* on Hadoop, but only XDF can be written to HDFS. Moreover, when using a CSV file or directory of CSV files as the input data the *outData* argument must be set to an *RxXdfData* object.

6.8 Performing Data Operations on Large Data

To create or modify data in HDFS on Hadoop we can use the *rxDataStep* function. Suppose we want to repeat the analyses with a “cleaner” version of the large airline dataset. To do this we will first need to remove the new prediction variables that we added in the previous section and keep only the flights having arrival delay information and flights that did not depart more than one hour early. We can put a call to *rxDataStep* to output a new composite XDF to HDFS. (As in an earlier example, the second “user” in the newAirDir path should be changed to the actual user name of your Hadoop user.)

```
newAirDir <- "/user/RevoShare/<user>/newAirData"
```

```
newAirXdf <- RxXdfData(newAirDir,fileSystem=hdfsFS)

rxDataStep(inData = airData, outFile = newAirXdf,
           varsToDrop=c("ArrDel15_Pred","ArrDel15_Resid"),
           rowSelection = !is.na(ArrDelay) & (DepDelay > -60))
```

As with *rxPredict*, an *RxTextData* object can be used as the input data for *rxDataStep* on Hadoop, but only a composite XDF can be written out. In this case, an *outFile* must be specified. To modify an existing composite XDF using *rxDataStep* set the *overwrite* argument to *TRUE* and either omit the *outFile* argument or set it to the same composite XDF data source specified for *inData*.

6.9 Writing to CSV in HDFS

If you converted your CSV to XDF, as in section 6.3, to take advantage of the efficiency while running analyses, but now wish to convert your data back to CSV you can do so using *rxDataStep*. (This feature is still experimental.) To create a folder of CSV files, first create an RxTextData object using a directory name as the file argument; this represents the folder in which to create the CSV files. This directory will be created when you run the *rxDataStep*.Then, point to this *RxTextData* object in the *outFile* argument of the *rxDataStep*. Each CSV created will be named based on the directory name and followed by a number.

Suppose we want to write out a folder of CSV in HDFS from our airData composite XDF after we performed the logistic regression and prediction, so that the new CSV files contain the predicted values and residuals. We can do this as follows:

```
airDataCsvDir <- file.path("/user/RevoShare/<user>/airDataCsv")
airDataCsvDS <- RxTextData(airDataCsvDir,fileSystem=hdfsFS)
rxDataStep(inData=airData, outFile=airDataCsvDS)
```

You will notice that the *rxDataStep* wrote out one CSV for every .xdfd file in the input composite XDF file. This is the default behaviour for writing CSV from composite XDF to HDFS when the compute context is set to *RxSpark*.

Alternatively, you could switch your compute context back to “local” when you are done performing your analyses and take advantage of two arguments within *RxTextData* that give you slightly more control when writing out CSV files to HDFS: *createFileSet* and *rowsPerOutFile*. When *createFileSet* is set to TRUE a folder of CSV files is written to the directory you specify. When *createFileSet* is set to FALSE a single CSV file is written. The second argument, *rowsPerOutFile*, can be set to an integer to indicate how many rows to write to each CSV file when *createFileSet* is TRUE. Returning to the example above, suppose we wanted to write out a folder of CSVs but we wanted to write out the airData but into only 6 CSV files.

```
rxSetComputeContext("local")
airDataCsvRowsDir <-
      file.path("/user/RevoShare/<user>/airDataCsvRows")
```

```
airDataCsvRowsDS <- RxTextData(airDataCsvRowsDir,
      fileSystem=hdfsFS, createFileSet=TRUE,
      rowsPerOutFile=1000000)
rxDataStep(inData=airData, outFile=airDataCsvRowsDS)
```

Note that when using a `RxSpark` compute context, createFileSet defaults to TRUE and rowsPerOutFile has no effect. Thus if you wish to create a single CSV or customize the number of rows per file you must perform the rxDataStep in a local compute context (data can still be in HDFS).

Now set the compute context back to the Hadoop cluster for further analyses:

```
rxSetComputeContext(myHadoopCluster)
```

7 Parallel Partial Decision Forests

As mentioned in Section 2, both `rxDForest` and `rxBTrees` are available on Hadoop--these provide two different methods for fitting classification and regression decision forests. In the default case, both algorithms generate multiple stages in the Spark job, and thus can tend to incur significant overhead, particularly with smaller data sets. However, the `scheduleOnce` argument to both functions allows the computation to be performed via `rxExec`, which generates only a single stage in the Spark job, and thus incurs significantly less overhead. When using the `scheduleOnce` argument, you can specify the number of trees to be grown within each `rxExec` task using the forest function's `nTree` argument together with `rxExec's rxElemArgs` function, as in the following regression example using the built-in claims data:

```
file.name <- "claims.xdf"
sourceFile <- system.file(file.path("SampleData", file.name),
    package="RevoScaleR")
inputDir <- "/share/claimsXdf"
rxHadoopMakeDir(inputDir)
rxHadoopCopyFromLocal(sourceFile, inputDir)
input <- RxXdfData(file.path(inputDir, file.name,
    fsep="/"),fileSystem=hdfsFS)

partial.forests <-rxDForest(formula = age ~ car.age +
    type + cost + number, data = input,
    minSplit = 5, maxDepth = 2,
    nTree = rxElemArg(rep(2,8)), seed = 0,
    maxNumBins = 200, computeOobError = -1,
    reportProgress = 2, verbose = 0,
    scheduleOnce = TRUE)
```

Equivalently, you can use `nTree` together with `rxExec's timesToRun` argument:

```
partial.forests <-rxDForest(formula = age ~ car.age +
    type + cost + number, data = input,
    minSplit = 5, maxDepth = 2,
    nTree = 2, seed = 0,
    maxNumBins = 200, computeOobError = -1,
    reportProgress = 2, verbose = 0,
    scheduleOnce = TRUE, timesToRun = 8)
```

In this example, using `scheduleOnce` can be multiple times faster than the default, and so we recommend it for decision forests with small to moderate-sized data sets.

Similarly, the `rxPredict` methods for `rxDForest` and `rxBTrees` objects include the `scheduleOnce` argument, and should be used when using these methods on small to moderate-sized data sets.

8 Cleaning up Data

You can run the following commands to clean up data in this tutorial:

```
rxHadoopRemoveDir("/share/AirlineDemoSmall")
rxHadoopRemoveDir("/share/airOnTime12")
rxHadoopRemoveDir("/share/claimsXdf")
rxHadoopRemoveDir("/user/RevoShare/<user>/AirlineOnTime2012")
rxHadoopRemoveDir("/user/RevoShare/<user>/airDataCsv")
rxHadoopRemoveDir("/user/RevoShare/<user>/airDataCsvRows")
rxHadoopRemoveDir("/user/RevoShare/<user>/newAirData")
```

9 Continuing with Distributed Computing

With the linear model and logistic regression performed in the previous sections, you have seen a taste of high-performance analytics on the Hadoop Spark platform. You are now ready to continue with the *RevoScaleR Distributed Computing Guide* (RevoScaleR_Distributed_Computing.pdf), which continues the analysis of the 2012 airline on-time data with examples for all of RevoScaleR's HPA functions. You will find this analysis in Chapter 3 of the guide, Running Distributed Analyses.

The *Distributed Computing Guide* also provides more examples of using non-waiting compute contexts, including managing multiple jobs, and examples of using rxExec to perform traditional high-performance computing, including Monte Carlo simulations and other embarrassingly parallel problems.