Analysing data in SQL Server 16, combining R and SQL

# Overview

R is the most popular programming language for statistics and machine learning, and SQL is the lingua franca for the data manipulation. Dealing with an advanced analytics scenario, we need to pre-process the data and to build machine learning models. A good solution consists in using each tool for the purpose it’s designed for: the standard data preparation using a tool supporting SQL, the custom data preparation and the machine learning models using R.

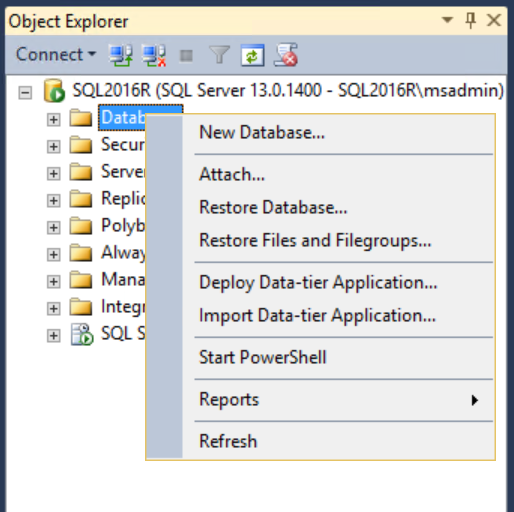
**SQL Server 2016** has the option to include an extension: **Microsoft R Services**. It’s based on **MRS** (Microsoft R Server), a tool designed by Revolution Analytics to scale R across large data volumes. In SQL Server 2016, R Services provides a direct interface between MRS and the SQL databases. In this way, it’s possible to analyse the SQL tables and to create new tables using the MRS tools. In addition, the R package RODBC allows to run SQL queries from R.

This article shows a simple example integrating R and SQL. To follow all the steps, some prior knowledge about the basic R functions is required.

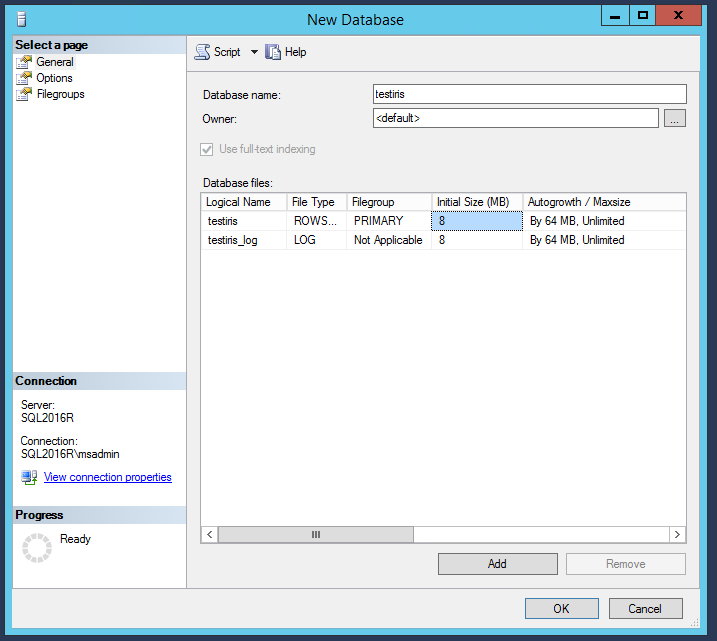
# Setting-up the environment

In this article, we are using a SQL Server 2016 VM. To set-up the environment, the steps are

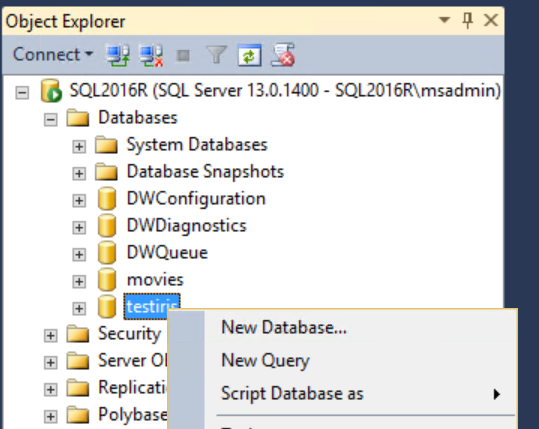
1. Log-in to Azure and set-up a SQL Server machine with R Services. To install R Services, refer to this: <https://msdn.microsoft.com/en-us/library/mt696069.aspx>
2. Connect to SQL Server: from the Windows task bar, open SQL Server Management Studio and connect to it using your credentials.
3. Create a new database: right-click on Databases and choose *“New Database…”.*



1. Give a name to the new database, e.g. *iristest*. Then, click on *Add* and *OK*.



1. Open the query editor: right-click on the new database and choose “*New Query*”.



1. Define the credentials to authenticate to the database: in the editor, paste and run the following query. You should replace the parameter *testiris* with the name of your database, and *username* and *password* with new credentials to access this database.

*USE [testiris]*

*GO*

*CREATE LOGIN [username] WITH PASSWORD='password', CHECK\_EXPIRATION=OFF,*

*CHECK\_POLICY=OFF;*

*CREATE USER [username] FOR LOGIN [username] WITH DEFAULT\_SCHEMA=[db\_datareader]*

*ALTER ROLE [db\_datareader] ADD MEMBER [username]*

*ALTER ROLE [db\_ddladmin] ADD MEMBER [username]*

1. Build a new RStudio or Visual Studio project and create a new R script. See <https://www.rstudio.com/products/RStudio/>
2. Define a list with the parameters to connect to the database. They should be the same as in the previous SQL query.

*sql\_par <- list(*

*Driver = "SQL Server",*

*Server = ".",*

*Database = "testiris",*

*Uid = "username",*

*Pwd = "password"*

*)*

1. Starting from the list of parameters, define a connection string for the SQL database. For this purpose, you can use the function *paste*.

*sql\_connect <- paste(names(sql\_par), unlist(sql\_par),*

*sep = "=", collapse = ";")*

1. Define a MRS data source object containing the information to access the *iristest* database. The inputs are the table name, e.g. *iris*, and the connection string.

*table\_iris <- "iris"*

*sql\_iris <- RxSqlServerData(connectionString = sql\_connect,*

*table = table\_iris)*

1. Import the data into a SQL table. For this purpose, we can use *rxDataStep*. Its inputs are the local dataframe *iris*, the data source object *sql\_iris*, and *overwrite = TRUE*, specifying that we are overwriting the table if existing already. To avoid issues in the SQL queries, we also re-name the columns of *iris*, replacing the “*.”* with an “*\_”*.

*names(iris) <- gsub("[.]", "\_", names(iris))*

*rxDataStep(iris, sql\_iris, overwrite = TRUE)*

1. Install and load the package *RODBC* and initialise it using the connection string. We define the object *channel* containing the information to access the database.

*install.packages("RODBC")*

*library(RODBC)*

*channel <- RODBC::odbcDriverConnect(sql\_connect)*

# Exploring the data

To explore the data, we can either use MRS or SQL. In MRS, there is the option to quickly access the metadata using *rxGetVarInfo*.

*rxGetVarInfo(sql\_iris)*

Var 1: Sepal\_Length, Type: numeric

Var 2: Sepal\_Width, Type: numeric

Var 3: Petal\_Length, Type: numeric

Var 4: Petal\_Width, Type: numeric

Var 5: Species, Type: character

The output contains the name and type of each field.

Also, it’s possible to compute a quick summary of the table using *rxSummary*. The inputs are

* *Formula*: what variables we want to summarise, using the *formula* syntax. For more info about the formula, have a look at the related material typing *?formula* into the R console. To include all the variables, we can use *“.”*.
* *Data*: the data source object, containing the information to access the data.

*summary\_iris <- rxSummary(~ ., sql\_iris)*

*summary\_iris$sDataFrame*

Name Mean StdDev Min Max ValidObs

1 Sepal\_Length 5.843333 0.8280661 4.3 7.9 150

2 Sepal\_Width 3.057333 0.4358663 2.0 4.4 150

3 Petal\_Length 3.758000 1.7652982 1.0 6.9 150

4 Petal\_Width 1.199333 0.7622377 0.1 2.5 150

5 Species NA NA NA NA 0

MissingObs

1 0

2 0

3 0

4 0

5 0

This summary contains some basic statistics like the mean and the standard deviation.

Using *RODBC*, we can run any SQL query. The inputs are the connection object *channel*, defined previously, and the query string. Let’s see a simple example.

*RODBC::sqlQuery(channel, "select count(\*) from iris")*

1 150

As expected, the output is the number of records.

R contains good string manipulation tools that allow us to build complex queries and to write extensions. To show the approach, we perform a simple group by operation. The steps are

1. Define the skeleton of the query, leaving the column names as *%s* parameters
2. Define the parameters
3. Define the query inclusive of its parameters using *sprintf*
4. Run the query

The R code is

*query\_sql <- "select %s, avg(%s) as avg\_sl from iris group by %s"*

*col\_group <- "Species"*

*col\_value <- "Sepal\_Length"*

*query\_sql <- sprintf(query\_sql, col\_group, col\_value, col\_group)*

*df\_avg\_value <- RODBC::sqlQuery(channel, query\_sql)*

*df\_avg\_value*

Species avg\_sl

1 setosa 5.006

2 versicolor 5.936

3 virginica 6.588

Using a similar approach, we can define more complex queries. Also, if we need to run similar queries many times, it’s possible to build R functions that build the query depending on some given parameters.

# Processing the data

To process the data, it’s possible to use both MRS and SQL.

With MRS, we can process the data using the function *rxDataStep*. This function allows to build custom data processing operations. A simple example consists in defining a new column *Sepal.Size* with simple maths operations. The steps are

1. Define a new SQL data source object
2. Define the SQL table using *RxSqlServerData*. Its arguments are the input and output data object, and *overwrite = TRUE*, similarly to before. In addition, we define the new column using the *transforms* input.
3. Have a look at the new table using *rxGetVarInfo*

*table\_iris\_2 <- "iris2"*

*sql\_iris\_2 <- RxSqlServerData(connectionString = sql\_connect,*

*table = table\_iris\_2)*

*rxDataStep(inData = sql\_iris,*

*outFile = sql\_iris\_2,*

*transforms = list(Sepal.Size = Sepal.Length \* Sepal.Width),*

*overwrite = TRUE)*

*rxGetVarInfo(sql\_iris\_2)*

Var 1: Sepal\_Length, Type: numeric

Var 2: Sepal\_Width, Type: numeric

Var 3: Petal\_Length, Type: numeric

Var 4: Petal\_Width, Type: numeric

Var 5: Species, Type: character

Var 6: Sepal\_Size, Type: numeric

As expected, we have a new column: *Sepal\_Size*.

Using the same approach, it’s possible to build complex custom operations, making use of the open-source R functions.

Using RODBC, it’s possible to run an SQL query defining a new table. For instance, we can join two tables. Using an approach similar to the previous SQL query, we build the related string and execute it using *RODBC*.

*table\_avg\_value <- "avg\_value"*

*sql\_avg\_value <- RxSqlServerData(connectionString = sql\_connect,*

*table = table\_avg\_value)*

*rxDataStep(df\_avg\_value, sql\_avg\_value, overwrite = TRUE)*

*query\_join <- "select \* from iris*

*left join avg\_value*

*on iris.Species = avg\_value.Species"*

*df\_join <- RODBC::sqlQuery(channel, query\_join)*

*head(df\_join)*

Sepal\_Length Sepal\_Width Petal\_Length Petal\_Width

1 5.1 3.5 1.4 0.2

2 4.9 3.0 1.4 0.2

3 4.7 3.2 1.3 0.2

4 4.6 3.1 1.5 0.2

5 5.0 3.6 1.4 0.2

6 5.4 3.9 1.7 0.4

Species Species.1 avg\_sl

1 setosa setosa 5.006

2 setosa setosa 5.006

3 setosa setosa 5.006

4 setosa setosa 5.006

5 setosa setosa 5.006

6 setosa setosa 5.006

After having processed the data, it’s possible to build machine learning models starting from the SQL tables. In this way, we don’t need to pull the data in-memory and we can deal with a large data volume. For instance, to build a linear regression, we can use *rxLinMod*. Similarly to rxSummary, the arguments are *formula*, defining the variables to include, and *data*, defining the data source. After having built the model, we can explore is using *summary*, similarly to open-source R.

*model\_lm <- rxLinMod(formula = Petal\_Length ~ Sepal\_Length + Petal\_Width,*

*data = sql\_iris)*

*summary(model\_lm)*

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) -1.50714 0.33696 -4.473 1.54e-05 \*\*\*

Sepal\_Length 0.54226 0.06934 7.820 9.41e-13 \*\*\*

Petal\_Width 1.74810 0.07533 23.205 2.22e-16 \*\*\*

The output is the same as using open-source R function *lm*. The difference is that we can apply this function on a large table without needing to pull it in-memory.

# Conclusions

SQL and MRS are designed for different purposes. Having them into the same environment, it’s possible to use each of them in the most proper context: SQL to prepare the data, MRS to build advanced machine learning models. Also, the string manipulation tools provided by R allow to build SQL queries, making it possible to write code extensions. Another option is to include R code within the SQL queries although we haven’t dealt with that in this article.