USE IBM IN-DATABASE ANALYTICS WITH R

M. WURST, C. BLAHA, IBM GERMANY RESEARCH AND DEVELOPMENT

Introduction

To process data, most native R functions require that the data first is extracted from a database to working memory. However, If you need to analyze a large amount of data, that is often impractical or even impossible. The ibmdbR package provides methods to make working with databases more efficient by seamlessly pushing operations of R into the underlying database for execution. This not only lifts the memory limit of R, it also allows users to profit from performance-enhancing features of the underlying database management system, such as columnar technology and parallel processing, without having to interact with the database explicitly. Keeping the data in the database also avoids security issues that are associated with extracting data and ensures that the data that is being analyzed is as current as possible. Some in-database functions additionally use lazy loading to load only those parts of the data that are actually required, to further increase efficiency.

Prerequisites

The ibmdbR package is designed to work with IBM DB2© for Linux and Windows Version 10.5 (in the following abbreviated to DB2), as well as IBM dashDB© (in the following dashDB). Before you start, you must install DB2 or provision an instance of dashDB (see „Further Reading“).

If you use dashDB from your web brower, everything is pre-configured and ready to be used, so you can skip directly to the instructions in „Getting Started“.

If you connect to a DB2 server or if you want to use dashDB remotely, you must first install the appropriate client driver packages. For DB2, the driver packages come with the product. For IBM dashDB, they can be downloaded from the dashDB web console. After you install the driver packages you must configure an ODBC source. Refer to the documentation of your operating system for information on how to perform this task. In the following we assume you created an ODBC data source called „BLUDB“.

Getting Started

Before you can use the any of the push-down functions of the ibmdbR package, you must connect to a database. This is done by executing the idaConnect function. Specify the name of the ODBC source, the user name and the password as parameters. If you use R from the dashDB web console, the following statement will connect you directly to the database BLUDB:

```
> con <- idaConnect('BLUDB','','')
```

Next, initialize the in-database functionality by executing the idaInit function. The idaInit function creates a singleton for the connection object such that you do not need to pass it as parameter later on:

```
> idaInit(con)
```

Now, we can try to issue a first command. The idaShowTables function will return a *data.frame* that contains a list of all tables in the current schema:

```
> idaShowTables()
    Schema   Name  Owner  Type
1   USER1    A     USER1   T
2   USER1    B     USER1   T
```

Normally, at the end of each session, you would close the connection to the database:

```
idaClose(con)
```

However, to run the samples in the next sections, we will keep it open.

Working with ida.data.frame

Instead of holding data in memory, an object of class *ida.data.frame* contains only a reference to a table or a query. Many operations can be performed without loading the content of this table or query into memory. You can create an object of class *ida.data.frame* either by pointing to an existing table in the database or by uploading a local R *data.frame* to a table. For example, if there already is a table 'IRIS' in the database, you can create an *ida.data.frame* object by executing the following statement:

```
> iris.ida <- ida.data.frame('IRIS')
```

If the table 'IRIS' does not yet exist, you can create an *ida.data.frame* object by uploading local data:

```
> iris.ida <- as.ida.data.frame(iris,'IRIS')
```

To check the class and print the *ida.data.frame* object, execute the following statements:

```
> class(iris.ida)
'ida.data.frame'

> iris.ida
SELECT "SepalLength","SepalWidth","PetalLength","PetalWidth","Species" FROM "IRIS"
```

Optimally, R users should not need to care about SQL too much. For this reason, the ibmdbR package overwrites many methods and functions defined for regular R *data.frame* objects. It uses SQL to push the execution of these methods down into the database. The *dim* method is a simple example:

```
> dim(iris.ida)
[1] 150   5
```

It calculates the dimensions of the *ida.data.frame*, just as the *dim* method would do for a regular *data.frame*. Internally, however, it executes, among other statements, a *SELECT COUNT(*) FROM IRIS* SQL query. Another example is the *head* method, which retrieves only the first rows from a *ida.data.frame* by executing the appropriate SQL statements in the background:

```
> head(iris.ida)
  SepalLength SepalWidth PetalLength PetalWidth Species
1         5.1        3.5         1.4        0.2  setosa
2         4.9        3.0         1.4        0.2  setosa
3         4.7        3.2         1.3        0.2  setosa
4         4.6        3.1         1.5        0.2  setosa
5         5.0        3.6         1.4        0.2  setosa
6         5.4        3.9         1.7        0.4  setosa
```

The ibmdbR package overwrites several other methods (defined on the *data.frame* class) in a similar way. Among these are *as.data.frame*, *sd*, *max*, *mean*, *min*, *length*, *print*, *names*, *colnames*, *summary*, *NROW*, *NCOL*, *var*, *cor* and *cov*.

Sometimes, you might not want to work on a full table but only on a selection. You can do this in a way that is similar to the way you would do this for a regular *data.frame*. For example, the following statements, would select only rows for which the column 'Species' equals 'setosa', and only the columns 'PetalLength' and'PetalWidth':

```
> iris.ida2 <- iris.ida[iris.ida$Species=='setosa',c('PetalLength','PetalWidth')]
> dim(iris.ida2)
[1] 50  2
```

All methods and functions that are applied to an *ida.data.frame* object with selection will reflect it, which is why the *dim* method returns 50 rows instead of 150 rows in the previous example.

Selection can also be useful to remove missing values. The following statement keeps all columns but removes rows for which column 'Species' is NULL:

```
iris.ida3 <- iris.ida[!db.is.null(iris.ida$Species),]
[1] 150   5
```

As the IRIS table does not have any missing values, the *dim* method returns 150 rows.

Instead of selecting columns, you can also add new columns, that are based on existing ones. These columns are kept locally and are only materialized at query time. The following statements show a few examples:

```
> iris.ida$X <- iris.ida$SepalLength+iris.ida$SepalWidth
> iris.ida$Y <- ifelse((iris.ida$SepalLength>4)&(iris.ida$SepalWidth<3),'a','b')
> iris.ida$Z <- as.character(iris.ida$SepalLength)
> head(iris.ida)
  SepalLength SepalWidth PetalLength PetalWidth Species   X Y     Z
1         5.1        3.5         1.4        0.2  setosa 8.6 a 5.1E0
2         4.9        3.0         1.4        0.2  setosa 7.9 b 4.9E0
3         4.7        3.2         1.3        0.2  setosa 7.9 b 4.7E0
4         4.6        3.1         1.5        0.2  setosa 7.7 b 4.6E0
5         5.0        3.6         1.4        0.2  setosa 8.6 b 5.0E0
6         5.4        3.9         1.7        0.4  setosa 9.3 b 5.4E0
```

If you want to store these columns to the database (for example, to make them accessible to others), you can use the idaCreateView function to create a view of an *ida.data.frame* object.

Preprocess and Analyze Data

The ibmdbR package provides several functions for preprocessing and statistical analysis.

The idaSample function draws a (stratified) sample from an *ida.data.frame*. In the following statement, 'Species' is the stratification column.

```
> df <- idaSample(iris.ida,6,'Species')
> df
  SepalLength SepalWidth PetalLength PetalWidth    Species   X Y     Z
1         6.4        2.7         5.3        1.9  virginica 9.1 a 6.4E0
2         5.8        2.7         4.1        1.0 versicolor 8.5 a 5.8E0
3         5.0        3.3         1.4        0.2     setosa 8.3 b 5.0E0
4         6.3        2.9         5.6        1.8  virginica 9.2 a 6.3E0
5         6.1        3.0         4.6        1.4 versicolor 9.1 a 6.1E0
6         5.1        3.3         1.7        0.5     setosa 8.4 a 5.1E0
```

The idaSample function can also be applied to *ida.data.frame* objects with column or row selection or with defined columns. The following example would excludes all rows with 'Species' equal to 'setosa':

```
> df <- idaSample(iris.ida[iris.ida$Species!='setosa',],4,'Species')
> df
  SepalLength SepalWidth PetalLength PetalWidth    Species    X Y     Z
1         7.1        3.0         5.9        2.1  virginica 10.1 a 7.1E0
2         6.2        2.9         4.3        1.3 versicolor  9.1 a 6.2E0
3         5.5        2.6         4.4        1.2 versicolor  8.1 a 5.5E0
4         6.7        3.0         5.2        2.3  virginica  9.7 a 6.7E0
```

For linear regression, the idaLm function is provided. It is very similar to the lm function for this purpose, but can be easily applied to millions of rows. The following statement calculates a linear regression model on an *ida.data.frame* object:

```
l <- idaLm(SepalLength~SepalWidth+PetalLength,iris.ida)
l$coefficients
                [,1]
SepalWidth  0.5955247
```

```
PetalLength 0.4719200
Intercept   2.2491402
attr(,"names")
[1] "SepalWidth"  "PetalLength" "Intercept"
```

Other functions include, for instance, idaTable for cross tabulation or idaMerge, for merging two *ida.data.frame* objects. The reference manual contains more details and examples.

Store and Share R Objects

Users often need to store objects in their workspace across two R sessions. The ibmdbR package provides methods that you can use to store R objects in database tables. This is achieved through a class called *ida.list*. An *ida.list* object can be used in a similar way as regular R *list*. However, the objects in an *ida.list* object are stored in database tables instead of being stored locally. There are three possible ways to initialize a *ida.list* object, depending on what information is to be accessed:

- Objects that are to be stored in a private container and are not to be readable by other users
  `l <- ida.list(type='private')`
- Objects that should be readable by all users of the database
  `l <- ida.list(type='public')`
- Public objects of another user
  `l <- ida.list(user='user2')`

The first two, when used for the first time, create the tables that are needed to hold the objects. The tables are created under the current schema:

Objects can be accessed using the R *list* operators:

```
> l <- ida.list(type='public')
> l['a'] <- 1:100
> l['b'] <- 'c'
> l['b']
"c"
> l['b'] <- NULL
```

All object keys can be listed using names

```
> names(l)
"a"
```

Objects are written to tables in a serialized format, so even though you can see the tables in your schema, you will usually not be able to read their contents directly.

Further Reading

The ibmdbR package allows you to seamlessly scale from small data sets to larger data sets, especially ones that no longer fit in memory. To learn more about the functionality of the package, refer to the reference documentation. Another good starting point is (IBM dashDB), which allows you to provision a database online within a few minutes. It contains many samples that can run online. To learn more about DB2 and how to install ODBC on your client machine, please refer to the *IBM DB2 10.5 Information center*.