Using DatabaseConnector

Martijn J. Schuemie

2019-02-21

Contents

1 Introduction

DatabaseConnector is an R package for connecting to various database platforms using Java's JDBC drivers.

Supported database platforms:

- Microsoft SQL Server
- Oracle
- PostgresSql
- Microsoft Parallel Data Warehouse (a.k.a. Analytics Platform System)
- Amazon Redshift
- Apache Impala
- Google BigQuery
- IBM Netezza
- SQLite

2 Obtaining drivers for BigQuery, Netezza and Impala

The package already contins most drivers, but because of licensing reasons the drivers for BigQuery, Netezza and Impala are not included but must be obtained by the user. Type

```
?jdbcDrivers
```

for instructions on how to download these drivers. Once downloaded, you can use the `pathToDriver` argument of the `connect`, `dbConnect`, and `createConnectionDetails` functions.

3 Creating a connection

To connect to a database a number of details need to be specified, such as the database platform, the location of the server, the user name, and password. We can call the `connect` function and specify these details directly:

```
conn <- connect(dbms = "postgresql",
                server = "localhost/postgres",
                user = "joe",
                password = "secret",
                schema = "cdm")
```

```
#> Connecting using PostgreSQL driver
```

See `?connect` for information on which details are required for each platform. Don't forget to close any connection afterwards:

```
disconnect(conn)
```

Note that, instead of providing the server name, it is also possible to provide the JDBC connection string if this is more convenient:

```
conn <- connect(dbms = "postgresql",
                connectionString = "jdbc:postgresql://localhost:5432/postgres",
                user = "joe",
                password = "secret",
                schema = "cdm")
```

```
#> Connecting using PostgreSQL driver
```

Sometimes we may want to first specify the connection details, and defer connecting until later. This may be convenient for example when the connection is established inside a function, and the details need to be passed as an argument. We can use the `createConnectionDetails` function for this purpose:

```
details <- createConnectionDetails(dbms = "postgresql",
                                   server = "localhost/postgres",
                                   user = "joe",
                                   password = "secret",
                                   schema = "cdm")
conn <- connect(details)
```

```
#> Connecting using PostgreSQL driver
```

3.1 Specifying the driver location for BigQuery, Netezza and Impala

For BigQuery, Netezza and Impala the drivers are not included in the `DatabaseConnector` package and need to be downloaded separately, as noted earlier. Once downloaded, we can point to the folder containing the jar files using the `pathToDriver` argument:

```
details <- createConnectionDetails(dbms = "netezza",
                                   server = "myserver.com/mainDb",
                                   user = "joe",
                                   password = "secret",
                                   schema = "cdm",
                                   pathToDriver = "c:/temp")
conn <- connect(details)
```

```
#> Connecting using Netezza driver
```

4 Querying

The main functions for querying database are the `querySql` and `executeSql` functions. The difference between these functions is that `querySql` expects data to be returned by the database, and can handle only one SQL statement at a time. In contrast, `executeSql` does not expect data to be returned, and accepts multiple SQL statments in a single SQL string.

Some examples:

```
querySql(conn, "SELECT TOP 3 * FROM person")
```

```
#>   PERSON_ID GENDER_CONCEPT_ID YEAR_OF_BIRTH
#> 1         1              8507          1975
#> 2         2              8507          1976
#> 3         3              8507          1977
```

```
executeSql(conn, "TRUNCATE TABLE foo; DROP TABLE foo; CREATE TABLE foo (bar INT);")
```

Both function provide extensive error reporting: When an error is thrown by the server, the error message and the offending piece of SQL are written to a text file to allow better debugging. The `executeSql` function also by default shows a progress bar, indicating the percentage of SQL statements that has been executed. If those attributes are not desired, the package also offers the `lowLevelQuerySql` and `lowLevelExecuteSql` functions.

4.1 Querying using ffdf objects

Sometimes the data to be fetched from the database is too large to fit into memory. In this case one can use the `ff` package to store R data objects on file, and use them as if they are available in memory. `DatabaseConnector` can download data directly into ffdf objects:

```
x <- querySql.ffdf(conn, "SELECT * FROM person")
```

Where x is now an ffdf object.

4.2 Querying different platforms using the same SQL

One challenge when writing code that is intended to run on multiple database platforms is that each platform has its own unique SQL dialect. To tackle this problem the SqlRender package was developed. SqlRender can translate SQL from a single starting dialect (SQL Server SQL) into any of the platforms supported by DatabaseConnector. The following convenience functions are available that first call the `render` and `translate` functions in `SqlRender`: `renderTranslateExecuteSql`, `renderTranslateQuerySql`, `renderTranslateQuerySql.ffdf`. For example:

```
persons <- renderTranslatequerySql(conn,
                                   sql = "SELECT TOP 10 * FROM @schema.person",
                                   schema = "cdm_synpuf")
```

Note that the SQL Server-specific 'TOP 10' syntax will be translated to for example 'LIMIT 10' on PostgreSQL, and that the SQL parameter `@schema` will be instantiated with the provided value 'cdm_synpuf'.

5 Inserting tables

Although it is also possible to insert data in the database by sending SQL statements using the `executeSql` function, it is often convenient and faster to use the `insertTable` function:

```
data(mtcars)
insertTable(conn, "mtcars", mtcars, createTable = TRUE)
```

In this example, we're uploading the mtcars data frame to a table called 'mtcars' on the server, that will be automatically created.

6 DBI interface

`DatabaseConnector` implements the DBI interface for compatability with other R packages. One can use the DBI functions instead of the ones described before, for example:

```
conn <- dbConnect(DatabaseConnectorDriver(),
                  dbms = "postgresql",
                  server = "localhost/postgres",
                  user = "joe",
                  password = "secret",
                  schema = "cdm")
```

```
#> Connecting using PostgreSQL driver
```

```
dbIsValid(conn)
```

```
#> [1] TRUE
```

```
res <- dbSendQuery(conn, "SELECT TOP 3 * FROM person")
dbFetch(res)
```

```
#>   PERSON_ID GENDER_CONCEPT_ID YEAR_OF_BIRTH
#> 1         1              8507          1975
#> 2         2              8507          1976
#> 3         3              8507          1977
```

```
dbHasCompleted(res)
```

```
#> [1] TRUE
```

```
dbClearResult(res)
dbDisconnect(res)
```

7 SQLite support

DatabaseConnector also supports SQLite through the RSQLite package, mainly for testing and demonstration purposes. Provide the path to the SQLite file as the `server` argument when connecting. If no file exists it will be created:

```
conn <- connect(dbms = "sqlite", server = tempfile())
```

```
#> Connecting using SQLite driver
```

```
# Upload cars dataset as table:
insertTable(conn, "cars", cars)

querySql(conn, "SELECT COUNT(*) FROM cars;")
```

```
#>   COUNT(*)
#> 1       50
```

```
disconnect(conn)
```