Querying a database

Martijn J. Schuemie

2023-03-15

Contents

1 Introduction

This vignette describes how to use the `DatabaseConnector` package to query a database. It assumes you already know how to create a connection as described in the 'Connecting to a database' vignette.

2 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 statements in a single SQL string.

Some examples:

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

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

```
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.

2.1 Querying using Andromeda objects

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

```
library(Andromeda)
x <- andromeda()
querySqlToAndromeda(connection = conn,
                    sql = "SELECT * FROM person",
                    andromeda = x,
                    andromedaTableName = "person")
```

Where `x` is now an `Andromeda` object with table `person`.

2.2 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()`, `renderTranslateQuerySqlToAndromeda()`. 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'.

Note that, on some platforms like Oracle, when using temp tables, it might be required to provide the `tempEmulationSchema` argument, since these platforms do not support tables the way other platforms do.

3 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.

4 Logging all queries

For several reasons it might be helpful to log all queries sent to the server (and the time to completion), for example to understand performance issues. For this one can use the `ParallelLogger` package. If the `LOG_DATABASECONNECTOR_SQL` option is set to `TRUE`, each query will be logged at the 'trace' level. For example:

```
options(LOG_DATABASECONNECTOR_SQL = TRUE)
ParallelLogger::addDefaultFileLogger("sqlLog.txt", name = "TEST_LOGGER")

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

readLines("sqlLog.txt")
```

```
[2] "2022-11-18 10:23:59\t[Main thread]\tTRACE\tDatabaseConnector\tlogTrace\tQuerying SQL: SELECT TOP 1
[3] "2022-11-18 10:23:59\t[Main thread]\tTRACE\tDatabaseConnector\tlogTrace\tQuerying SQL took 0.105010
```