Using SqlRender

Contents

1 Introduction

This vignette describes how one could use the SqlRender R package.

2 Installation instructions

SqlRender is maintained in a Github repository. It can be downloaded and installed from within R using the `devtools` package:

```
install.packages("devtools")
library("devtools")
install_github("ohdsi/SqlRender")
```

3 SQL parameterization

One of the main functions of the package is to support parameterization of SQL. Often, small variations of SQL need to be generated based on some parameters. SqlRender offers a simple markup syntax inside the SQL code to allow paramerization. Rendering the SQL based on parameter values is done using the `renderSql()` function.

3.1 Substituting parameter values

The `@` character can be used to indicate parameter names that need to be exchange for actual parameter values when rendering. In the following example, a variable called `a` is mentioned in the SQL. In the call to the renderSql function the value of this parameter is defined:

```
sql <- "SELECT * FROM table WHERE id = @a;"
renderSql(sql, a = 123)$sql
```

```
#> [1] "SELECT * FROM table WHERE id = 123;"
```

Note that, unlike the parameterization offered by most database management systems, it is just as easy to parameterize table or field names as values:

```
sql <- "SELECT * FROM @x WHERE id = @a;"
renderSql(sql, x = "my_table", a = 123)$sql
```

```
#> [1] "SELECT * FROM my_table WHERE id = 123;"
```

The parameter values can be numbers, strings, booleans, as well as vectors, which are converted to comma-delimited lists:

```
sql <- "SELECT * FROM table WHERE id IN (@a);"
renderSql(sql, a = c(1, 2, 3))$sql
```

```
#> [1] "SELECT * FROM table WHERE id IN (1,2,3);"
```

3.2 Default parameter values

For some or all parameters, it might make sense to define default values that will be used unless the user specifies another value. This can be done using the `{DEFAULT @parameter = value}` syntax:

```
sql <- "{DEFAULT @a = 1} SELECT * FROM table WHERE id = @a;"
renderSql(sql)$sql
```

```
#> [1] " SELECT * FROM table WHERE id = 1;"
```

```
renderSql(sql, a = 2)$sql
```

```
#> [1] " SELECT * FROM table WHERE id = 2;"
```

Defaults for multiple variables can be defined:

```
sql <- "{DEFAULT @a = 1} {DEFAULT @x = 'my_table'} SELECT * FROM @x WHERE id = @a;"
renderSql(sql)$sql
```

```
#> [1] "  SELECT * FROM my_table WHERE id = 1;"
```

3.3 If-then-else

Sometimes blocks of codes need to be turned on or off based on the values of one or more parameters. This is done using the `{Condition} ? {if true} : {if false}` syntax. If the *condition* evaluates to true or 1, the *if true* block is used, else the *if false* block is shown (if present).

```
sql <- "SELECT * FROM table {@x} ? {WHERE id = 1}"
renderSql(sql, x = FALSE)$sql
```

```
#> [1] "SELECT * FROM table "
```

```
renderSql(sql, x = TRUE)$sql
```

```
#> [1] "SELECT * FROM table WHERE id = 1"
```

Simple comparisons are also supported:

```
sql <- "SELECT * FROM table {@x == 1} ? {WHERE id = 1};"
renderSql(sql, x = 1)$sql
```

```
#> [1] "SELECT * FROM table WHERE id = 1;"
```

```
renderSql(sql, x = 2)$sql
```

```
#> [1] "SELECT * FROM table ;"
```

As well as the `IN` operator:

```
sql <- "SELECT * FROM table {@x IN (1,2,3)} ? {WHERE id = 1}; "
renderSql(sql, x = 2)$sql
```

```
#> [1] "SELECT * FROM table WHERE id = 1; "
```

4 Translation to other SQL dialects

SQL for one platform (e.g. Microsoft SQL Server) will not always execute on other platforms (e.g. Oracle). The `translateSql()` function can be used to translate between different dialects, but there are some limitations.

A first limitation is that **the starting dialect has to be SQL Server**. The reason for this is that this dialect is in general the most specific. For example, the number of days between two dates in SQL Server has to be computed using the DATEDIFF function: `DATEDIFF(dd,a,b)`. In other languages one can simply subtract the two dates: `b-a`. Since you'd need to know a and b are dates, it is not possible to go from other languages to SQL Server, only the other way around.

A second limitation is that currently only these dialects are supported as targets: **Oracle**, **PostgreSQL**, with some support for **Amazon Redhift**.

A third limitation is that only a limited set of translation rules have currently been implemented, although adding them to the list should not be hard.

A last limitation is that not all functions supported in one dialect have an equivalent in other dialects.

Below an example:

```
sql <- "SELECT DATEDIFF(dd,a,b) FROM table; "
translateSql(sql, targetDialect = "oracle")$sql
```

```
#> [1] "SELECT (b - a) FROM table; "
```

The `targetDialect` parameter can have the following values:

- "oracle"
- "postgresql"
- "redshift"
- "sql server" (no translation)

4.1 Functions and structures supported by translateSql

Giving an exhaustive list of all functions is not feasible. Many SQL functions actually are the same in all dialects and therefore already work fine, and the list of functions that are translated by translateSql is growing over time. See the latest version of the rule list to see which ones are covered. Some of the functions we've used succesfully are `AVG`, `CAST`, `COUNT`, `COUNT_BIG`, `DATEADD`, `DATEDIFF`, `EXP`, `FLOOR`, `GETDATE`, `ISNULL`, `LEFT`, `LEN`, `LOG`, `MAX`, `MIN`, `MONTH`, `RIGHT`, `ROUND`, `STDEV`, `SQRT`, `SUM` and `YEAR`.

Similarly, many SQL syntax structures are supported. Here is a non-exhaustive lists of things that we know will translate well:

```
SELECT * FROM table;                                          -- Simple selects

SELECT * FROM table_1 INNER JOIN table_2 ON a = b;            -- Selects with joins

SELECT * FROM (SELECT * FROM table_1) tmp WHERE a = b;        -- Nested queries

SELECT * INTO new_table FROM table;                           -- Selecting into a new table

CREATE TABLE table (field INT);                               -- Creating tables

INSERT INTO other_table (field_1) VALUES (1);                 -- Inserting verbatim values

INSERT INTO other_table (field_1) SELECT value FROM table;    -- Inserting from SELECT

DROP TABLE table;                                             -- Simple drop commands

IF OBJECT_ID('ACHILLES_analysis', 'U') IS NOT NULL            -- Drop table if it exists
  drop table ACHILLES_analysis;

WITH cte AS (SELECT * FROM table) SELECT * FROM cte;          -- Common table expressions

SELECT ROW_NUMBER() OVER (PARTITION BY a ORDER BY b)          -- OVER clauses
  AS "Row Number" FROM table;

SELECT CASE WHEN a=1 THEN a ELSE 0 END AS value FROM table; -- CASE WHEN clauses
```

4.2 String concatenation

String concatentation is the only part where SQL Server is less specific than other dialects. In SQL Server, one would write `SELECT first_name + ' ' + last_name AS full_name FROM table`, but this should be

`SELECT first_name || ' ' || last_name AS full_name FROM table` in PostgreSQL and Oracle. SqlRender tries to guess when values that are being concatenated are strings. In the example above, because we have an explicit string (the space surrounded by single quotation marks), the translation will be correct. However, if the query had been `SELECT first_name + last_name AS full_name FROM table`, SqlRender would have had no clue the two fields were strings, and would incorrectly leave the plus sign. Another clue that a value is a string is an explicit cast to VARCHAR, so `SELECT last_name + CAST(age AS VARCHAR(3)) AS full_name FROM table` would also be translated correctly.

4.3 Temp tables

Temp tables can be very useful to store intermediate results, and when used correctly can be used to dramatically improve performance of queries. In Postgres and SQL Server temp tables have very nice properties: they're only visible to the current user, are automatically dropped when the session ends, and can be created even when the user has no write access. Unfortunately, in Oracle temp tables are basically permanent tables, with the only difference that the data inside the table is only visible to the current user. We therefore recommend using temp tables as little as possible. If you do use them, make sure to

- Always set the current schema to somewhere where the user has write access, and adding explicit schema references to all other tables you need (e.g. @cdm_schema.dbo.person).
- Add truncate table and drop table statements for temp tables at the beginning and end of the SQL code.
- Generate random temp table names so they cannot conflict with the temp tables of another user.

Instead, it is probably best to use Common Table Expressions (CTEs) when one would normally use a temp table. For example, instead of

```
SELECT * INTO #children FROM person WHERE year_of_birth > 2000;
SELECT * FROM #children WHERE gender = 8507;
```

it is preferred to use

```
WITH children AS (SELECT * FROM person WHERE year_of_birth > 2000)
SELECT * FROM children WHERE gender = 8507;
```

4.4 Schemas and databases

In SQL Server, tables are located in a schema, and schemas reside in a database. For example, `cdm_data.dbo.person` refers to the `person` table in the `dbo` schema in the `cdm_data` database. In other dialects, even though a similar hierarchy often exists they are used very differently. In SQL Server, there is typically one schema per database (often called `dbo`), and users can easily use data in different databases. On other platforms, for example in PostgreSQL, it is not possible to use data across databases in a single session, but there are often many schemas in a database. In PostgreSQL one could say that the equivalent of SQL Server's database is the schema.

We therefore recommend concatenating SQL Server's database and schema into a single parameter, which we typically call `@databaseSchema`. For example, we could have the parameterized SQL

```
SELECT * FROM @databaseSchema.person
```

where on SQL Server we can include both database and schema names in the value: `databaseSchema = "cdm_data.dbo"`. On otherplatforms, we can use the same code, but now only specify the schema as the parameter value: `databaseSchema = "cdm_data"`.

The one situation where this will fail is the `USE` command, since `USE cdm_data.dbo;` will throw an error. It is therefore preferred not to use the `USE` command, but always specify the database / schema where a table is located. However, if one wanted to use it anyway, we recommend creating two variables, one called `@database` and the other called `@databaseSchema`. For example, for this parameterized SQL:

```
SELECT * FROM @databaseSchema.person;
USE @database;
SELECT * FROM person
```

we can set `database = "cdm_data"` and the other called `databaseSchema = "cdm_data.dbo"`. On platforms other than SQL Server, the two variables will hold the same value and only on SQL Server will they be different. Within an R function, it is even possible to derive one variable from the other, so the user of your function would need to specify only one value:

```
foo <- function(databaseSchema, dbms) {
    database <- strsplit(databaseSchema, "\\.")[[1]][1]
    sql <- "SELECT * FROM @databaseSchema.person; USE @database; SELECT * FROM person;"
    sql <- renderSql(sql, databaseSchema = databaseSchema, database = database)$sql
    sql <- translateSql(sql, targetDialect = dbms)$sql
    return(sql)
}
foo("cdm_data.dbo", "sql server")
```

```
#> [1] "SELECT * FROM cdm_data.dbo.person; USE cdm_data; SELECT * FROM person;"
```

```
foo("cdm_data", "postgresql")
```

```
#> [1] "SELECT * FROM cdm_data.person; SET search_path TO  cdm_data; SELECT * FROM person;"
```

5 Debugging parameterized SQL

Debugging parameterized SQL can be a bit complicated; Only the rendered SQL can be tested against a database server, but changes to the code should be made in the parameterized (pre-rendered) SQL. Two functions have been developed to aid the debugging process: `renderSqlFile()` and `translateSqlFile()`. These can be used to read SQL from file, render or translate it, and write it back to file. For example:

```
translateSqlFile("parameterizedSql.txt", "renderedSql.txt")
```

will render the file, using the default parameter values specified in the SQL. What works well for us is editing in the parameterized file, (re)running the command above, and have the rendered SQL file open in a SQL client for testing. Any problems reported by the server can be dealt with in the source SQL, and can quickly be re-rendered.

6 Developing R packages that contain parameterized SQL

Often, the SQL code will become part of an R package, where it might be used to perform initial data-preprocesing and extraction before further analysis. We've developed the following practice for doing so: The parameterized SQL should be located in the *inst/sql/* folder of the package. The parameterized SQL for SQL Server should be in the *inst/sql/sql_server/* folder. If for some reason you do not want to use the translation

functions to generate the SQL for some dialect (e.g because dialect specific code might be written that gives better performance), a dialect-specific version of the parameterized SQL should be placed in a folder with the name of that dialect, for example *inst/sql/oracle/*. SqlRender has a function `loadRenderTranslateSql()` that will first check if a dialect-specific version is available for the target dialect. If it is, that version will be rendered, else the SQL Server version will be rendered and subsequently translated to the target dialect.

The `createRWrapperForSql()` function can be used to create an R wrapper around a rendered SQL file, using the `loadRenderTranslateSql()` function . For example, suppose we have a text file called *test.sql* containing the following parameterized SQL:

```
{DEFAULT @selected_value = 1}
SELECT * FROM table INTO result where x = @selected_value;
```

Then the command

```
createRWrapperForSql(sqlFilename = "test.sql",
                     rFilename = "test.R",
                     packageName = "myPackage")
```

would result in the file *test.R* being generated containing this R code:

```
#' Todo: add title
#'
#' @description
#' Todo: add description
#'
#' @details
#' Todo: add details
#'
#' @param connectionDetails   An R object of type \code{ConnectionDetails} created ...
#' @param selectedValue
#'
#' @export
test <- function(connectionDetails, selectedValue = 1) {
    renderedSql <- loadRenderTranslateSql("test.txt", packageName = "myPackage",
        dbms = connectionDetails$dbms, selected_value = selectedValue)
    conn <- connect(connectionDetails)

    writeLines("Executing multiple queries. This could take a while")
    executeSql(conn, renderedSql)
    writeLines("Done")

    dummy <- dbDisconnect(conn)
}
```

This code expects the file *test.sql* to be located in the *inst/sql/sql_server/* folder of the package source.

Note that the parameters are identified by the declaration of default values, and that snake_case names (our standard for SQL) are converted to camelCase names (our standard for R).