

# Package ‘SqlRender’

June 11, 2015

**Type** Package

**Title** Rendering parameterized SQL and translation to dialects

**Version** 1.1.2

**Date** 2015-03-10

**Author** Martijn J. Schuemie and Marc A. Suchard

**Maintainer** Martijn Schuemie <schuemie@ohdsi.org>

**Description** This is an R package for rendering parameterized SQL, and translating it to different SQL dialects.

**License** Apache License

**VignetteBuilder** knitr

**Imports** rJava

**Suggests** testthat,  
knitr,  
rmarkdown,  
OhdsiRTools

**LazyData** false

## R topics documented:

camelCaseToSnakeCase . . . . .	2
createRWrapperForSql . . . . .	2
loadRenderTranslateSql . . . . .	3
readSql . . . . .	4
renderSql . . . . .	4
renderSqlFile . . . . .	5
snakeCaseToCamelCase . . . . .	6
splitSql . . . . .	7
translateSql . . . . .	7
translateSqlFile . . . . .	8
writeSql . . . . .	9
<b>Index</b>	<b>10</b>

---

`camelCaseToSnakeCase`    *Convert a camel case string to snake case*

---

**Description**

Convert a camel case string to snake case

**Usage**

```
camelCaseToSnakeCase(string)
```

**Arguments**

`string`                      The string to be converted

**Value**

A string

**Examples**

```
camelCaseToSnakeCase("cdmDatabaseSchema")  
# > 'cdm_database_schema'
```

---

`createRWrapperForSql`    *Create an R wrapper for SQL*

---

**Description**

`createRWrapperForSql` creates an R wrapper for a parameterized SQL file. The created R script file will contain a single function, that executes the SQL, and accepts the same parameters as specified in the SQL.

**Usage**

```
createRWrapperForSql(sqlFilename, rFilename, packageName,  
  createRoxygenTemplate = TRUE)
```

**Arguments**

`sqlFilename`            The SQL file.

`rFilename`              The name of the R file to be generated. Defaults to the name of the SQL file with the extension reset to R.

`packageName`          The name of the package that will contains the SQL file.

`createRoxygenTemplate`  
                         If true, a template of Roxygen comments will be added.

## Details

This function reads the declarations of defaults in the parameterized SQL file, and creates an R function that exposes the parameters. It uses the `loadRenderTranslateSql` function, and assumes the SQL will be used inside a package. To use inside a package, the SQL file should be placed in the `inst/sql/sql_server` folder of the package.

## Examples

```
## Not run:
# This will create a file called CohortMethod.R:
createRWrapperForSql("CohortMethod.sql", packageName = "CohortMethod")

## End(Not run)
```

---

```
loadRenderTranslateSql
```

*Load, render, and translate a SQL file in a package*

---

## Description

`loadRenderTranslateSql` Loads a SQL file contained in a package, renders it and translates it to the specified dialect

## Usage

```
loadRenderTranslateSql(sqlFilename, packageName, dbms = "sql server", ...,
  oracleTempSchema = NULL)
```

## Arguments

<code>sqlFilename</code>	The source SQL file
<code>packageName</code>	The name of the package that contains the SQL file
<code>dbms</code>	The target dialect. Currently 'sql server', 'oracle', 'postgres', and 'redshift' are supported
<code>...</code>	Parameter values used for <code>renderSql</code>
<code>oracleTempSchema</code>	A schema that can be used to create temp tables in when using Oracle.

## Details

This function looks for a SQL file with the specified name in the `inst/sql/<dbms>` folder of the specified package. If it doesn't find it in that folder, it will try and load the file from the `inst/sql/sql_server` folder and use the `translateSql` function to translate it to the requested dialect. It will subsequently call the `renderSql` function with any of the additional specified parameters.

## Value

Returns a string containing the rendered SQL.

### Examples

```
## Not run:
renderedSql <- loadRenderTranslateSql("CohortMethod.sql",
                                     packageName = "CohortMethod",
                                     dbms = connectionDetails$dbms,
                                     CDM_schema = "cdmSchema")

## End(Not run)
```

---

readSql	<i>Reads a SQL file</i>
---------	-------------------------

---

### Description

readSql loads SQL from a file

### Usage

```
readSql(sourceFile)
```

### Arguments

sourceFile      The source SQL file

### Details

readSql loads SQL from a file

### Value

Returns a string containing the SQL.

### Examples

```
## Not run:
readSql("myParamStatement.sql")

## End(Not run)
```

---

renderSql	<i>renderSql</i>
-----------	------------------

---

### Description

renderSql Renders SQL code based on parameterized SQL and parameter values.

### Usage

```
renderSql(sql = "", ...)
```

**Arguments**

<code>sql</code>	The parameterized SQL
<code>...</code>	Parameter values

**Details**

This function takes parameterized SQL and a list of parameter values and renders the SQL that can be send to the server. Parameterization syntax:

**@parameterName** Parameters are indicated using a @ prefix, and are replaced with the actual values provided in the renderSql call.

**{DEFAULT @parameterName = parameterValue}** Default values for parameters can be defined using curly and the DEFAULT keyword.

**{if}?{then}:{else}** The if-then-else pattern is used to turn on or off blocks of SQL code.

**Value**

A list containing the following elements:

**parameterizedSql** The original parameterized SQL code

**sql** The rendered sql

**Examples**

```
renderSql("SELECT * FROM @a;", a = "myTable")
renderSql("SELECT * FROM @a {@b}?{WHERE x = 1};", a = "myTable", b = "true")
renderSql("SELECT * FROM @a {@b == ''}?{WHERE x = 1}:{ORDER BY x};", a = "myTable", b = "true")
renderSql("SELECT * FROM @a {@b != ''}?{WHERE @b = 1};", a = "myTable", b = "y")
renderSql("SELECT * FROM @a {1 IN (@c)}?{WHERE @b = 1};",
  a = "myTable",
  b = "y",
  c = c(1, 2, 3, 4))
renderSql("{DEFAULT @b = \"someField\"}SELECT * FROM @a {@b != ''}?{WHERE @b = 1};",
  a = "myTable")
renderSql("SELECT * FROM @a {@a == 'myTable' & @b != 'x'}?{WHERE @b = 1};",
  a = "myTable",
  b = "y")
```

---

renderSqlFile

*Render a SQL file*


---

**Description**

renderSqlFile Renders SQL code in a file based on parameterized SQL and parameter values, and writes it to another file.

**Usage**

```
renderSqlFile(sourceFile, targetFile, ...)
```

**Arguments**

sourceFile	The source SQL file
targetFile	The target SQL file
...	Parameter values

**Details**

This function takes parameterized SQL and a list of parameter values and renders the SQL that can be send to the server. Parameterization syntax:

**@parameterName** Parameters are indicated using a @ prefix, and are replaced with the actual values provided in the renderSql call.

**{DEFAULT @parameterName = parameterValue}** Default values for parameters can be defined using curly and the DEFAULT keyword.

**{if}?{then}:{else}** The if-then-else pattern is used to turn on or off blocks of SQL code.

**Examples**

```
## Not run:
renderSqlFile("myParamStatement.sql", "myRenderedStatement.sql", a = "myTable")

## End(Not run)
```

---

snakeCaseToCamelCase    *Convert a snake case string to camel case*

---

**Description**

Convert a snake case string to camel case

**Usage**

```
snakeCaseToCamelCase(string)
```

**Arguments**

string	The string to be converted
--------	----------------------------

**Value**

A string

**Examples**

```
snakeCaseToCamelCase("cdm_database_schema")
# > 'cdmDatabaseSchema'
```

---

splitSql	<i>splitSql</i>
----------	-----------------

---

**Description**

splitSql splits a string containing multiple SQL statements into a vector of SQL statements

**Usage**

```
splitSql(sql)
```

**Arguments**

sql	The SQL string to split into separate statements
-----	--

**Details**

This function is needed because some DBMSs (like ORACLE) do not accept multiple SQL statements being sent as one execution.

**Value**

A vector of strings, one for each SQL statement

**Examples**

```
splitSql("SELECT * INTO a FROM b; USE x; DROP TABLE c;")
```

---

translateSql	<i>translateSql</i>
--------------	---------------------

---

**Description**

translateSql translates SQL from one dialect to another

**Usage**

```
translateSql(sql = "", sourceDialect = "sql server",  
  targetDialect = "oracle", oracleTempSchema = NULL)
```

**Arguments**

sql	The SQL to be translated
sourceDialect	The source dialect. Currently, only "sql server" for Microsoft SQL Server is supported
targetDialect	The target dialect. Currently "oracle", "postgresql", "pdw", and "redshift" are supported
oracleTempSchema	A schema that can be used to create temp tables in when using Oracle.

### Details

This function takes SQL in one dialect and translates it into another. It uses simple pattern replacement, so its functionality is limited.

### Value

A list containing the following elements:

**originalSql** The original parameterized SQL code

**sql** The translated SQL

### Examples

```
## Not run:
translateSql("USE my_schema", "sql server", "oracle")

## End(Not run)
```

---

translateSqlFile	<i>Translate a SQL file</i>
------------------	-----------------------------

---

### Description

This function takes SQL and translates it to a different dialect.

### Usage

```
translateSqlFile(sourceFile, targetFile, sourceDialect = "sql server",
  targetDialect = "oracle", oracleTempSchema = NULL)
```

### Arguments

sourceFile	The source SQL file
targetFile	The target SQL file
sourceDialect	The source dialect. Currently, only 'sql server' for Microsoft SQL Server is supported
targetDialect	The target dialect. Currently 'oracle', 'postgresql', and 'redshift' are supported
oracleTempSchema	A schema that can be used to create temp tables in when using Oracle.

### Details

This function takes SQL and translates it to a different dialect.

### Examples

```
## Not run:
translateSqlFile("myRenderedStatement.sql",
  "myTranslatedStatement.sql",
  targetDialect = "postgresql")

## End(Not run)
```



---

writeSql	<i>Write SQL to a SQL (text) file</i>
----------	---------------------------------------

---

**Description**

writeSql writes SQL to a file

**Usage**

```
writeSql(sql, targetFile)
```

**Arguments**

sql	A string containing the sql
targetFile	The target SQL file

**Details**

writeSql writes SQL to a file

**Examples**

```
## Not run:  
sql <- "SELECT * FROM @table_name"  
writeSql(sql, "myParamStatement.sql")  
  
## End(Not run)
```

# Index

camelCaseToSnakeCase, [2](#)  
createRWrapperForSql, [2](#)  
  
loadRenderTranslateSql, [3](#)  
  
readSql, [4](#)  
renderSql, [4](#)  
renderSqlFile, [5](#)  
  
snakeCaseToCamelCase, [6](#)  
splitSql, [7](#)  
  
translateSql, [7](#)  
translateSqlFile, [8](#)  
  
writeSql, [9](#)