Package 'SqlRender'

October 27, 2020

Type Package

Title Rendering Parameterized SQL and Translation to Dialects

Version 1.7.0

Date 2020-10-27

Maintainer Martijn Schuemie <schuemie@ohdsi.org>

Description A rendering tool for parameterized SQL that also translates into
different SQL dialects. These dialects include 'Microsoft Sql Server', 'Oracle',
'PostgreSql', 'Amazon RedShift', 'Apache Impala', 'IBM Netezza', 'Google BigQuery', 'Microsoft PDW', and 'SQLite'.

License Apache License 2.0

VignetteBuilder knitr

URL https://ohdsi.github.io/SqlRender/, https://github.com/OHDSI/SqlRender

BugReports https://github.com/OHDSI/SqlRender/issues

Imports rJava,
rlang

Suggests testthat,
knitr,
rmarkdown,
shiny,
shinydashboard

LazyData false

RoxygenNote 7.1.1

Encoding UTF-8

R topics documented:

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("exposureConceptId1")
# > 'exposure_concept_id_1'
```

camelCaseToTitleCase   *Convert a camel case string to title case*

Description

Convert a camel case string to title case

Usage

```
camelCaseToTitleCase(string)
```

Arguments

string   The string to be converted

Value

A string

Examples

```
camelCaseToTitleCase("exposureConceptId1")
# > 'Exposure Concept Id 1'
```

---

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)
```

---

launchSqlRenderDeveloper

Launch the SqlRender Developer Shiny app

Description

Launch the SqlRender Developer Shiny app

Usage

```
launchSqlRenderDeveloper(launch.browser = TRUE)
```

Arguments

launch.browser   Should the app be launched in your default browser, or in a Shiny window. Note: copying to clipboard will not work in a Shiny window.

Details

Launches a Shiny app that allows the user to develop SQL and see how it translates to the supported dialects.

---

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",
  ...,
  tempEmulationSchema = getOption("sqlRenderTempEmulationSchema"),
  oracleTempSchema = NULL,
  warnOnMissingParameters = TRUE
)
```

Arguments

`sqlFilename`	The source SQL file
`packageName`	The name of the package that contains the SQL file
`dbms`	The target dialect. Currently 'sql server', 'oracle', 'postgres', and 'redshift' are supported
`...`	Parameter values used for `render`

`tempEmulationSchema`

Some database platforms like Oracle and Impala do not truly support temp tables. To emulate temp tables, provide a schema with write privileges where temp tables can be created.

`oracleTempSchema`

DEPRECATED: use `tempEmulationSchema` instead.

`warnOnMissingParameters`

Should a warning be raised when parameters provided to this function do not appear in the parameterized SQL that is being rendered? By default, this is TRUE.

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 `translate` function to translate it to the requested dialect. It will subsequently call the `render` 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	*Reads a SQL file*

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)
```

---

render   *Render SQL code based on parameterized SQL and parameter values*

Description

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

Usage

```
render(sql = "", warnOnMissingParameters = TRUE, ...)
```

Arguments

sql   The parameterized SQL

warnOnMissingParameters
                 Should a warning be raised when parameters provided to this function do not
                 appear in the parameterized SQL that is being rendered? By default, this is
                 TRUE.

...   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 render 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 character string containing the rendered SQL.

Examples

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

---

renderSql	*Deprecated: Render SQL code based on parameterized SQL and parameter values*

Description

This function has been deprecated. Use [render](#) instead. This new function returns a character vector instead of a list.

Usage

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

Arguments

sql   The parameterized SQL

warnOnMissingParameters

                  Should a warning be raised when parameters provided to this function do not appear in the parameterized SQL that is being rendered? By default, this is TRUE.

...   Parameter values

Value

A list containing the following elements:

parameterizedSql The original parameterized SQL code

sql The rendered sql

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, warnOnMissingParameters = TRUE, ...)
```

Arguments

sourceFile   The source SQL file

targetFile   The target SQL file

warnOnMissingParameters
                    Should a warning be raised when parameters provided to this function do not appear in the parameterized SQL that is being rendered? By default, this is TRUE.

...   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 render 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("exposure_concept_id_1")
# > 'exposureConceptId1'
```

---

splitSql *Split a single SQL string into one or more SQL statements*

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 accepts 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;")
```

---

translate   *Translates SQL from one dialect to another*

Description

`translate` translates SQL from one dialect to another.

Usage

```
translate(
  sql = "",
  targetDialect,
  tempEmulationSchema = getOption("sqlRenderTempEmulationSchema"),
  oracleTempSchema = NULL
)
```

Arguments

sql   The SQL to be translated

targetDialect   The target dialect. Currently "oracle", "postgresql", "pdw", "impala", "sqlite",
                       "netezza", "bigquery", and "redshift" are supported.

tempEmulationSchema

                    Some database platforms like Oracle and Impala do not truly support temp tables. To emulate temp tables, provide a schema with write privileges where temp tables can be created.

oracleTempSchema

                    DEPRECATED: use `tempEmulationSchema` instead.

Details

This function takes SQL in one dialect and translates it into another. It uses simple pattern replacement, so its functionality is limited. Note that trailing semicolons are not removed for Oracle, which is required before sending a statement through JDBC. This will be done by `splitSql`.

Value

A character string containing the translated SQL.

Examples

```
translate("USE my_schema;", targetDialect = "oracle")
```

translateSingleStatement

Translates a single SQL statement from one dialect to another

Description

translateSingleStatement translates a single SQL statement from one dialect to another.

Usage

```
translateSingleStatement(
  sql = "",
  targetDialect,
  tempEmulationSchema = getOption("sqlRenderTempEmulationSchema"),
  oracleTempSchema = NULL
)
```

Arguments

sql   The SQL to be translated

targetDialect   The target dialect. Currently "oracle", "postgresql", "pdw", "impala", "sqlite",
                 "netezza", "bigquery", and "redshift" are supported.

tempEmulationSchema
                 Some database platforms like Oracle and Impala do not truly support temp ta-
                 bles. To emulate temp tables, provide a schema with write privileges where
                 temp tables can be created.

oracleTempSchema
                 DEPRECATED: use tempEmulationSchema instead.

Details

This function takes SQL in one dialect and translates it into another. It uses simple pattern replace-
ment, so its functionality is limited. This removes any trailing semicolon as required by Oracle
when sending through JDBC. An error is thrown if more than one statement is encountered in the
SQL.

Value

A character vector with the translated SQL.

Examples

```
translateSingleStatement("USE my_schema;", targetDialect = "oracle")
```

---

translateSql   *Deprecated: Translates SQL from one dialect to another*

Description

This function has been deprecated. Use [translate](#) instead. This new function returns a character vector instead of a list.

Usage

```
translateSql(sql = "", targetDialect, oracleTempSchema = NULL)
```

Arguments

sql   The SQL to be translated

targetDialect   The target dialect. Currently "oracle", "postgresql", "pdw", "impala", "netezza", "bigquery", and "redshift" are supported

oracleTempSchema

                    A schema that can be used to create temp tables in when using Oracle or Impala.

Value

A list containing the following elements:

originalSql The original parameterized SQL code

sql The translated SQL

---

translateSqlFile   *Translate a SQL file*

Description

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

Usage

```
translateSqlFile(
  sourceFile,
  targetFile,
  targetDialect,
  tempEmulationSchema = getOption("sqlRenderTempEmulationSchema"),
  oracleTempSchema = NULL
)
```

Arguments

sourceFile	The source SQL file
targetFile	The target SQL file
targetDialect	The target dialect. Currently "oracle", "postgresql", "pdw", "impala", "sqlite", "netezza", "bigquery", and "redshift" are supported.
tempEmulationSchema	
	Some database platforms like Oracle and Impala do not truly support temp tables. To emulate temp tables, provide a schema with write privileges where temp tables can be created.
oracleTempSchema	
	DEPRECATED: use `tempEmulationSchema` instead.

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	*Write SQL to a SQL (text) file*

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