Using SqlRender

Martijn J. Schuemie

2026-11-27

Contents

Introduction **1**

SQL parameterization **1**
Substituting parameter values . . . 1
Default parameter values . . . 2
If-then-else . . . 2

Translation to other SQL dialects **3**
Functions and structures supported by translate . . . 4
String concatenation . . . 5
Bitwise operators . . . 5
Table aliases and the AS keyword . . . 5
Temp tables . . . 6
Implicit casts . . . 7
Case sensitivity in string comparisons . . . 7
Schemas and databases . . . 7
Optimization for massively parallel processing . . . 8

Debugging parameterized SQL **9**

Developing R packages that contain parameterized SQL **9**

Spark SQL **10**

Introduction

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

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 parameterization. Rendering the SQL based on parameter values is done using the `render()` function.

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 render function the value of this parameter is defined:

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

```
## [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;"
render(sql, x = "my_table", a = 123)
```

```
## [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);"
render(sql, a = c(1, 2, 3))
```

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

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;"
render(sql)
```

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

```
render(sql, a = 2)
```

```
## [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;"
render(sql)
```

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

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}"
render(sql, x = FALSE)
```

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

```
render(sql, x = TRUE)
```

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

Simple comparisons are also supported:

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

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

```
render(sql, x = 2)
```

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

As well as the `IN` operator:

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

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

Clauses can combined with boolean operators:

```
sql <- "SELECT * FROM table {@x IN (1,2,3) | @y != 3} ? {WHERE id = @x AND value = @y};"
render(sql, x = 4, y = 4)
```

```
## [1] "SELECT * FROM table WHERE id = 4 AND value = 4;"
```

```
sql <- "SELECT * FROM table {(@x == 1 | @x == 3) & @y != 3} ? {WHERE id = @x AND val = @y};"
render(sql, x = 3, y = 4)
```

```
## [1] "SELECT * FROM table WHERE id = 3 AND val = 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 `translate()` 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**, **Microsoft PDW (Parallel Data Warehouse)**, **Impala**, **Netezza**, **Google BigQuery**, **Amazon Redshift**, **Snowflake**, **Azure Synapse**, **Apache Spark**, **SQLite**, and **InterSystems IRIS**.

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; "
translate(sql, targetDialect = "oracle")
```

```
## [1] "SELECT CEIL(CAST(b AS DATE) - CAST(a AS DATE)) FROM table ; "
## attr(,"sqlDialect")
## [1] "oracle"
```

The `targetDialect` parameter can have the following values:

- "oracle"
- "postgresql"
- "pdw"
- "redshift"
- "impala"

- “netezza”
- “bigquery”
- “snowflake”
- “synapse”
- “spark”
- “sqlite”
- “sqlite extended”
- “sql server”
- “iris”

Functions and structures supported by translate

These SQL Server functions have been tested and were found to be translated correctly to the various dialects:

```
## Warning in matrix(funs, ncol = 4): data length [53] is not a sub-multiple or
## multiple of the number of rows [14]
```

Table 1: Functions supported by translate

Function	Function	Function	Function
ABS	DATEFROMPARTS	LOG10	RTRIM
ACOS	DATETIMEFROMPARTS	LOWER	SIN
ASIN	DAY	LTRIM	SQRT
ATAN	EOMONTH	MAX	SQUARE
AVG	EXP	MIN	STDEV
CAST	FLOOR	MONTH	SUM
CEILING	GETDATE	NEWID	TAN
CHARINDEX	HASHBYTES*	PI	UPPER
CONCAT	IIF	POWER	VAR
COS	ISNULL	RAND	YEAR
COUNT	ISNUMERIC	RANK	
COUNT_BIG	LEFT	RIGHT	ABS
DATEADD	LEN	ROUND	ACOS
DATEDIFF	LOG	ROW_NUMBER	ASIN

- Requires special privileges on Oracle. Has no equivalent on SQLite.

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 TOP 10 * FROM table;                             -- Limiting to top rows

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 my_table;                                          -- Simple drop commands

DROP TABLE IF EXISTS ACHILLES_analysis;                       -- Drop table if it exists

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

SELECT * FROM a UNION SELECT * FROM b                         -- UNIONs

SELECT * FROM a INTERSECT SELECT * FROM b                     -- INTERSECTIONs

SELECT * FROM a EXCEPT SELECT * FROM b                        -- EXCEPT

CAST('20220101' AS DATE)                                      -- Cast to date
```

String concatenation

String concatenation is one area 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. To avoid ambiguity altogether, it is probable best to use the `CONCAT()` function to concatenate two or more strings.

Bitwise operators

Some DMBS, such as Oracle, only support one bitwise operator: AND (through the `BITWISE()` function). Luckily this is enough to implement all others operators, such as OR and XOR. It does mean you are limited to using the `&` operator in OHDSI SQL.

A further limitation is that the `&` operator's scope must be defined by parentheses So for example

```
SELECT (a & b);
```

can be translated to

```
SELECT BITAND(a, b) FROM DUAL;
```

in Oracle SQL.

Table aliases and the AS keyword

Many SQL dialects allow the use of the `AS` keyword when defining a table alias, but will also work fine without the keyword. For example, both these SQL statements are fine for SQL Server, PostgreSQL, RedShift, etc.:

```
-- Using AS keyword
SELECT *
FROM my_table AS table_1
```

```
INNER JOIN (
  SELECT * FROM other_table
) AS table_2
ON table_1.person_id = table_2.person_id;

-- Not using AS keyword
SELECT *
FROM my_table table_1
INNER JOIN (
  SELECT * FROM other_table
) table_2
ON table_1.person_id = table_2.person_id;
```

However, Oracle will throw an error when the `AS` keyword is used. In the above example, the first query will fail. It is therefore recommended to not use the `AS` keyword when aliasing tables. (Note: we can't make SqlRender handle this, because it can't easily distinguish between table aliases where Oracle doesn't allow `AS` to be used, and field aliases, where Oracle requires `AS` to be used.)

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 platforms like Postgres, PDW, RedShift 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. Other platforms, like Impala and Spark, don't support temp tables at all. This is why for some platforms SqlRender will try to emulate temp tables by

1. Adding a random string to the table name so tables from different users will not conflict.
2. Allowing the user to specify the schema where the temp tables will be created.

For example:

```
sql <- "SELECT * FROM #children;"
translate(sql, targetDialect = "oracle", tempEmulationSchema = "temp_schema")
```

```
## [1] "SELECT * FROM temp_schema.rzd6qwypchildren ;"
## attr(,"sqlDialect")
## [1] "oracle"
```

Note that the user will need to have write privileges on `temp_schema`. You can also set the `tempEmulationSchema` globally by using

```
options(sqlRenderTempEmulationSchema = "temp_schema")
```

Also note that because Oracle has a limit on table names of 30 characters, **temp table names are only allowed to be at most 22 characters long** because else the name will become too long after appending the session ID.

Furthermore, remember that emulated temp tables are not automatically dropped, so you will need to explicitly `TRUNCATE` and `DROP` all temp tables once you're done with them to prevent orphan tables accumulating in the Oracle temp schema.

If possible, try to avoid using temp tables altogether. Sometimes one could 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;
```

you could use

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

Implicit casts

One of the few points where SQL Server is less explicit than other dialects is that it allows implicit casts. For example, this code will work on SQL Server:

```
CREATE TABLE #temp (txt VARCHAR);

INSERT INTO #temp
SELECT '1';

SELECT * FROM #temp WHERE txt = 1;
```

Even though `txt` is a VARCHAR field and we are comparing it with an integer, SQL Server will automatically cast one of the two to the correct type to allow the comparison. In contrast, other dialects such as PosgreSQL will throw an error when trying to compare a VARCHAR with an INT.

You should therefore always make casts explicit. In the above example, the last statement should be replaced with either

```
SELECT * FROM #temp WHERE txt = CAST(1 AS VARCHAR);
```

or

```
SELECT * FROM #temp WHERE CAST(txt AS INT) = 1;
```

Case sensitivity in string comparisons

Some DBMS platforms such as SQL Server always perform string comparisons in a case-insensitive way, while others such as PostgreSQL are always case sensitive. It is therefore recommended to always assume case-sensitive comparisons, and to explicitly make comparisons case-insensitive when unsure about the case. For example, instead of

```
SELECT * FROM concept WHERE concep_class_id = 'Clinical Finding'
```

it is preferred to use

```
SELECT * FROM concept WHERE LOWER(concep_class_id) = 'clinical finding'
```

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 other platforms, 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 <- render(sql, databaseSchema = databaseSchema, database = database)
  sql <- translate(sql, targetDialect = dbms)
  return(sql)
}
foo("cdm_data.dbo", "sql server")
```

```
## [1] "SELECT * FROM cdm_data.dbo.person; USE cdm_data; SELECT * FROM person;"
## attr(,"sqlDialect")
## [1] "sql server"
```

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

```
## [1] "SELECT * FROM cdm_data.person; SET search_path TO cdm_data; SELECT * FROM person;"
## attr(,"sqlDialect")
## [1] "postgresql"
```

Optimization for massively parallel processing

Both PDW and RedShift are massively parallel processing platforms, meaning they consist of many nodes that work together. In such an environment, significant increases in performance can be achieved by fine-tuning the SQL for these platforms. Probably most importantly, developers can specify the way data is distributed over the nodes. Ideally, data in a node only needs to be combined with data in the same node. For example, if I have two tables with the field `person_id`, I would like all records with the same person ID to be on the same node, so a join on `person_id` can be performed locally without exchanging data between nodes.

SQL Server SQL, our source dialect, does not allow for these optimizations, so we've introduced the notion of hints. In the following example, a hint is provided on which field should be used for the distribution of data across nodes:

```
--HINT DISTRIBUTE_ON_KEY(person_id)
SELECT * INTO one_table FROM other_table;
```

which will translate into the following on PDW:

```
--HINT DISTRIBUTE_ON_KEY(person_id)
IF XACT_STATE() = 1 COMMIT;
CREATE TABLE one_table WITH (DISTRIBUTION = HASH(person_id)) AS
SELECT * FROM other_table;
```

Another tuning parameter is the key to sort a table on. This can be also be specified in a hint:

```
--HINT SORT_ON_KEY(INTERLEAVED:start_date)
CREATE TABLE cdm.my_table (row_id INT, start_date);
```

translates to the following on RedShift:

```
--HINT SORT_ON_KEY(INTERLEAVED:start_date)
CREATE TABLE cdm.my_table (row_id INT, start_date)
INTERLEAVED SORTKEY(start_date);
```

The hints should be formatted exactly as shown above, and directly precede the statement where the table is created.

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.

A Shiny app is included in the SqlRender package for interactively editing source SQL and generating rendered and translated SQL. The app can be started using:

```
launchSqlRenderDeveloper()
```

Which will open the default browser with the app.

In addition, 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:

```
renderSqlFile("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.

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-preprocessing 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).

Spark SQL

Spark SQL v3.1 improves the ANSI SQL compliance of Spark, however, at this time, INSERT statements are a bit limited, compared to other dialects:

- INSERT commands must feature a full column list in both the parenthetical column list and in the actual data to be inserted. A full column list here means that every column in the destination table must be present.
- The column list does not need to be in the correct order as the destination table, provided the parenthetical column list explicitly states the column names you're targeting.

To mitigate this issue, SqlRender has a Java function with R wrapper named `sparkHandleInsert()`, which will connect to the database and interrogate it for the full column list of a destination table used in an INSERT command, to then reconstitute the command to have the full column list.

For example:

Suppose we were inserting a record to the COHORT table with this call:

```
sql <- "insert into results_schema.cohort (subject_id, cohort_definition_id) values (1,
2)'
```

In Spark SQL, this would fail, as the COHORT table consists of cohort_definition_id, subject_id, cohort_start_date, and cohort_end_date.

Instead, we can run this:

```
sql <- "insert into results_schema.cohort (subject_id, cohort_definition_id) values (1, 2)'
connection <- DatabaseConnector::connect(connectionDetails = connectionDetails)
SqlRender::sparkHandleInsert(sql, connection)
```

and obtain: `INSERT INTO results_schema.cohort VALUES (2, 1, NULL, NULL);`, which places the columns in correct table order and pads it out with NULL values.