Using Query Namespaces

James P. Gilbert

2023-09-12

Contents

1 Purpose **1**

2 Basic usage **1**

3 Adding replacement variables at runtime **2**

1 Purpose

The `QueryNamespace` class is designed to be a convenient way to write (and re-write) SQL queries for packages with defined result model specifications. The convenience is that only passed parameters must be set in the query - any table names or other pre-defined variables can be set once in a result spec and reused if table prefixes are applied to the tables. The intention is to save time and limit bugs. This also builds on the `SqlRender/DatabaseConnector` principle of "Write sql once, use anywhere" principle across ohdsi packages. This is not intended to replace usage of `dbplyr` style operations which are expressive and allow use of sql. However, many find that writing SQL strings is often more convenient and portable to other programming language than `dplyr` calls allow.

2 Basic usage

The most basic usage is to create a specification with a single table that conforms to a valid data model specification

```
library(ResultModelManager)

tableSpecification <- data.frame(
  tableName = "cohort_definition",
  columnName = c("cohort_definition_id", "cohort_name", "json", "sql"),
  primaryKey = c("yes", "no", "no", "no"),
  dataType = c("bigint", "varchar", "varchar", "varchar")
)
```

Note, that generally we would save these tables to a csv file that can be loaded.

We then load a `QueryNamespace` instance with this table:

```r
connectionDetails <- DatabaseConnector::createConnectionDetails("sqlite", server = tempfile())
qns <- createQueryNamespace(
  connectionDetails = connectionDetails,
  usePooledConnection = FALSE,
  tableSpecification = tableSpecification,
  tablePrefix = "rwe_study_99_",
  snakeCaseToCamelCase = TRUE,
  database_schema = "main"
)
```

```
## Connecting using SQLite driver
```

```r
# Create our schema within the namespace
sql <- generateSqlSchema(schemaDefinition = tableSpecification)
# note - the table prefix and schema parameters are not neeeded
qns$executeSql(sql)
```

```
##   |
```

```
## Executing SQL took 0.00819 secs
```

We can then query the table with sql that automatically replaces the table names:

```r
qns$queryDb("SELECT * FROM @database_schema.@cohort_definition")
```

```
## [1] cohortDefinitionId cohortName         json               sql
## <0 rows> (or 0-length row.names)
```

Note that the underlying query is already handling our `tablePrefix` for us, so we don't need to add it:

```r
qns$queryDb("SELECT * FROM @database_schema.@cohort_definition")
```

```
## [1] cohortDefinitionId cohortName         json               sql
## <0 rows> (or 0-length row.names)
```

3 Adding replacement variables at runtime

Variables can naturally be added at runtime, for example, in a query:

```r
qns$queryDb("SELECT * FROM @database_schema.@cohort_definition WHERE cohort_definition_id = @id",
  id = 5
)
```

```
## [1] cohortDefinitionId cohortName         json               sql
## <0 rows> (or 0-length row.names)
```

Alternatively we can persist the id in the object for use in all queries.

```
qns$addReplacementVariable("database_id", "my_cdm")
```

Note that replacing the same variable will result in an error

```
qns$addReplacementVariable("database_id", "my_cdm")
```

We can also add to the table specification

```
tableSpecification2 <- data.frame(
  tableName = "database_info",
  columnName = c("database_id", "database_name"),
  primaryKey = c("yes", "no"),
  dataType = c("varchar", "varchar")
)
qns$addTableSpecification(tableSpecification2)
```