

# Package: ResultModelManager (via r-universe)

August 21, 2024

**Title** Result Model Manager

**Version** 0.5.10

**Description** Database data model management utilities for R packages in the Observational Health Data Sciences and Informatics program <<https://ohdsi.org>>. 'ResultModelManager' provides utility functions to allow package maintainers to migrate existing SQL database models, export and import results in consistent patterns.

**URL** <https://github.com/OHDSI/ResultModelManager>

**BugReports** <https://github.com/OHDSI/ResultModelManager/issues>

**License** Apache License (== 2.0)

**Encoding** UTF-8

**VignetteBuilder** knitr

**Roxygen** list(markdown = TRUE)

**RoxygenNote** 7.3.2

**Depends** R (>= 4.1.0), R6, DatabaseConnector (>= 6.0.0)

**Imports** SqlRender, ParallelLogger, checkmate, DBI, pool, readr, zip, dplyr, dbplyr, rlang, lubridate, fastmap, withr

**Suggests** testthat (>= 3.0.0), RSQLite, duckdb, knitr, rmarkdown, keyring, devtools, pkgdown, remotes, styler, Andromeda

**Config/testthat/edition** 3

**Repository** <https://ohdsi.r-universe.dev>

**RemoteUrl** <https://github.com/ohdsi/resultmodelmanager>

**RemoteRef** HEAD

**RemoteSha** c734b7e55a851f8e1eba0d2a0f78f4a1603641ff

## Contents

ConnectionHandler . . . . .	2
createQueryNamespace . . . . .	5
createResultExportManager . . . . .	6
DataMigrationManager . . . . .	7
deleteAllRowsForDatabaseId . . . . .	10
deleteAllRowsForPrimaryKey . . . . .	10
generateSqlSchema . . . . .	11
grantTablePermissions . . . . .	12
loadResultsDataModelSpecifications . . . . .	12
PooledConnectionHandler . . . . .	13
QueryNamespace . . . . .	16
ResultExportManager . . . . .	20
unzipResults . . . . .	23
uploadResults . . . . .	24
<b>Index</b>	<b>26</b>

---

ConnectionHandler	<i>ConnectionHandler</i>
-------------------	--------------------------

---

### Description

Class for handling DatabaseConnector:connection objects with consistent R6 interfaces for pooled and non-pooled connections. Allows a connection to cleanly be opened and closed and stored within class/object variables

### Value

DatabaseConnector Connection instance close Connection

boolean TRUE if connection is valid queryDb

boolean TRUE if connection is valid executeSql

### Public fields

connectionDetails DatabaseConnector connectionDetails object

con DatabaseConnector connection object

isActive Is connection active or not#'

snakeCaseToCamelCase (Optional) Boolean. return the results columns in camel case (default)

**Methods****Public methods:**

- `ConnectionHandler$new()`
- `ConnectionHandler$dbms()`
- `ConnectionHandler$tbl()`
- `ConnectionHandler$renderTranslateSql()`
- `ConnectionHandler$initConnection()`
- `ConnectionHandler$getConnection()`
- `ConnectionHandler$closeConnection()`
- `ConnectionHandler$finalize()`
- `ConnectionHandler$dbIsValid()`
- `ConnectionHandler$queryDb()`
- `ConnectionHandler$executeSql()`
- `ConnectionHandler$queryFunction()`
- `ConnectionHandler$executeFunction()`
- `ConnectionHandler$clone()`

**Method** `new()`:*Usage:*

```
ConnectionHandler$new(
  connectionDetails,
  loadConnection = TRUE,
  snakeCaseToCamelCase = TRUE
)
```

*Arguments:*

`connectionDetails` DatabaseConnector::connectionDetails class  
`loadConnection` Boolean option to load connection right away  
`snakeCaseToCamelCase` (Optional) Boolean. return the results columns in camel case (default)  
`get dbms`

**Method** `dbms()`: Get the dbms type of the connection get table*Usage:*

```
ConnectionHandler$dbms()
```

**Method** `tbl()`: get a dplyr table object (i.e. lazy loaded)*Usage:*

```
ConnectionHandler$tbl(table, databaseSchema = NULL)
```

*Arguments:*

`table` table name  
`databaseSchema` databaseSchema to which table belongs Render Translate Sql.

**Method** `renderTranslateSql()`: Masked call to SqlRender*Usage:*

```
ConnectionHandler$renderTranslateSql(sql, ...)
```

*Arguments:*

sql Sql query string

... Elipsis initConnection

**Method** `initConnection()`: Load connection Get Connection

*Usage:*

```
ConnectionHandler$initConnection()
```

**Method** `getConnection()`: Returns connection for use with standard DatabaseConnector calls. Connects automatically if it isn't yet loaded

*Usage:*

```
ConnectionHandler$getConnection()
```

**Method** `closeConnection()`: Closes connection (if active) close Connection

*Usage:*

```
ConnectionHandler$closeConnection()
```

**Method** `finalize()`: Closes connection (if active) db Is Valid

*Usage:*

```
ConnectionHandler$finalize()
```

**Method** `dbIsValid()`: Masks call to DBI::dbIsValid. Returns False if connection is NULL

*Usage:*

```
ConnectionHandler$dbIsValid()
```

**Method** `queryDb()`: query database and return the resulting data.frame

If environment variable LIMIT\_ROW\_COUNT is set Returned rows are limited to this value (no default) Limit row count is intended for web applications that may cause a denial of service if they consume too many resources.

*Usage:*

```
ConnectionHandler$queryDb(
  sql,
  snakeCaseToCamelCase = self$snakeCaseToCamelCase,
  overrideRowLimit = FALSE,
  ...
)
```

*Arguments:*

sql sql query string

snakeCaseToCamelCase (Optional) Boolean. return the results columns in camel case (default)

overrideRowLimit (Optional) Boolean. In some cases, where row limit is enforced on the system You may wish to ignore it.

... Additional query parameters

**Method** `executeSql()`: execute set of database queries

*Usage:*

```
ConnectionHandler$executeSql(sql, ...)
```

*Arguments:*

```
sql sql query string
... Additional query parameters query Function
```

**Method** queryFunction(): queryFunction that can be overridden with subclasses (e.g. use different base function or intercept query) Does not translate or render sql.

*Usage:*

```
ConnectionHandler$queryFunction(
  sql,
  snakeCaseToCamelCase = self$snakeCaseToCamelCase,
  connection = self$getConnection()
)
```

*Arguments:*

```
sql sql query string
snakeCaseToCamelCase (Optional) Boolean. return the results columns in camel case (default)
connection (Optional) connection object execute Function
```

**Method** executeFunction(): exec query Function that can be overridden with subclasses (e.g. use different base function or intercept query) Does not translate or render sql.

*Usage:*

```
ConnectionHandler$executeFunction(sql, connection = self$getConnection())
```

*Arguments:*

```
sql sql query string
connection connection object
```

**Method** clone(): The objects of this class are cloneable with this method.

*Usage:*

```
ConnectionHandler$clone(deep = FALSE)
```

*Arguments:*

```
deep Whether to make a deep clone.
```

---

createQueryNamespace *Create query namespace*

---

**Description**

Create a QueryNamespace instance from either a connection handler or a connectionDetails object  
Allows construction with various options not handled by QueryNamespace\$new

Note - currently not supported is having multiple table prefixes for multiple table namespaces

**Usage**

```

createQueryNamespace(
    connectionDetails = NULL,
    connectionHandler = NULL,
    usePooledConnection = FALSE,
    tableSpecification = NULL,
    resultModelSpecificationPath = NULL,
    tablePrefix = "",
    snakeCaseToCamelCase = TRUE,
    ...
)

```

**Arguments**

connectionDetails	An object of type connectionDetails as created using the <a href="#">createConnectionDetails</a> function in the DatabaseConnector package.
connectionHandler	ResultModelManager ConnectionHandler or PooledConnectionHandler instance
usePooledConnection	Use Pooled database connection instead of standard DatabaseConnector single connection.
tableSpecification	Table specification data.frame
resultModelSpecificationPath	(optional) csv file or files for tableSpecifications - must conform to table spec format.
tablePrefix	String to prefix table names with - default is empty string
snakeCaseToCamelCase	convert snakecase results to camelCase field names (TRUE by default)
...	Elipsis - use for any additional string keys to replace

---

```
createResultExportManager
```

*Create Result Export Manager*

---

**Description**

For a give table specification file, create an export manager instance for creating results data sets that conform to the data model.

This checks that, at export time, internal validity is assured for the data (e.g. primary keys are valid, data types are compatible)

In addition this utility will create a manifest object that can be used to maintain the validity of data.

If an instance of a DataMigrationManager is present and available a packageVersion reference (where applicable) and migration set will be referenced. Allowing data to be imported into a database schema at a specific version.

**Usage**

```
createResultExportManager(
  tableSpecification,
  exportDir,
  minCellCount = getOption("ohdsi.minCellCount", default = 5),
  databaseId = NULL
)
```

**Arguments**

tableSpecification	Table specification data.frame
exportDir	Directory files are being exported to
minCellCount	Minimum cell count - recommended that you set with options("ohdsi.minCellCount" = count) in all R projects. Default is 5
databaseId	database identifier - required when exporting according to many specs

---

DataMigrationManager *DataMigrationManager (DMM)*

---

**Description**

R6 class for management of database migration

**Value**

data frame all migrations, including file name, order and execution status Get connection handler

**Public fields**

migrationPath Path migrations exist in  
 databaseSchema Path migrations exist in  
 packageName packageName, can be null  
 tablePrefix tablePrefix, can be empty character vector  
 packageTablePrefix packageTablePrefix, can be empty character vector

**Methods****Public methods:**

- [DataMigrationManager\\$new\(\)](#)
- [DataMigrationManager\\$migrationTableExists\(\)](#)
- [DataMigrationManager\\$getMigrationsPath\(\)](#)
- [DataMigrationManager\\$getStatus\(\)](#)
- [DataMigrationManager\\$getConnectionHandler\(\)](#)

- `DataMigrationManager$check()`
- `DataMigrationManager$executeMigrations()`
- `DataMigrationManager$isPackage()`
- `DataMigrationManager$finalize()`
- `DataMigrationManager$clone()`

**Method new():**

*Usage:*

```
DataMigrationManager$new(
  connectionDetails,
  databaseSchema,
  tablePrefix = "",
  packageTablePrefix = "",
  migrationPath,
  packageName = NULL,
  migrationRegexp = .defaultMigrationRegexp
)
```

*Arguments:*

`connectionDetails` DatabaseConnector connection details object

`databaseSchema` Database Schema to execute on

`tablePrefix` Optional table prefix for all tables (e.g. plp, cm, cd etc)

`packageTablePrefix` A table prefix when used in conjunction with other package results schema, e.g. "cd\_", "scs\_", "plp\_", "cm\_"

`migrationPath` Path to location of migration sql files. If in package mode, this should just be a folder (e.g. "migrations") that lives in the location "sql/sql\_server" (and) other database platforms. If in folder model, the folder must include "sql\_server" in the relative path, (e.g if migrationPath = 'migrations' then the folder 'migrations/sql\_server' should exist)

`packageName` If in package mode, the name of the R package

`migrationRegexp` (Optional) regular expression pattern default is `(Migration_([0-9]+))-(.+).sql`  
Migration table exists

**Method migrationTableExists():** Check if migration table is present in schema

*Usage:*

```
DataMigrationManager$migrationTableExists()
```

*Returns:* boolean Get path of migrations

**Method getMigrationsPath():** Get path to sql migration files

*Usage:*

```
DataMigrationManager$getMigrationsPath(dbms = "sql server")
```

*Arguments:*

`dbms` Optionally specify the dbms that the migration fits under Get status of result model

**Method getStatus():** Get status of all migrations (executed or not)

*Usage:*



DataMigrationManager\$getStatus()

**Method** getConnectionHandler(): Return connection handler instance

*Usage:*

DataMigrationManager\$getConnectionHandler()

*Returns:* ConnectionHandler instance Check migrations in folder

**Method** check(): Check if file names are valid for migrations Execute Migrations

*Usage:*

DataMigrationManager\$check()

**Method** executeMigrations(): Execute any unexecuted migrations

*Usage:*

DataMigrationManager\$executeMigrations(stopMigrationVersion = NULL)

*Arguments:*

stopMigrationVersion (Optional) Migrate to a specific migration number isPackage

**Method** isPackage(): is a package folder structure or not finalize

*Usage:*

DataMigrationManager\$isPackage()

**Method** finalize(): close database connection

*Usage:*

DataMigrationManager\$finalize()

**Method** clone(): The objects of this class are cloneable with this method.

*Usage:*

DataMigrationManager\$clone(deep = FALSE)

*Arguments:*

deep Whether to make a deep clone.

### See Also

[ConnectionHandler](#) for information on returned class

deleteAllRowsForDatabaseId

*Delete all rows for database id*

---

### **Description**

Delete all rows for database id

### **Usage**

```
deleteAllRowsForDatabaseId(  
    connection,  
    schema,  
    tableName,  
    databaseId,  
    idIsInt = TRUE  
)
```

### **Arguments**

connection	DatabaseConnector connection instance
schema	The schema on the postgres server where the results table exists
tableName	Database table name
databaseId	Results source database identifier
idIsInt	Identified is a numeric type? If not character is used

### **Details**

Only PostgreSQL servers are supported.

---

deleteAllRowsForPrimaryKey

*Delete results rows for primary key values from database server tables*

---

### **Description**

Delete results rows for primary key values from database server tables

### **Usage**

```
deleteAllRowsForPrimaryKey(connection, schema, tableName, keyValues)
```

**Arguments**

connection	DatabaseConnector connection instance
schema	The schema on the postgres server where the results table exists
tableName	Database table name
keyValues	Key values of results rows to be deleted

**Details**

Only PostgreSQL servers are supported.

---

generateSqlSchema	<i>Schema generator</i>
-------------------	-------------------------

---

**Description**

Take a csv schema definition and create a basic sql script with it. returns string containing the sql for the table

**Usage**

```
generateSqlSchema(
  csvFilepath = NULL,
  schemaDefinition = NULL,
  sqlOutputPath = NULL,
  overwrite = FALSE
)
```

**Arguments**

csvFilepath	Path to schema file. Csv file must have the columns: "table_name", "column_name", "data_type", "primary_key"
schemaDefinition	A schemaDefintion data.frame' with the columns: tableName, columnName, dataType, isRequired, primaryKey
sqlOutputPath	File to write sql to.
overwrite	Boolean - overwrite existing file?

---

grantTablePermissions *Grant Table Permissions*

---

### Description

Grant a given permission for all tables on a given tableSpecification

Very useful if you're hosting studies on data.ohdsi.org or other postgresql instances

NOTE: only tested on postgresql, users' of other platforms may have Sql translation issues

### Usage

```
grantTablePermissions(
  connectionDetails = NULL,
  connection = NULL,
  tableSpecification,
  databaseSchema,
  tablePrefix = "",
  permissions = "SELECT",
  user
)
```

### Arguments

connectionDetails	An object of type connectionDetails as created using the <a href="#">createConnectionDetails</a> function in the DatabaseConnector package.
connection	DatabaseConnector connection instance
tableSpecification	data.frame conforming to table spec (must contain tableName field)
databaseSchema	database schema to run this on
tablePrefix	String to prefix table names with - default is empty string
permissions	permissions to generate must be one of SELECT, INSERT, DELETE or UPDATE
user	database user to grant permissions to

---

loadResultsDataModelSpecifications

*Get specifications from a given file path*

---

### Description

Get specifications from a given file path

**Usage**

```
loadResultsDataModelSpecifications(filePath)
```

**Arguments**

filePath            path to a valid csv file

**Value**

A tibble data frame object with specifications

---

PooledConnectionHandler

*Pooled Connection Handler*

---

**Description**

Transparently works the same way as a standard connection handler but stores pooled connections. Useful for long running applications that serve multiple concurrent requests. Note that a side effect of using this is that each call to this increments the .GlobalEnv attribute RMPooledHandlerCount

**Value**

boolean TRUE if connection is valid executeSql

**Super class**

[ResultModelManager::ConnectionHandler](#) -> PooledConnectionHandler

**Methods****Public methods:**

- [PooledConnectionHandler\\$new\(\)](#)
- [PooledConnectionHandler\\$initConnection\(\)](#)
- [PooledConnectionHandler\\$getCheckedOutConnectionPath\(\)](#)
- [PooledConnectionHandler\\$getConnection\(\)](#)
- [PooledConnectionHandler\\$dbms\(\)](#)
- [PooledConnectionHandler\\$closeConnection\(\)](#)
- [PooledConnectionHandler\\$queryDb\(\)](#)
- [PooledConnectionHandler\\$executeSql\(\)](#)
- [PooledConnectionHandler\\$queryFunction\(\)](#)
- [PooledConnectionHandler\\$executeFunction\(\)](#)
- [PooledConnectionHandler\\$clone\(\)](#)

**Method** [new\(\)](#):

*Usage:*

```
PooledConnectionHandler$new(
  connectionDetails = NULL,
  snakeCaseToCamelCase = TRUE,
  loadConnection = TRUE,
  dbConnectArgs = NULL,
  forceJdbcConnection = TRUE
)
```

*Arguments:*

connectionDetails DatabaseConnector::connectionDetails class  
snakeCaseToCamelCase (Optional) Boolean. return the results columns in camel case (default)  
loadConnection Boolean option to load connection right away  
dbConnectArgs Optional arguments to call pool::dbPool overrides default usage of connectionDetails  
forceJdbcConnection Force JDBC connection (requires using DatabaseConnector ConnectionDetails) initialize pooled db connection

**Method** `initConnection()`: Overrides ConnectionHandler Call Used for getting a checked out connection from a given environment (if one exists)

*Usage:*

```
PooledConnectionHandler$initConnection()
```

**Method** `getCheckedOutConnectionPath()`:

*Usage:*

```
PooledConnectionHandler$getCheckedOutConnectionPath()
```

*Arguments:*

.deferredFrame defaults to the parent frame of the calling block. Get Connection

**Method** `getConnection()`: Returns a connection from the pool When the desired frame exits, the connection will be returned to the pool As a side effect, the connection is stored as an attribute within the calling frame (e.g. the same function) to prevent multiple connections being spawned, which limits performance.

If you call this somewhere you need to think about returning the object or you may create a connection that is never returned to the pool.

*Usage:*

```
PooledConnectionHandler$getConnection(.deferredFrame = parent.frame(n = 2))
```

*Arguments:*

.deferredFrame defaults to the parent frame of the calling block. get dbms

**Method** `dbms()`: Get the dbms type of the connection Close Connection

*Usage:*

```
PooledConnectionHandler$dbms()
```

**Method** `closeConnection()`: Overrides ConnectionHandler Call - closes all active connections called with getConnection queryDb

*Usage:*

```
PooledConnectionHandler$closeConnection()
```

**Method** queryDb(): query database and return the resulting data.frame

If environment variable LIMIT\_ROW\_COUNT is set Returned rows are limited to this value (no default) Limit row count is intended for web applications that may cause a denial of service if they consume too many resources.

*Usage:*

```
PooledConnectionHandler$queryDb(
  sql,
  snakeCaseToCamelCase = self$snakeCaseToCamelCase,
  overrideRowLimit = FALSE,
  ...
)
```

*Arguments:*

sql sql query string

snakeCaseToCamelCase (Optional) Boolean. return the results columns in camel case (default)

overrideRowLimit (Optional) Boolean. In some cases, where row limit is enforced on the system You may wish to ignore it.

... Additional query parameters

**Method** executeSql(): execute set of database queries

*Usage:*

```
PooledConnectionHandler$executeSql(sql, ...)
```

*Arguments:*

sql sql query string

... Additional query parameters query Function

**Method** queryFunction(): Overrides ConnectionHandler Call. Does not translate or render sql.

*Usage:*

```
PooledConnectionHandler$queryFunction(
  sql,
  snakeCaseToCamelCase = self$snakeCaseToCamelCase,
  connection
)
```

*Arguments:*

sql sql query string

snakeCaseToCamelCase (Optional) Boolean. return the results columns in camel case (default)  
query Function

connection db connection assumes pooling is handled outside of call

**Method** executeFunction(): Overrides ConnectionHandler Call. Does not translate or render sql.

*Usage:*

PooledConnectionHandler\$executeFunction(sql, connection)

*Arguments:*

sql sql query string

connection DatabaseConnector connection. Assumes pooling is handled outside of call

**Method clone():** The objects of this class are cloneable with this method.

*Usage:*

PooledConnectionHandler\$clone(deep = FALSE)

*Arguments:*

deep Whether to make a deep clone.

---

QueryNamespace

*QueryNamespace*

---

## Description

Given a results specification and ConnectionHandler instance - this class allow queries to be namespaced within any tables specified within a list of pre-determined tables. This allows the encapsulation of queries, using specific table names in a consistent manner that is straightforward to maintain over time.

## Public fields

tablePrefix tablePrefix to use

## Methods

### Public methods:

- [QueryNamespace\\$new\(\)](#)
- [QueryNamespace\\$setConnectionHandler\(\)](#)
- [QueryNamespace\\$getConnectionHandler\(\)](#)
- [QueryNamespace\\$addReplacementVariable\(\)](#)
- [QueryNamespace\\$addTableSpecification\(\)](#)
- [QueryNamespace\\$render\(\)](#)
- [QueryNamespace\\$queryDb\(\)](#)
- [QueryNamespace\\$executeSql\(\)](#)
- [QueryNamespace\\$getVars\(\)](#)
- [QueryNamespace\\$finalize\(\)](#)
- [QueryNamespace\\$clone\(\)](#)

**Method new():** initialize class

*Usage:*



```

QueryNamespace$new(
  connectionHandler = NULL,
  tableSpecification = NULL,
  tablePrefix = "",
  ...
)

```

*Arguments:*

**connectionHandler** ConnectionHandler instance @seealso [ConnectionHandler](#)  
**tableSpecification** tableSpecification data.frame  
**tablePrefix** constant string to prefix all tables with  
 ... additional replacement variables e.g. database\_schema, vocabulary\_schema etc Set Connection Handler

**Method** `setConnectionHandler()`: set connection handler object for object

*Usage:*

```
QueryNamespace$setConnectionHandler(connectionHandler)
```

*Arguments:*

**connectionHandler** ConnectionHandler instance Get connection handler

**Method** `getConnectionHandler()`: get connection handler object or throw error if not set

*Usage:*

```
QueryNamespace$getConnectionHandler()
```

**Method** `addReplacementVariable()`: add a variable to automatically be replaced in query strings (e.g. @database\_schema.@table\_name becomes 'database\_schema.table\_1')

*Usage:*

```
QueryNamespace$addReplacementVariable(key, value, replace = FALSE)
```

*Arguments:*

**key** variable name string (without @) to be replaced, eg. "table\_name"  
**value** atomic value for replacement  
**replace** if a variable of the same key is found, overwrite it add table specification

**Method** `addTableSpecification()`: add a variable to automatically be replaced in query strings (e.g. @database\_schema.@table\_name becomes 'database\_schema.table\_1')

*Usage:*

```

QueryNamespace$addTableSpecification(
  tableSpecification,
  useTablePrefix = TRUE,
  tablePrefix = self$tablePrefix,
  replace = TRUE
)

```

*Arguments:*

**tableSpecification** table specification data.frame conforming to column names tableName, columnName, dataType and primaryKey

useTablePrefix prefix the results with the tablePrefix (TRUE)  
 tablePrefix prefix string - defaults to class variable set during initialization  
 replace replace existing variables of the same name Render

**Method** render(): Call to SqlRender::render replacing names stored in this class

*Usage:*

QueryNamespace\$render(sql, ...)

*Arguments:*

sql query string

... additional variables to be passed to SqlRender::render - will overwrite anything in namespace query Sql

**Method** queryDb(): Call to

*Usage:*

QueryNamespace\$queryDb(sql, ...)

*Arguments:*

sql query string

... additional variables to send to SqlRender::render execute Sql

**Method** executeSql(): Call to execute sql within namespaced queries

*Usage:*

QueryNamespace\$executeSql(sql, ...)

*Arguments:*

sql query string

... additional variables to send to SqlRender::render get vars

**Method** getVars(): returns full list of variables that will be replaced Destruct object

*Usage:*

QueryNamespace\$getVars()

**Method** finalize(): Close connections etc

*Usage:*

QueryNamespace\$finalize()

**Method** clone(): The objects of this class are cloneable with this method.

*Usage:*

QueryNamespace\$clone(deep = FALSE)

*Arguments:*

deep Whether to make a deep clone.

**Examples**

```

library(ResultModelManager)

# Create some junk test data
connectionDetails <-
  DatabaseConnector::createConnectionDetails(
    server = "test_db.sqlite",
    dbms = "sqlite"
  )

conn <- DatabaseConnector::connect(connectionDetails)
DatabaseConnector::insertTable(
  connection = conn,
  tableName = "cd_cohort",
  data = data.frame(
    cohort_id = c(1, 2, 3),
    cohort_name = c("cohort one", "cohort two", "cohort three"),
    json = "{}",
    sql = "SELECT 1"
  )
)
DatabaseConnector::disconnect(conn)

connectionHandler <- ConnectionHandler$new(connectionDetails = connectionDetails)
tableSpecification <- data.frame(
  tableName = "cohort",
  columnName = c(
    "cohort_id",
    "cohort_name",
    "json",
    "sql"
  ),
  primaryKey = c(TRUE, FALSE, FALSE, FALSE),
  dataType = c("int", "varchar", "varchar", "varchar")
)

cohortNamespace <- QueryNamespace$new(
  connectionHandler = connectionHandler,
  tableSpecification = tableSpecification,
  result_schema = "main",
  tablePrefix = "cd_"
)
sql <- "SELECT * FROM @result_schema.@cohort WHERE cohort_id = @cohort_id"
# Returns : "SELECT * FROM main.cd_cohort WHERE cohort_id = @cohort_id"
print(cohortNamespace$render(sql))
# Returns query result
result <- cohortNamespace$queryDb(sql, cohort_id = 1)
# cleanup test data
unlink("test_db.sqlite")

```

---

ResultExportManager     *Result Set Export Manager*

---

## Description

EXPERIMENTAL - this feature is still in design stage and it is not recommended that you implement this for your package at this stage. Utility for simplifying export of results to files from sql queries

Note that this utility is not strictly thread safe though separate processes can export separate tables without issue. When exporting a the same table across multiple threads primary key checks may create issues.

## Public fields

exportDir    directory path to export files to Init

## Methods

### Public methods:

- [ResultExportManager\\$new\(\)](#)
- [ResultExportManager\\$getTableSpec\(\)](#)
- [ResultExportManager\\$getMinColValues\(\)](#)
- [ResultExportManager\\$checkRowTypes\(\)](#)
- [ResultExportManager\\$listTables\(\)](#)
- [ResultExportManager\\$checkPrimaryKeys\(\)](#)
- [ResultExportManager\\$exportDataFrame\(\)](#)
- [ResultExportManager\\$exportQuery\(\)](#)
- [ResultExportManager\\$getManifestList\(\)](#)
- [ResultExportManager\\$writeManifest\(\)](#)
- [ResultExportManager\\$clone\(\)](#)

**Method new():** Create a class for exporting results from a study in a standard, consistent manner

*Usage:*

```
ResultExportManager$new(
  tableSpecification,
  exportDir,
  minCellCount = getOption("ohdsi.minCellCount", default = 5),
  databaseId = NULL
)
```

*Arguments:*

tableSpecification    Table specification data.frame

exportDir    Directory files are being exported to

minCellCount    Minimum cell count - recommended that you set with options("ohdsi.minCellCount" = count) in all R projects. Default is 5

databaseId database identifier - required when exporting according to many specs  
 spec

**Method** getTableSpec(): Get specification of table

*Usage:*

```
ResultExportManager$getTableSpec(exportTableName)
```

*Arguments:*

exportTableName table name Get min col values

**Method** getMinColValues(): Columns to convert to minimum for a given table name

*Usage:*

```
ResultExportManager$getMinColValues(rows, exportTableName)
```

*Arguments:*

rows data.frame of rows

exportTableName string table name - must be defined in spec Check row types

**Method** checkRowTypes(): Check types of rows before exporting

*Usage:*

```
ResultExportManager$checkRowTypes(rows, exportTableName)
```

*Arguments:*

rows data.frame of rows to export

exportTableName table name List tables

**Method** listTables(): list all tables in schema Check primary keys of exported data

*Usage:*

```
ResultExportManager$listTables()
```

**Method** checkPrimaryKeys(): Checks to see if the rows conform to the valid primary keys If the same table has already been checked in the life of this object set "invalidateCache" to TRUE as the keys will be cached in a temporary file on disk.

*Usage:*

```
ResultExportManager$checkPrimaryKeys(
  rows,
  exportTableName,
  invalidateCache = FALSE
)
```

*Arguments:*

rows data.frame to export

exportTableName Table name (must be in spec)

invalidateCache logical - if starting a fresh export use this to delete cache of primary keys  
 Export data frame

**Method** exportDataFrame(): This method is intended for use where exporting a data.frame and not a query from a rdbms table For example, if you perform a transformation in R this method will check primary keys, min cell counts and data types before writing the file to according to the table spec

*Usage:*

```
ResultExportManager$exportDataFrame(rows, exportTableName, append = FALSE)
```

*Arguments:*

rows Rows to export

exportTableName Table name

append logical - if true will append the result to a file, otherwise the file will be overwritten  
Export Data table with sql query

**Method** exportQuery(): Writes files in batch to stop overflowing system memory Checks primary keys on write Checks minimum cell count

*Usage:*

```
ResultExportManager$exportQuery(
  connection,
  sql,
  exportTableName,
  transformFunction = NULL,
  transformFunctionArgs = list(),
  append = FALSE,
  ...
)
```

*Arguments:*

connection DatabaseConnector connection instance

sql OHDSI sql string to export tables

exportTableName Name of table to export (in snake\_case format)

transformFunction (optional) transformation of the data set callback. must take two paramters  
- rows and pos

Following this transformation callback, results will be verified against data model,  
Primary keys will be checked and minCellValue rules will be enforced

transformFunctionArgs arguments to be passed to the transformation function

append Logical add results to existing file, if FALSE (default) creates a new file and removes  
primary key validation cache

... extra parameters passed to sql get manifest list

**Method** getManifestList(): Create a meta data set for each collection of result files with sha256 has for all files

*Usage:*

```
ResultExportManager$getManifestList(
  packageName = NULL,
  packageVersion = NULL,
  migrationsPath = NULL,
  migrationRegexp = .defaultMigrationRegexp
)
```

*Arguments:*

packageName if an R analysis package, specify the name

packageVersion if an analysis package, specify the version  
 migrationsPath path to sql migrations (use top level folder (e.g. sql/sql\_server/migrations)  
 migrationRegexp (optional) regular expression to search for sql files. It is not recommended to  
 change the default. Write manifest

**Method** writeManifest(): Write manifest json

*Usage:*

ResultExportManager\$writeManifest(...)

*Arguments:*

... @seealso getManifestList

**Method** clone(): The objects of this class are cloneable with this method.

*Usage:*

ResultExportManager\$clone(deep = FALSE)

*Arguments:*

deep Whether to make a deep clone.

---

unzipResults	<i>Unzips a results.zip file and enforces standards required by uploadResults</i>
--------------	---

---

## Description

This function will unzip the zipFile to the resultsFolder and assert that the file resultsDataModel-Specification.csv exists in the resultsFolder to ensure that it will work with uploadResults

## Usage

```
unzipResults(zipFile, resultsFolder)
```

## Arguments

zipFile	The location of the .zip file that holds the results to upload
resultsFolder	The folder to use when unzipping the .zip file. If this folder does not exist, this function will attempt to create the folder.

---

 uploadResults

*Upload results to the database server.*


---

### Description

Requires the results data model tables have been created using following the specifications, generateSqlSchema function.

Results files should be in the snake\_case format for table headers and not camelCase

Set the POSTGRES\_PATH environmental variable to the path to the folder containing the psql executable to enable bulk upload (recommended).

### Usage

```
uploadResults(
  connection = NULL,
  connectionDetails = NULL,
  schema,
  resultsFolder,
  tablePrefix = "",
  forceOverWriteOfSpecifications = FALSE,
  purgeSiteDataBeforeUploading = TRUE,
  databaseIdentifierFile = "cdm_source_info.csv",
  runCheckAndFixCommands = FALSE,
  warnOnMissingTable = TRUE,
  purgeDataModel = FALSE,
  specifications
)
```

### Arguments

- |                                |  |
|--------------------------------|--|
| connection                     | An object of type connection as created using the <a href="#">connect</a> function in the DatabaseConnector package. Can be left NULL if connectionDetails is provided, in which case a new connection will be opened at the start of the function, and closed when the function finishes. |
| connectionDetails              | An object of type connectionDetails as created using the <a href="#">createConnectionDetails</a> function in the DatabaseConnector package.  |
| schema                         | The schema on the postgres server where the tables have been created.  |
| resultsFolder                  | The path to the folder containing the results to upload. See unzipResults for more information.  |
| tablePrefix                    | String to prefix table names with - default is empty string  |
| forceOverWriteOfSpecifications | If TRUE, specifications of the phenotypes, cohort definitions, and analysis will be overwritten if they already exist on the database. Only use this if these specifications have changed since the last upload.   |



- `purgeSiteDataBeforeUploading` If TRUE, before inserting data for a specific `databaseId` all the data for that site will be dropped. This assumes the results folder contains the full data for that data site.
- `databaseIdentifierFile` File contained that references `databaseId` field (used when `purgeSiteDataBeforeUploading == TRUE`). You may specify a relative path for the `cdmSourceFile` and the function will assume it resides in the `resultsFolder`. Alternatively, you can provide a path outside of the `resultsFolder` for this file.
- `runCheckAndFixCommands` If TRUE, the upload code will attempt to fix column names, data types and duplicate rows. This parameter is kept for legacy reasons - it is strongly recommended that you correct errors in your results where those results are assembled instead of relying on this option to try and fix it during upload.
- `warnOnMissingTable` Boolean, print a warning if a table file is missing.
- `purgeDataModel` This function will purge all data from the tables in the specification prior to upload. Use with care. If interactive this will require further input.
- `specifications` A tibble data frame object with specifications.

# Index

connect, [24](#)  
ConnectionHandler, [2](#), [9](#), [17](#)  
createConnectionDetails, [6](#), [12](#), [24](#)  
createQueryNamespace, [5](#)  
createResultExportManager, [6](#)  
  
DataMigrationManager, [7](#)  
deleteAllRowsForDatabaseId, [10](#)  
deleteAllRowsForPrimaryKey, [10](#)  
  
generateSqlSchema, [11](#)  
grantTablePermissions, [12](#)  
  
loadResultsDataModelSpecifications, [12](#)  
  
PooledConnectionHandler, [13](#)  
  
QueryNamespace, [16](#)  
  
ResultExportManager, [20](#)  
ResultModelManager::ConnectionHandler,  
[13](#)  
  
unzipResults, [23](#)  
uploadResults, [24](#)