# Package 'DatabaseConnector'

May 7, 2026

**Type** Package

**Title** Connecting to Various Database Platforms

**Version** 7.1.0

**Date** 2026-01-08

**Description** An R 'DataBase Interface' ('DBI') compatible interface to various database platforms ('PostgreSQL', 'Oracle', 'Microsoft SQL Server', 'Amazon Redshift', 'Microsoft Parallel Database Warehouse', 'IBM Netezza', 'Apache Impala', 'Google BigQuery', 'Snowflake', 'Spark', 'SQLite', and 'InterSystems IRIS'). Also includes support for fetching data as 'Andromeda' objects. Uses either 'Java Database Connectivity' ('JDBC') or other 'DBI' drivers to connect to databases.

**SystemRequirements** Java (>= 8)

**Depends** R (>= 4.0.0)

**Imports** rJava, SqlRender (>= 1.19.2), methods, stringr, readr, rlang, utils, DBI (>= 1.0.0), urltools, bit64, checkmate, digest, dbplyr (>= 2.2.0)

**Suggests** aws.s3, R.utils, withr, testthat, DBItest, knitr, rmarkdown, RSQLite, ssh, Andromeda, dplyr, RPostgres, odbc, duckdb, bigrquery, pool, ParallelLogger, AzureStor

**License** Apache License

**VignetteBuilder** knitr

**URL** https://ohdsi.github.io/DatabaseConnector/, https://github.com/OHDSI/DatabaseConnector

**BugReports** https://github.com/OHDSI/DatabaseConnector/issues

**Copyright** See file COPYRIGHTS

**RoxygenNote** 7.3.3

**Encoding** UTF-8

**NeedsCompilation** no

**Author** Martijn Schuemie [aut, cre],
Marc Suchard [aut],
Adam Black [aut],
Observational Health Data Science and Informatics [cph],
Microsoft Inc. [cph] (SQL Server JDBC driver),
PostgreSQL Global Development Group [cph] (PostgreSQL JDBC driver),
Oracle Inc. [cph] (Oracle JDBC driver),
Amazon Inc. [cph] (RedShift JDBC driver)

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

**Repository** CRAN

**Date/Publication** 2026-01-09 09:31:28 UTC

# Contents

---

`assertTempEmulationSchemaSet` *Assert the temp emulation schema is set*

---

## Description

Asserts the temp emulation schema is set for DBMSs requiring temp table emulation.

If you know your code uses temp tables, it is a good idea to call this function first, so it can throw an informative error if the user forgot to set the temp emulation schema.

### Usage

```
assertTempEmulationSchemaSet(
  dbms,
  tempEmulationSchema = getOption("sqlRenderTempEmulationSchema")
)
```

### Arguments

| | |
|---|---|
| dbms | The type of DBMS running on the server. See connect() or createConnectionDetails() for valid values. |
| tempEmulationSchema | The temp emulation schema specified by the user. |

### Value

Does not return anything. Throws an error if the DBMS requires temp emulation but the temp emulation schema is not set.

---

## computeDataHash *Compute hash of data*

---

### Description

Compute a hash of the data in the database schema. If the data changes, this should produce a different hash code. Specifically, the hash is based on the field names, field types, and table row counts.

### Usage

```
computeDataHash(connection, databaseSchema, tables = NULL, progressBar = TRUE)
```

### Arguments

| | |
|---|---|
| connection | The connection to the database server created using either connect() or DBI::dbConnect(). |
| databaseSchema | The name of the database schema. See details for platform-specific details. |
| tables | (Optional) A list of tables to restrict to. |
| progressBar | When true, a progress bar is shown based on the number of tables in the database schema. |

### Details

The databaseSchema argument is interpreted differently according to the different platforms: SQL Server and PDW: The databaseSchema schema should specify both the database and the schema, e.g. 'my_database.dbo'. Impala: the databaseSchema should specify the database. Oracle: The databaseSchema should specify the Oracle 'user'. All other : The databaseSchema should specify the schema.

## Value

A string representing the MD5 hash code.

---

`connect` *connect*

---

## Description

Creates a connection to a database server .There are four ways to call this function:

- `connect(dbms, user, password, server, port, extraSettings, oracleDriver, pathToDriver)`
- `connect(connectionDetails)`
- `connect(dbms, connectionString, pathToDriver))`
- `connect(dbms, connectionString, user, password, pathToDriver)`

**DBMS parameter details::**

Depending on the DBMS, the function arguments have slightly different interpretations:

Oracle:

- `user`. The user name used to access the server
- `password`. The password for that user
- `server`. This field contains the SID, or host and servicename, SID, or TNSName: 'sid', 'host/sid', 'host/service name', or 'tnsname'
- `port`. Specifies the port on the server (default = 1521)
- `extraSettings`. The configuration settings for the connection (i.e. SSL Settings such as "(PROTOCOL=tcps)")
- `oracleDriver`. The driver to be used. Choose between "thin" or "oci".
- `pathToDriver`. The path to the folder containing the Oracle JDBC driver JAR files.

Microsoft SQL Server:

- `user`. The user used to log in to the server. If the user is not specified, Windows Integrated Security will be used, which requires the SQL Server JDBC drivers to be installed (see details below).
- `password`. The password used to log on to the server
- `server`. This field contains the host name of the server
- `port`. Not used for SQL Server
- `extraSettings`. The configuration settings for the connection (i.e. SSL Settings such as "encrypt=true; trustServerCertificate=false;")
- `pathToDriver`. The path to the folder containing the SQL Server JDBC driver JAR files.

Microsoft PDW:

- `user`. The user used to log in to the server. If the user is not specified, Windows Integrated Security will be used, which requires the SQL Server JDBC drivers to be installed (see details below).
- `password`. The password used to log on to the server

- `server`. This field contains the host name of the server
- `port`. Not used for SQL Server
- `extraSettings`. The configuration settings for the connection (i.e. SSL Settings such as "encrypt=true; trustServerCertificate=false;")
- `pathToDriver`. The path to the folder containing the SQL Server JDBC driver JAR files.

PostgreSQL:

- `user`. The user used to log in to the server
- `password`. The password used to log on to the server
- `server`. This field contains the host name of the server and the database holding the relevant schemas: host/database
- `port`. Specifies the port on the server (default = 5432)
- `extraSettings`. The configuration settings for the connection (i.e. SSL Settings such as "ssl=true")
- `pathToDriver`. The path to the folder containing the PostgreSQL JDBC driver JAR files.

Redshift:

- `user`. The user used to log in to the server
- `password`. The password used to log on to the server
- `server`. This field contains the host name of the server and the database holding the relevant schemas: host/database
- `port`. Specifies the port on the server (default = 5439)
- ‘extraSettings The configuration settings for the connection (i.e. SSL Settings such as "ssl=true&sslfactory=com.amazo
- `pathToDriver`. The path to the folder containing the RedShift JDBC driver JAR files.

Netezza:

- `user`. The user used to log in to the server
- `password`. The password used to log on to the server
- `server`. This field contains the host name of the server and the database holding the relevant schemas: host/database
- `port`. Specifies the port on the server (default = 5480)
- `extraSettings`. The configuration settings for the connection (i.e. SSL Settings such as "ssl=true")
- `pathToDriver`. The path to the folder containing the Netezza JDBC driver JAR file (nzjdbc.jar).

Impala:

- `user`. The user name used to access the server
- `password`. The password for that user
- `server`. The host name of the server
- `port`. Specifies the port on the server (default = 21050)
- `extraSettings`. The configuration settings for the connection (i.e. SSL Settings such as "SSLKeyStorePwd=*****")
- `pathToDriver`. The path to the folder containing the Impala JDBC driver JAR files.

SQLite:

- `server`. The path to the SQLIte file.

DuckDB:

- `server`. The path to the DuckDB file.
- `extraSettings`. Additional settings for DuckDB. For DuckDB specifically, if `extraSettings$config` is provided, it will be passed to the `duckdb::duckdb()` constructor. For example: `extraSettings = list(config = list(memory_limit = "8GB", preserve_insertion_order = "false"))`

Spark / Databricks:
Currently both JDBC and ODBC connections are supported for Spark. Set the `connectionString` argument to use JDBC, otherwise ODBC is used:

- `connectionString`. The JDBC connection string (e.g. something like 'jdbc:databricks://my-org.cloud.databricks.com:443/default;transportMode=http;ssl=1;AuthMech=3;httpPath=/sql/1.0/warehouses/abcde12
- `user`. The user name used to access the server. This can be set to 'token' when using a personal token (recommended).
- `password`. The password for that user. This should be your personal token when using a personal token (recommended).
- `server`. The host name of the server (when using ODBC), e.g. 'my-org.cloud.databricks.com')
- `port`. Specifies the port on the server (when using ODBC)
- `extraSettings`. Additional settings for the ODBC connection, for example `extraSettings = list(HTTPPath = "/sql/1.0/warehouses/abcde12345", SSL = 1, ThriftTransport = 2, AuthMech = 3)`

Snowflake:

- `connectionString`. The connection string (e.g. starting with 'jdbc:snowflake://host:port/?db=database').
- `user`. The user name used to access the server.
- `password`. The password for that user.

InterSystems IRIS:

- `connectionString`. The connection string (e.g. starting with 'jdbc:IRIS://host:port/namespace'). Alternatively, you can provide values for `server` and `port`, in which case the default `USER` namespace is used to connect.
- `user`. The user name used to access the server.
- `password`. The password for that user.
- `pathToDriver`. The path to the folder containing the InterSystems IRIS JDBC driver JAR file.

**Windows authentication for SQL Server::**
To be able to use Windows authentication for SQL Server (and PDW), you have to install the JDBC driver. Download the version 9.2.0 .zip from Microsoft and extract its contents to a folder. In the extracted folder you will find the file sqljdbc_9.2/enu/auth/x64/mssql-jdbc_auth-9.2.0.x64.dll (64-bits) or ssqljdbc_9.2/enu/auth/x86/mssql-jdbc_auth-9.2.0.x86.dll (32-bits), which needs to be moved to location on the system path, for example to c:/windows/system32. If you not have write access to any folder in the system path, you can also specify the path to the folder containing the dll by setting the environmental variable PATH_TO_AUTH_DLL, so for example `Sys.setenv("PATH_TO_AUTH_DLL" = "c:/temp")` Note that the environmental variable needs to be set before calling `connect()` for the first time.

## Arguments

`connectionDetails`
: An object of class `connectionDetails` as created by the `createConnectionDetails()` function.

`dbms` The type of DBMS running on the server. Valid values are

- "oracle" for Oracle
- "postgresql" for PostgreSQL
- "redshift" for Amazon Redshift
- "sql server" for Microsoft SQL Server
- "pdw" for Microsoft Parallel Data Warehouse (PDW)
- "netezza" for IBM Netezza
- "bigquery" for Google BigQuery
- "sqlite" for SQLite
- "sqlite extended" for SQLite with extended types (DATE and DATETIME)
- "spark" for Spark
- "snowflake" for Snowflake
- "iris" for InterSystems IRIS

`user` The user name used to access the server.

`password` The password for that user.

`server` The name of the server.

`port` (optional) The port on the server to connect to.

`extraSettings` (optional) Additional configuration settings specific to the database provider to configure things as security for SSL. For connections using JDBC these will be appended to end of the connection string. For connections using DBI, these settings will additionally be used to call `DBI::dbConnect()`.

`oracleDriver` Specify which Oracle drive you want to use. Choose between `"thin"` or `"oci"`.

`connectionString` The JDBC connection string. If specified, the `server`, `port`, `extraSettings`, and `oracleDriver` fields are ignored. If `user` and `password` are not specified, they are assumed to already be included in the connection string.

`pathToDriver` Path to a folder containing the JDBC driver JAR files. See `downloadJdbcDrivers()` for instructions on how to download the relevant drivers.

## Details

This function creates a connection to a database.

## Value

An object that extends `DBIConnection` in a database-specific manner. This object is used to direct commands to the database engine.

## Examples

```
## Not run:
connectionDetails <- createConnectionDetails(
  dbms = "postgresql",
  server = "localhost/postgres",
  user = "root",
```

```
  password = "xxx"
)
conn <- connect(connectionDetails)
dbGetQuery(conn, "SELECT COUNT(*) FROM person")
disconnect(conn)

conn <- connect(dbms = "sql server", server = "RNDUSRDHIT06.jnj.com")
dbGetQuery(conn, "SELECT COUNT(*) FROM concept")
disconnect(conn)

conn <- connect(
  dbms = "oracle",
  server = "127.0.0.1/xe",
  user = "system",
  password = "xxx",
  pathToDriver = "c:/temp"
)
dbGetQuery(conn, "SELECT COUNT(*) FROM test_table")
disconnect(conn)

conn <- connect(
  dbms = "postgresql",
  connectionString = "jdbc:postgresql://127.0.0.1:5432/cmd_database"
)
dbGetQuery(conn, "SELECT COUNT(*) FROM person")
disconnect(conn)

## End(Not run)
```

---

`createConnectionDetails` *createConnectionDetails*

---

## Description

Creates a list containing all details needed to connect to a database. There are three ways to call this function:

- `createConnectionDetails(dbms, user, password, server, port, extraSettings, oracleDriver, pathToDriver)`
- `createConnectionDetails(dbms, connectionString, pathToDriver)`
- `createConnectionDetails(dbms, connectionString, user, password, pathToDriver)`

**DBMS parameter details::**

Depending on the DBMS, the function arguments have slightly different interpretations:

Oracle:

- `user`. The user name used to access the server
- `password`. The password for that user

- `server`. This field contains the SID, or host and servicename, SID, or TNSName: 'sid', 'host/sid', 'host/service name', or 'tnsname'
- `port`. Specifies the port on the server (default = 1521)
- `extraSettings`. The configuration settings for the connection (i.e. SSL Settings such as "(PROTOCOL=tcps)")
- `oracleDriver`. The driver to be used. Choose between "thin" or "oci".
- `pathToDriver`. The path to the folder containing the Oracle JDBC driver JAR files.

Microsoft SQL Server:

- `user`. The user used to log in to the server. If the user is not specified, Windows Integrated Security will be used, which requires the SQL Server JDBC drivers to be installed (see details below).
- `password`. The password used to log on to the server
- `server`. This field contains the host name of the server
- `port`. Not used for SQL Server
- `extraSettings`. The configuration settings for the connection (i.e. SSL Settings such as "encrypt=true; trustServerCertificate=false;")
- `pathToDriver`. The path to the folder containing the SQL Server JDBC driver JAR files.

Microsoft PDW:

- `user`. The user used to log in to the server. If the user is not specified, Windows Integrated Security will be used, which requires the SQL Server JDBC drivers to be installed (see details below).
- `password`. The password used to log on to the server
- `server`. This field contains the host name of the server
- `port`. Not used for SQL Server
- `extraSettings`. The configuration settings for the connection (i.e. SSL Settings such as "encrypt=true; trustServerCertificate=false;")
- `pathToDriver`. The path to the folder containing the SQL Server JDBC driver JAR files.

PostgreSQL:

- `user`. The user used to log in to the server
- `password`. The password used to log on to the server
- `server`. This field contains the host name of the server and the database holding the relevant schemas: host/database
- `port`. Specifies the port on the server (default = 5432)
- `extraSettings`. The configuration settings for the connection (i.e. SSL Settings such as "ssl=true")
- `pathToDriver`. The path to the folder containing the PostgreSQL JDBC driver JAR files.

Redshift:

- `user`. The user used to log in to the server
- `password`. The password used to log on to the server
- `server`. This field contains the host name of the server and the database holding the relevant schemas: host/database
- `port`. Specifies the port on the server (default = 5439)

- ‘extraSettings The configuration settings for the connection (i.e. SSL Settings such as "ssl=true&sslfactory=com.amazo
- `pathToDriver`. The path to the folder containing the RedShift JDBC driver JAR files.

Netezza:

- `user`. The user used to log in to the server
- `password`. The password used to log on to the server
- `server`. This field contains the host name of the server and the database holding the relevant schemas: host/database
- `port`. Specifies the port on the server (default = 5480)
- `extraSettings`. The configuration settings for the connection (i.e. SSL Settings such as "ssl=true")
- `pathToDriver`. The path to the folder containing the Netezza JDBC driver JAR file (nzjdbc.jar).

Impala:

- `user`. The user name used to access the server
- `password`. The password for that user
- `server`. The host name of the server
- `port`. Specifies the port on the server (default = 21050)
- `extraSettings`. The configuration settings for the connection (i.e. SSL Settings such as "SSLKeyStorePwd=*****")
- `pathToDriver`. The path to the folder containing the Impala JDBC driver JAR files.

SQLite:

- `server`. The path to the SQLIte file.

DuckDB:

- `server`. The path to the DuckDB file.
- `extraSettings`. Additional settings for DuckDB. For DuckDB specifically, if `extraSettings$config` is provided, it will be passed to the `duckdb::duckdb()` constructor. For example: `extraSettings = list(config = list(memory_limit = "8GB", preserve_insertion_order = "false"))`

Spark / Databricks:

Currently both JDBC and ODBC connections are supported for Spark. Set the `connectionString` argument to use JDBC, otherwise ODBC is used:

- `connectionString`. The JDBC connection string (e.g. something like ’jdbc:databricks://my-org.cloud.databricks.com:443/default;transportMode=http;ssl=1;AuthMech=3;httpPath=/sql/1.0/warehouses/abcde12
- `user`. The user name used to access the server. This can be set to ’token’ when using a personal token (recommended).
- `password`. The password for that user. This should be your personal token when using a personal token (recommended).
- `server`. The host name of the server (when using ODBC), e.g. ’my-org.cloud.databricks.com’)
- `port`. Specifies the port on the server (when using ODBC)
- `extraSettings`. Additional settings for the ODBC connection, for example `extraSettings = list(HTTPPath = "/sql/1.0/warehouses/abcde12345", SSL = 1, ThriftTransport = 2, AuthMech = 3)`

Snowflake:

- `connectionString`. The connection string (e.g. starting with ’jdbc:snowflake://host:port/?db=database’).

- `user`. The user name used to access the server.
- `password`. The password for that user.

InterSystems IRIS:

- `connectionString`. The connection string (e.g. starting with 'jdbc:IRIS://host:port/namespace'). Alternatively, you can provide values for `server` and `port`, in which case the default `USER` namespace is used to connect.
- `user`. The user name used to access the server.
- `password`. The password for that user.
- `pathToDriver`. The path to the folder containing the InterSystems IRIS JDBC driver JAR file.

**Windows authentication for SQL Server::**

To be able to use Windows authentication for SQL Server (and PDW), you have to install the JDBC driver. Download the version 9.2.0 .zip from Microsoft and extract its contents to a folder. In the extracted folder you will find the file sqljdbc_9.2/enu/auth/x64/mssql-jdbc_auth-9.2.0.x64.dll (64-bits) or ssqljdbc_9.2/enu/auth/x86/mssql-jdbc_auth-9.2.0.x86.dll (32-bits), which needs to be moved to location on the system path, for example to c:/windows/system32. If you not have write access to any folder in the system path, you can also specify the path to the folder containing the dll by setting the environmental variable PATH_TO_AUTH_DLL, so for example `Sys.setenv("PATH_TO_AUTH_DLL" = "c:/temp")` Note that the environmental variable needs to be set before calling `connect()` for the first time.

## Arguments

| | |
|---|---|
| `dbms` | The type of DBMS running on the server. Valid values are<br>• "oracle" for Oracle<br>• "postgresql" for PostgreSQL<br>• "redshift" for Amazon Redshift<br>• "sql server" for Microsoft SQL Server<br>• "pdw" for Microsoft Parallel Data Warehouse (PDW)<br>• "netezza" for IBM Netezza<br>• "bigquery" for Google BigQuery<br>• "sqlite" for SQLite<br>• "sqlite extended" for SQLite with extended types (DATE and DATETIME)<br>• "spark" for Spark<br>• "snowflake" for Snowflake<br>• "iris" for InterSystems IRIS |
| `user` | The user name used to access the server. |
| `password` | The password for that user. |
| `server` | The name of the server. |
| `port` | (optional) The port on the server to connect to. |
| `extraSettings` | (optional) Additional configuration settings specific to the database provider to configure things as security for SSL. For connections using JDBC these will be appended to end of the connection string. For connections using DBI, these settings will additionally be used to call `DBI::dbConnect()`. |

`oracleDriver` Specify which Oracle drive you want to use. Choose between `"thin"` or `"oci"`.

`connectionString`
The JDBC connection string. If specified, the `server`, `port`, `extraSettings`, and `oracleDriver` fields are ignored. If `user` and `password` are not specified, they are assumed to already be included in the connection string.

`pathToDriver` Path to a folder containing the JDBC driver JAR files. See `downloadJdbcDrivers()` for instructions on how to download the relevant drivers.

## Details

This function creates a list containing all details needed to connect to a database. The list can then be used in the `connect()` function.

It is highly recommended to use a secure approach to storing credentials, so not to have your credentials in plain text in your R scripts. The examples demonstrate how to use the `keyring` package.

## Value

A list with all the details needed to connect to a database.

## Examples

```
## Not run:
# Needs to be done only once on a machine. Credentials will then be stored in
# the operating system's secure credential manager:
keyring::key_set_with_value("server", password = "localhost/postgres")
keyring::key_set_with_value("user", password = "root")
keyring::key_set_with_value("password", password = "secret")

# Create connection details using keyring. Note: the connection details will
# not store the credentials themselves, but the reference to get the credentials.
connectionDetails <- createConnectionDetails(
  dbms = "postgresql",
  server = keyring::key_get("server"),
  user = keyring::key_get("user"),
  password = keyring::key_get("password"),
)
conn <- connect(connectionDetails)
dbGetQuery(conn, "SELECT COUNT(*) FROM person")
disconnect(conn)

## End(Not run)
```

---

`createDbiConnectionDetails`
*Create DBI connection details*

---

### Description

For advanced users only. This function will allow `DatabaseConnector` to wrap any DBI driver. Using a driver that `DatabaseConnector` hasn't been tested with may give unpredictable performance. Use at your own risk. No support will be provided.

### Usage

```
createDbiConnectionDetails(dbms, drv, ...)
```

### Arguments

| | |
|---|---|
| `dbms` | The type of DBMS running on the server. Valid values are<br>• "oracle" for Oracle<br>• "postgresql" for PostgreSQL<br>• "redshift" for Amazon Redshift<br>• "sql server" for Microsoft SQL Server<br>• "pdw" for Microsoft Parallel Data Warehouse (PDW)<br>• "netezza" for IBM Netezza<br>• "bigquery" for Google BigQuery<br>• "sqlite" for SQLite<br>• "sqlite extended" for SQLite with extended types (DATE and DATETIME)<br>• "spark" for Spark<br>• "snowflake" for Snowflake<br>• "iris" for InterSystems IRIS |
| `drv` | An object that inherits from DBIDriver, or an existing DBIConnection object (in order to clone an existing connection). |
| `...` | authentication arguments needed by the DBMS instance; these typically include user, password, host, port, dbname, etc. For details see the appropriate DBIDriver |

### Value

A list with all the details needed to connect to a database.

---

## `createZipFile` *Compress files and/or folders into a single zip file*

---

### Description

Compress files and/or folders into a single zip file

### Usage

```
createZipFile(zipFile, files, rootFolder = getwd(), compressionLevel = 9)
```

### Arguments

| | |
|---|---|
| `zipFile` | The path to the zip file to be created. |
| `files` | The files and/or folders to be included in the zip file. Folders will be included recursively. |
| `rootFolder` | The root folder. All files will be stored with relative paths relative to this folder. |
| `compressionLevel` | A number between 1 and 9. 9 compresses best, but it also takes the longest. |

### Details

Uses Java's compression library to create a zip file. It is similar to `utils::zip`, except that it does not require an external zip tool to be available on the system path.

---

## `DatabaseConnectorDriver` *Create a DatabaseConnectorDriver object*

---

### Description

Create a DatabaseConnectorDriver object

### Usage

```
DatabaseConnectorDriver()
```

---

## `dbAppendTable,DatabaseConnectorConnection,character-method` *Insert rows into a table*

---

### Description

The `dbAppendTable()` method assumes that the table has been created beforehand, e.g. with `dbCreateTable()`. The default implementation calls `sqlAppendTableTemplate()` and then `dbExecute()` with the `param` argument. Use `dbAppendTableArrow()` to append data from an Arrow stream.

### Usage

```
## S4 method for signature 'DatabaseConnectorConnection,character'
dbAppendTable(
  conn,
  name,
  value,
  databaseSchema = NULL,
  temporary = FALSE,
  ...,
  row.names = NULL
)
```

## Arguments

| | |
|---|---|
| conn | A DBIConnection object, as returned by dbConnect(). |
| name | The table name, passed on to dbQuoteIdentifier(). Options are:<br>• a character string with the unquoted DBMS table name, e.g. "table_name",<br>• a call to Id() with components to the fully qualified table name, e.g. Id(schema = "my_schema", table = "table_name")<br>• a call to SQL() with the quoted and fully qualified table name given verbatim, e.g. SQL('"my_schema"."table_name"') |
| value | A data.frame (or coercible to data.frame). |
| databaseSchema | The name of the database schema. See details for platform-specific details. |
| temporary | Should the table created as a temp table? |
| ... | Other parameters passed on to methods. |
| row.names | Must be NULL. |

## Details

The databaseSchema argument is interpreted differently according to the different platforms: SQL Server and PDW: The databaseSchema schema should specify both the database and the schema, e.g. 'my_database.dbo'. Impala: the databaseSchema should specify the database. Oracle: The databaseSchema should specify the Oracle 'user'. All other : The databaseSchema should specify the schema.

## Value

dbAppendTable() returns a scalar numeric.

## See Also

Other DBIConnection generics: DBIConnection-class, dbAppendTableArrow(), dbCreateTable(), dbCreateTableArrow(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbGetQueryArrow(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbQuoteIdentifier(), dbReadTable(), dbReadTableArrow(), dbRemoveTable(), dbSendQuery(), dbSendQueryArrow(), dbSendStatement(), dbUnquoteIdentifier(), dbWriteTable(), dbWriteTableArrow()

---

# dbClearResult,DatabaseConnectorDbiResult-method

*Clear a result set*

---

## Description

Frees all resources (local and remote) associated with a result set. This step is mandatory for all objects obtained by calling dbSendQuery() or dbSendStatement().

### Usage

```
## S4 method for signature 'DatabaseConnectorDbiResult'
dbClearResult(res, ...)
```

### Arguments

| | |
|---|---|
| res | An object inheriting from DBIResult. |
| ... | Other arguments passed on to methods. |

### Value

dbClearResult() returns TRUE, invisibly, for result sets obtained from dbSendQuery(), dbSendStatement(), or dbSendQueryArrow(),

### See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteLiteral(), dbQuoteString()

Other DBIResultArrow generics: DBIResultArrow-class, dbBind(), dbFetchArrow(), dbFetchArrowChunk(), dbHasCompleted(), dbIsValid()

Other data retrieval generics: dbBind(), dbFetch(), dbFetchArrow(), dbFetchArrowChunk(), dbGetQuery(), dbGetQueryArrow(), dbHasCompleted(), dbSendQuery(), dbSendQueryArrow()

Other command execution generics: dbBind(), dbExecute(), dbGetRowsAffected(), dbSendStatement()

---

## dbClearResult,DatabaseConnectorJdbcResult-method

*Clear a result set*

---

### Description

Frees all resources (local and remote) associated with a result set. This step is mandatory for all objects obtained by calling dbSendQuery() or dbSendStatement().

### Usage

```
## S4 method for signature 'DatabaseConnectorJdbcResult'
dbClearResult(res, ...)
```

### Arguments

| | |
|---|---|
| res | An object inheriting from DBIResult. |
| ... | Other arguments passed on to methods. |

## Value

dbClearResult() returns TRUE, invisibly, for result sets obtained from dbSendQuery(), dbSendStatement(), or dbSendQueryArrow(),

## See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteLiteral(), dbQuoteString()

Other DBIResultArrow generics: DBIResultArrow-class, dbBind(), dbFetchArrow(), dbFetchArrowChunk(), dbHasCompleted(), dbIsValid()

Other data retrieval generics: dbBind(), dbFetch(), dbFetchArrow(), dbFetchArrowChunk(), dbGetQuery(), dbGetQueryArrow(), dbHasCompleted(), dbSendQuery(), dbSendQueryArrow()

Other command execution generics: dbBind(), dbExecute(), dbGetRowsAffected(), dbSendStatement()

---

# dbColumnInfo,DatabaseConnectorDbiResult-method

*Information about result types*

---

## Description

Produces a data.frame that describes the output of a query. The data.frame should have as many rows as there are output fields in the result set, and each column in the data.frame describes an aspect of the result set field (field name, type, etc.)

## Usage

```
## S4 method for signature 'DatabaseConnectorDbiResult'
dbColumnInfo(res, ...)
```

## Arguments

| | |
|---|---|
| res | An object inheriting from DBIResult. |
| ... | Other arguments passed on to methods. |

## Value

dbColumnInfo() returns a data frame with at least two columns "name" and "type" (in that order) (and optional columns that start with a dot). The "name" and "type" columns contain the names and types of the R columns of the data frame that is returned from dbFetch(). The "type" column is of type character and only for information. Do not compute on the "type" column, instead use dbFetch(res, n = 0) to create a zero-row data frame initialized with the correct data types.

## See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteLiteral(), dbQuoteString()

---

# dbColumnInfo,DatabaseConnectorJdbcResult-method

*Information about result types*

---

## Description

Produces a data.frame that describes the output of a query. The data.frame should have as many rows as there are output fields in the result set, and each column in the data.frame describes an aspect of the result set field (field name, type, etc.)

## Usage

```
## S4 method for signature 'DatabaseConnectorJdbcResult'
dbColumnInfo(res, ...)
```

## Arguments

| | |
|---|---|
| res | An object inheriting from DBIResult. |
| ... | Other arguments passed on to methods. |

## Value

dbColumnInfo() returns a data frame with at least two columns "name" and "type" (in that order) (and optional columns that start with a dot). The "name" and "type" columns contain the names and types of the R columns of the data frame that is returned from dbFetch(). The "type" column is of type character and only for information. Do not compute on the "type" column, instead use dbFetch(res, n = 0) to create a zero-row data frame initialized with the correct data types.

## See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteLiteral(), dbQuoteString()

dbConnect,DatabaseConnectorDriver-method
*Create a connection to a DBMS*

## Description

Connect to a database. This function is synonymous with the `connect()` function. except a dummy driver needs to be specified

## Usage

```
## S4 method for signature 'DatabaseConnectorDriver'
dbConnect(drv, ...)
```

## Arguments

| | |
|---|---|
| `drv` | The result of the `DatabaseConnectorDriver()` function |
| `...` | Other parameters. These are the same as expected by the `connect()` function. |

## Value

Returns a DatabaseConnectorConnection object that can be used with most of the other functions in this package.

## Examples

```
## Not run:
conn <- dbConnect(DatabaseConnectorDriver(),
  dbms = "postgresql",
  server = "localhost/ohdsi",
  user = "joe",
  password = "secret"
)
querySql(conn, "SELECT * FROM cdm_synpuf.person;")
dbDisconnect(conn)

## End(Not run)
```

dbCreateTable,DatabaseConnectorConnection-method
*Create a table in the database*

## Description

The default dbCreateTable() method calls sqlCreateTable() and dbExecute(). Use dbCreateTableArrow() to create a table from an Arrow schema.

## Usage

```
## S4 method for signature 'DatabaseConnectorConnection'
dbCreateTable(
  conn,
  name,
  fields,
  databaseSchema = NULL,
  ...,
  row.names = NULL,
  temporary = FALSE
)
```

## Arguments

| | |
|---|---|
| conn | A DBIConnection object, as returned by dbConnect(). |
| name | The table name, passed on to dbQuoteIdentifier(). Options are:<br>• a character string with the unquoted DBMS table name, e.g. "table_name",<br>• a call to Id() with components to the fully qualified table name, e.g. Id(schema = "my_schema", table = "table_name")<br>• a call to SQL() with the quoted and fully qualified table name given verbatim, e.g. SQL('"my_schema"."table_name"') |
| fields | Either a character vector or a data frame.<br>A named character vector: Names are column names, values are types. Names are escaped with dbQuoteIdentifier(). Field types are unescaped.<br>A data frame: field types are generated using dbDataType(). |
| databaseSchema | The name of the database schema. See details for platform-specific details. |
| ... | Other parameters passed on to methods. |
| row.names | Must be NULL. |
| temporary | Should the table created as a temp table? |

### Details

The databaseSchema argument is interpreted differently according to the different platforms: SQL Server and PDW: The databaseSchema schema should specify both the database and the schema, e.g. 'my_database.dbo'. Impala: the databaseSchema should specify the database. Oracle: The databaseSchema should specify the Oracle 'user'. All other : The databaseSchema should specify the schema.

### Value

`dbCreateTable()` returns `TRUE`, invisibly.

### See Also

Other DBIConnection generics: `DBIConnection-class`, `dbAppendTable()`, `dbAppendTableArrow()`, `dbCreateTableArrow()`, `dbDataType()`, `dbDisconnect()`, `dbExecute()`, `dbExistsTable()`, `dbGetException()`, `dbGetInfo()`, `dbGetQuery()`, `dbGetQueryArrow()`, `dbIsReadOnly()`, `dbIsValid()`, `dbListFields()`, `dbListObjects()`, `dbListResults()`, `dbListTables()`, `dbQuoteIdentifier()`, `dbReadTable()`, `dbReadTableArrow()`, `dbRemoveTable()`, `dbSendQuery()`, `dbSendQueryArrow()`, `dbSendStatement()`, `dbUnquoteIdentifier()`, `dbWriteTable()`, `dbWriteTableArrow()`

---

## dbDisconnect,DatabaseConnectorConnection-method

*Disconnect (close) a connection*

---

### Description

This closes the connection, discards all pending work, and frees resources (e.g., memory, sockets).

### Usage

```
## S4 method for signature 'DatabaseConnectorConnection'
dbDisconnect(conn)
```

### Arguments

| | |
|---|---|
| conn | A DBIConnection object, as returned by `dbConnect()`. |

### Value

`dbDisconnect()` returns `TRUE`, invisibly.

## See Also

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbAppendTableArrow(), dbCreateTable(), dbCreateTableArrow(), dbDataType(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbGetQueryArrow(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbQuoteIdentifier(), dbReadTable(), dbReadTableArrow(), dbRemoveTable(), dbSendQuery(), dbSendQueryArrow(), dbSendStatement(), dbUnquoteIdentifier(), dbWriteTable(), dbWriteTableArrow()

---

# dbExecute,DatabaseConnectorDbiConnection,character-method

*Change database state*

---

## Description

Executes a statement and returns the number of rows affected. dbExecute() comes with a default implementation (which should work with most backends) that calls dbSendStatement(), then dbGetRowsAffected(), ensuring that the result is always freed by dbClearResult(). For passing query parameters, see dbBind(), in particular the "The command execution flow" section.

## Usage

```
## S4 method for signature 'DatabaseConnectorDbiConnection,character'
dbExecute(conn, statement, ...)
```

## Arguments

| | |
|---|---|
| conn | A DBIConnection object, as returned by dbConnect(). |
| statement | a character string containing SQL. |
| ... | Other parameters passed on to methods. |

## Details

You can also use dbExecute() to call a stored procedure that performs data manipulation or other actions that do not return a result set. To execute a stored procedure that returns a result set, or a data manipulation query that also returns a result set such as INSERT INTO ... RETURNING ..., use dbGetQuery() instead.

## Value

dbExecute() always returns a scalar numeric that specifies the number of rows affected by the statement.

## See Also

For queries: dbSendQuery() and dbGetQuery().

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbAppendTableArrow(), dbCreateTable(), dbCreateTableArrow(), dbDataType(), dbDisconnect(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbGetQueryArrow(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbQuoteIdentifier(), dbReadTable(), dbReadTableArrow(), dbRemoveTable(), dbSendQuery(), dbSendQueryArrow(), dbSendStatement(), dbUnquoteIdentifier(), dbWriteTable(), dbWriteTableArrow()

Other command execution generics: dbBind(), dbClearResult(), dbGetRowsAffected(), dbSendStatement()

---

# dbExecute,DatabaseConnectorJdbcConnection,character-method

*Change database state*

---

## Description

Executes a statement and returns the number of rows affected. dbExecute() comes with a default implementation (which should work with most backends) that calls dbSendStatement(), then dbGetRowsAffected(), ensuring that the result is always freed by dbClearResult(). For passing query parameters, see dbBind(), in particular the "The command execution flow" section.

## Usage

```
## S4 method for signature 'DatabaseConnectorJdbcConnection,character'
dbExecute(conn, statement, ...)
```

## Arguments

| | |
|---|---|
| conn | A DBIConnection object, as returned by dbConnect(). |
| statement | a character string containing SQL. |
| ... | Other parameters passed on to methods. |

## Details

You can also use dbExecute() to call a stored procedure that performs data manipulation or other actions that do not return a result set. To execute a stored procedure that returns a result set, or a data manipulation query that also returns a result set such as INSERT INTO ... RETURNING ..., use dbGetQuery() instead.

## Value

dbExecute() always returns a scalar numeric that specifies the number of rows affected by the statement.

## See Also

For queries: dbSendQuery() and dbGetQuery().

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbAppendTableArrow(), dbCreateTable(), dbCreateTableArrow(), dbDataType(), dbDisconnect(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbGetQueryArrow(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbQuoteIdentifier(), dbReadTable(), dbReadTableArrow(), dbRemoveTable(), dbSendQuery(), dbSendQueryArrow(), dbSendStatement(), dbUnquoteIdentifier(), dbWriteTable(), dbWriteTableArrow()

Other command execution generics: dbBind(), dbClearResult(), dbGetRowsAffected(), dbSendStatement()

---

# dbExistsTable,DatabaseConnectorConnection,character-method

*Does a table exist?*

---

## Description

Returns if a table given by name exists in the database.

## Usage

```
## S4 method for signature 'DatabaseConnectorConnection,character'
dbExistsTable(conn, name, databaseSchema = NULL, ...)
```

## Arguments

| | |
|---|---|
| conn | A DBIConnection object, as returned by dbConnect(). |
| name | The table name, passed on to dbQuoteIdentifier(). Options are:<br>• a character string with the unquoted DBMS table name, e.g. "table_name",<br>• a call to Id() with components to the fully qualified table name, e.g. Id(schema = "my_schema", table = "table_name")<br>• a call to SQL() with the quoted and fully qualified table name given verbatim, e.g. SQL('"my_schema"."table_name"') |
| databaseSchema | The name of the database schema. See details for platform-specific details. |
| ... | Other parameters passed on to methods. |

## Details

The databaseSchema argument is interpreted differently according to the different platforms: SQL Server and PDW: The databaseSchema schema should specify both the database and the schema, e.g. 'my_database.dbo'. Impala: the databaseSchema should specify the database. Oracle: The databaseSchema should specify the Oracle 'user'. All other : The databaseSchema should specify the schema.

## Value

dbExistsTable() returns a logical scalar, TRUE if the table or view specified by the name argument exists, FALSE otherwise.

This includes temporary tables if supported by the database.

## See Also

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbAppendTableArrow(), dbCreateTable(), dbCreateTableArrow(), dbDataType(), dbDisconnect(), dbExecute(), dbGetException(), dbGetInfo(), dbGetQuery(), dbGetQueryArrow(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbQuoteIdentifier(), dbReadTable(), dbReadTableArrow(), dbRemoveTable(), dbSendQuery(), dbSendQueryArrow(), dbSendStatement(), dbUnquoteIdentifier(), dbWriteTable(), dbWriteTableArrow()

---

# dbFetch,DatabaseConnectorDbiResult-method

*Fetch records from a previously executed query*

---

## Description

Fetch the next n elements (rows) from the result set and return them as a data.frame.

## Usage

```
## S4 method for signature 'DatabaseConnectorDbiResult'
dbFetch(res, n = -1, ...)
```

## Arguments

| | |
|---|---|
| res | An object inheriting from DBIResult, created by dbSendQuery(). |
| n | maximum number of records to retrieve per fetch. Use n = -1 or n = Inf to retrieve all pending records. Some implementations may recognize other special values. |
| ... | Other arguments passed on to methods. |

## Details

fetch() is provided for compatibility with older DBI clients - for all new code you are strongly encouraged to use dbFetch(). The default implementation for dbFetch() calls fetch() so that it is compatible with existing code. Modern backends should implement for dbFetch() only.

## Value

dbFetch() always returns a data.frame with as many rows as records were fetched and as many columns as fields in the result set, even if the result is a single value or has one or zero rows. Passing n = NA is supported and returns an arbitrary number of rows (at least one) as specified by the driver, but at most the remaining rows in the result set.

## See Also

Close the result set with dbClearResult() as soon as you finish retrieving the records you want.

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteLiteral(), dbQuoteString()

Other data retrieval generics: dbBind(), dbClearResult(), dbFetchArrow(), dbFetchArrowChunk(), dbGetQuery(), dbGetQueryArrow(), dbHasCompleted(), dbSendQuery(), dbSendQueryArrow()

---

# dbFetch,DatabaseConnectorJdbcResult-method

*Fetch records from a previously executed query*

---

## Description

Fetch the next n elements (rows) from the result set and return them as a data.frame.

## Usage

```
## S4 method for signature 'DatabaseConnectorJdbcResult'
dbFetch(res, n = -1, ...)
```

## Arguments

| | |
|---|---|
| res | An object inheriting from DBIResult, created by dbSendQuery(). |
| n | maximum number of records to retrieve per fetch. Use n = -1 or n = Inf to retrieve all pending records. Some implementations may recognize other special values. |
| ... | Other arguments passed on to methods. |

## Details

fetch() is provided for compatibility with older DBI clients - for all new code you are strongly encouraged to use dbFetch(). The default implementation for dbFetch() calls fetch() so that it is compatible with existing code. Modern backends should implement for dbFetch() only.

## Value

dbFetch() always returns a data.frame with as many rows as records were fetched and as many columns as fields in the result set, even if the result is a single value or has one or zero rows. Passing n = NA is supported and returns an arbitrary number of rows (at least one) as specified by the driver, but at most the remaining rows in the result set.

## See Also

Close the result set with `dbClearResult()` as soon as you finish retrieving the records you want.

Other DBIResult generics: `DBIResult-class`, `dbBind()`, `dbClearResult()`, `dbColumnInfo()`, `dbGetInfo()`, `dbGetRowCount()`, `dbGetRowsAffected()`, `dbGetStatement()`, `dbHasCompleted()`, `dbIsReadOnly()`, `dbIsValid()`, `dbQuoteLiteral()`, `dbQuoteString()`

Other data retrieval generics: `dbBind()`, `dbClearResult()`, `dbFetchArrow()`, `dbFetchArrowChunk()`, `dbGetQuery()`, `dbGetQueryArrow()`, `dbHasCompleted()`, `dbSendQuery()`, `dbSendQueryArrow()`

---

# dbGetInfo,DatabaseConnectorConnection-method

*Get DBMS metadata*

---

## Description

Retrieves information on objects of class DBIDriver, DBIConnection or DBIResult.

## Usage

```
## S4 method for signature 'DatabaseConnectorConnection'
dbGetInfo(dbObj, ...)
```

## Arguments

| | |
|---|---|
| `dbObj` | An object inheriting from DBIObject, i.e. DBIDriver, DBIConnection, or a DBIResult |
| `...` | Other arguments to methods. |

## Value

For objects of class DBIDriver, `dbGetInfo()` returns a named list that contains at least the following components:

- `driver.version`: the package version of the DBI backend,
- `client.version`: the version of the DBMS client library.

For objects of class DBIConnection, `dbGetInfo()` returns a named list that contains at least the following components:

- `db.version`: version of the database server,
- `dbname`: database name,
- `username`: username to connect to the database,
- `host`: hostname of the database server,
- `port`: port on the database server. It must not contain a `password` component. Components that are not applicable should be set to `NA`.

For objects of class DBIResult, dbGetInfo() returns a named list that contains at least the following components:

- statatment: the statement used with dbSendQuery() or dbExecute(), as returned by dbGetStatement(),
- row.count: the number of rows fetched so far (for queries), as returned by dbGetRowCount(),
- rows.affected: the number of rows affected (for statements), as returned by dbGetRowsAffected()
- has.completed: a logical that indicates if the query or statement has completed, as returned by dbHasCompleted().

## See Also

Other DBIDriver generics: DBIDriver-class, dbCanConnect(), dbConnect(), dbDataType(), dbDriver(), dbIsReadOnly(), dbIsValid(), dbListConnections()

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbAppendTableArrow(), dbCreateTable(), dbCreateTableArrow(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetQuery(), dbGetQueryArrow(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbQuoteIdentifier(), dbReadTable(), dbReadTableArrow(), dbRemoveTable(), dbSendQuery(), dbSendQueryArrow(), dbSendStatement(), dbUnquoteIdentifier(), dbWriteTable(), dbWriteTableArrow()

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetRowCount(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteLiteral(), dbQuoteString()

---

# dbGetInfo,DatabaseConnectorDriver-method *Get DBMS metadata*

---

## Description

Retrieves information on objects of class DBIDriver, DBIConnection or DBIResult.

## Usage

```
## S4 method for signature 'DatabaseConnectorDriver'
dbGetInfo(dbObj, ...)
```

## Arguments

| | |
|---|---|
| dbObj | An object inheriting from DBIObject, i.e. DBIDriver, DBIConnection, or a DBIResult |
| ... | Other arguments to methods. |

## Value

For objects of class DBIDriver, `dbGetInfo()` returns a named list that contains at least the following components:

- `driver.version`: the package version of the DBI backend,
- `client.version`: the version of the DBMS client library.

For objects of class DBIConnection, `dbGetInfo()` returns a named list that contains at least the following components:

- `db.version`: version of the database server,
- `dbname`: database name,
- `username`: username to connect to the database,
- `host`: hostname of the database server,
- `port`: port on the database server. It must not contain a `password` component. Components that are not applicable should be set to `NA`.

For objects of class DBIResult, `dbGetInfo()` returns a named list that contains at least the following components:

- `statatment`: the statement used with `dbSendQuery()` or `dbExecute()`, as returned by `dbGetStatement()`,
- `row.count`: the number of rows fetched so far (for queries), as returned by `dbGetRowCount()`,
- `rows.affected`: the number of rows affected (for statements), as returned by `dbGetRowsAffected()`
- `has.completed`: a logical that indicates if the query or statement has completed, as returned by `dbHasCompleted()`.

## See Also

Other DBIDriver generics: `DBIDriver-class`, `dbCanConnect()`, `dbConnect()`, `dbDataType()`, `dbDriver()`, `dbIsReadOnly()`, `dbIsValid()`, `dbListConnections()`

Other DBIConnection generics: `DBIConnection-class`, `dbAppendTable()`, `dbAppendTableArrow()`, `dbCreateTable()`, `dbCreateTableArrow()`, `dbDataType()`, `dbDisconnect()`, `dbExecute()`, `dbExistsTable()`, `dbGetException()`, `dbGetQuery()`, `dbGetQueryArrow()`, `dbIsReadOnly()`, `dbIsValid()`, `dbListFields()`, `dbListObjects()`, `dbListResults()`, `dbListTables()`, `dbQuoteIdentifier()`, `dbReadTable()`, `dbReadTableArrow()`, `dbRemoveTable()`, `dbSendQuery()`, `dbSendQueryArrow()`, `dbSendStatement()`, `dbUnquoteIdentifier()`, `dbWriteTable()`, `dbWriteTableArrow()`

Other DBIResult generics: `DBIResult-class`, `dbBind()`, `dbClearResult()`, `dbColumnInfo()`, `dbFetch()`, `dbGetRowCount()`, `dbGetRowsAffected()`, `dbGetStatement()`, `dbHasCompleted()`, `dbIsReadOnly()`, `dbIsValid()`, `dbQuoteLiteral()`, `dbQuoteString()`

dbGetQuery,DatabaseConnectorDbiConnection,character-method
*Retrieve results from a query*

## Description

Returns the result of a query as a data frame. dbGetQuery() comes with a default implementation (which should work with most backends) that calls dbSendQuery(), then dbFetch(), ensuring that the result is always freed by dbClearResult(). For retrieving chunked/paged results or for passing query parameters, see dbSendQuery(), in particular the "The data retrieval flow" section. For retrieving results as an Arrow object, see dbGetQueryArrow().

## Usage

```
## S4 method for signature 'DatabaseConnectorDbiConnection,character'
dbGetQuery(conn, statement, ...)
```

## Arguments

| | |
|---|---|
| conn | A DBIConnection object, as returned by dbConnect(). |
| statement | a character string containing SQL. |
| ... | Other parameters passed on to methods. |

## Details

This method is for SELECT queries only (incl. other SQL statements that return a SELECT-alike result, e.g., execution of a stored procedure or data manipulation queries like INSERT INTO ... RETURNING ...). To execute a stored procedure that does not return a result set, use dbExecute().

Some backends may support data manipulation statements through this method for compatibility reasons. However, callers are strongly advised to use dbExecute() for data manipulation statements.

## Value

dbGetQuery() always returns a data.frame, with as many rows as records were fetched and as many columns as fields in the result set, even if the result is a single value or has one or zero rows.

## See Also

For updates: dbSendStatement() and dbExecute().

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbAppendTableArrow(), dbCreateTable(), dbCreateTableArrow(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQueryArrow(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbQuoteIdentifier(), dbReadTable(), dbReadTableArrow(), dbRemoveTable(), dbSendQuery(), dbSendQueryArrow(), dbSendStatement(), dbUnquoteIdentifier(), dbWriteTable(), dbWriteTableArrow()

Other data retrieval generics: dbBind(), dbClearResult(), dbFetch(), dbFetchArrow(), dbFetchArrowChunk(), dbGetQueryArrow(), dbHasCompleted(), dbSendQuery(), dbSendQueryArrow()

---

dbGetQuery,DatabaseConnectorJdbcConnection,character-method
*Retrieve results from a query*

---

## Description

Returns the result of a query as a data frame. dbGetQuery() comes with a default implementation (which should work with most backends) that calls dbSendQuery(), then dbFetch(), ensuring that the result is always freed by dbClearResult(). For retrieving chunked/paged results or for passing query parameters, see dbSendQuery(), in particular the "The data retrieval flow" section. For retrieving results as an Arrow object, see dbGetQueryArrow().

## Usage

```
## S4 method for signature 'DatabaseConnectorJdbcConnection,character'
dbGetQuery(conn, statement, ...)
```

## Arguments

| | |
|---|---|
| conn | A DBIConnection object, as returned by dbConnect(). |
| statement | a character string containing SQL. |
| ... | Other parameters passed on to methods. |

## Details

This method is for SELECT queries only (incl. other SQL statements that return a SELECT-alike result, e.g., execution of a stored procedure or data manipulation queries like INSERT INTO ... RETURNING ...). To execute a stored procedure that does not return a result set, use dbExecute().

Some backends may support data manipulation statements through this method for compatibility reasons. However, callers are strongly advised to use dbExecute() for data manipulation statements.

## Value

dbGetQuery() always returns a data.frame, with as many rows as records were fetched and as many columns as fields in the result set, even if the result is a single value or has one or zero rows.

## See Also

For updates: dbSendStatement() and dbExecute().

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbAppendTableArrow(), dbCreateTable(), dbCreateTableArrow(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQueryArrow(), dbIsReadOnly(), dbIsValid(), dbListFields(),

dbListObjects(), dbListResults(), dbListTables(), dbQuoteIdentifier(), dbReadTable(), dbReadTableArrow(), dbRemoveTable(), dbSendQuery(), dbSendQueryArrow(), dbSendStatement(), dbUnquoteIdentifier(), dbWriteTable(), dbWriteTableArrow()

Other data retrieval generics: dbBind(), dbClearResult(), dbFetch(), dbFetchArrow(), dbFetchArrowChunk(), dbGetQueryArrow(), dbHasCompleted(), dbSendQuery(), dbSendQueryArrow()

---

# dbGetRowCount,DatabaseConnectorDbiResult-method

*The number of rows fetched so far*

---

## Description

Returns the total number of rows actually fetched with calls to dbFetch() for this result set.

## Usage

```
## S4 method for signature 'DatabaseConnectorDbiResult'
dbGetRowCount(res, ...)
```

## Arguments

| | |
|---|---|
| res | An object inheriting from DBIResult. |
| ... | Other arguments passed on to methods. |

## Value

dbGetRowCount() returns a scalar number (integer or numeric), the number of rows fetched so far. After calling dbSendQuery(), the row count is initially zero. After a call to dbFetch() without limit, the row count matches the total number of rows returned. Fetching a limited number of rows increases the number of rows by the number of rows returned, even if fetching past the end of the result set. For queries with an empty result set, zero is returned even after fetching. For data manipulation statements issued with dbSendStatement(), zero is returned before and after calling dbFetch().

## See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteLiteral(), dbQuoteString()

---

dbGetRowCount,DatabaseConnectorJdbcResult-method
*The number of rows fetched so far*

---

## Description

Returns the total number of rows actually fetched with calls to dbFetch() for this result set.

## Usage

```
## S4 method for signature 'DatabaseConnectorJdbcResult'
dbGetRowCount(res, ...)
```

## Arguments

| | |
|---|---|
| res | An object inheriting from DBIResult. |
| ... | Other arguments passed on to methods. |

## Value

dbGetRowCount() returns a scalar number (integer or numeric), the number of rows fetched so far. After calling dbSendQuery(), the row count is initially zero. After a call to dbFetch() without limit, the row count matches the total number of rows returned. Fetching a limited number of rows increases the number of rows by the number of rows returned, even if fetching past the end of the result set. For queries with an empty result set, zero is returned even after fetching. For data manipulation statements issued with dbSendStatement(), zero is returned before and after calling dbFetch().

## See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteLiteral(), dbQuoteString()

---

dbGetRowsAffected,DatabaseConnectorDbiResult-method
*The number of rows affected*

---

## Description

This method returns the number of rows that were added, deleted, or updated by a data manipulation statement.

## Usage

```
## S4 method for signature 'DatabaseConnectorDbiResult'
dbGetRowsAffected(res, ...)
```

## Arguments

| | |
|---|---|
| res | An object inheriting from DBIResult. |
| ... | Other arguments passed on to methods. |

## Value

dbGetRowsAffected() returns a scalar number (integer or numeric), the number of rows affected by a data manipulation statement issued with dbSendStatement(). The value is available directly after the call and does not change after calling dbFetch(). NA_integer_ or NA_numeric_ are allowed if the number of rows affected is not known.

For queries issued with dbSendQuery(), zero is returned before and after the call to dbFetch(). NA values are not allowed.

## See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteLiteral(), dbQuoteString()

Other command execution generics: dbBind(), dbClearResult(), dbExecute(), dbSendStatement()

---

# dbGetRowsAffected,DatabaseConnectorJdbcResult-method

*The number of rows affected*

---

## Description

This method returns the number of rows that were added, deleted, or updated by a data manipulation statement.

## Usage

```
## S4 method for signature 'DatabaseConnectorJdbcResult'
dbGetRowsAffected(res, ...)
```

## Arguments

| | |
|---|---|
| res | An object inheriting from DBIResult. |
| ... | Other arguments passed on to methods. |

## Value

dbGetRowsAffected() returns a scalar number (integer or numeric), the number of rows affected by a data manipulation statement issued with dbSendStatement(). The value is available directly after the call and does not change after calling dbFetch(). NA_integer_ or NA_numeric_ are allowed if the number of rows affected is not known.

For queries issued with dbSendQuery(), zero is returned before and after the call to dbFetch(). NA values are not allowed.

## See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteLiteral(), dbQuoteString()

Other command execution generics: dbBind(), dbClearResult(), dbExecute(), dbSendStatement()

---

# dbGetStatement,DatabaseConnectorDbiResult-method

*Get the statement associated with a result set*

---

## Description

Returns the statement that was passed to dbSendQuery() or dbSendStatement().

## Usage

```
## S4 method for signature 'DatabaseConnectorDbiResult'
dbGetStatement(res, ...)
```

## Arguments

| | |
|---|---|
| res | An object inheriting from DBIResult. |
| ... | Other arguments passed on to methods. |

## Value

dbGetStatement() returns a string, the query used in either dbSendQuery() or dbSendStatement().

## See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteLiteral(), dbQuoteString()

## dbGetStatement,DatabaseConnectorJdbcResult-method

*Get the statement associated with a result set*

### Description

Returns the statement that was passed to dbSendQuery() or dbSendStatement().

### Usage

```
## S4 method for signature 'DatabaseConnectorJdbcResult'
dbGetStatement(res, ...)
```

### Arguments

| | |
|---|---|
| res | An object inheriting from DBIResult. |
| ... | Other arguments passed on to methods. |

### Value

dbGetStatement() returns a string, the query used in either dbSendQuery() or dbSendStatement().

### See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteLiteral(), dbQuoteString()

## dbHasCompleted,DatabaseConnectorDbiResult-method

*Completion status*

### Description

This method returns if the operation has completed. A SELECT query is completed if all rows have been fetched. A data manipulation statement is always completed.

### Usage

```
## S4 method for signature 'DatabaseConnectorDbiResult'
dbHasCompleted(res, ...)
```

### Arguments

| | |
|---|---|
| res | An object inheriting from DBIResult. |
| ... | Other arguments passed on to methods. |

## Value

dbHasCompleted() returns a logical scalar. For a query initiated by dbSendQuery() with non-empty result set, dbHasCompleted() returns FALSE initially and TRUE after calling dbFetch() without limit. For a query initiated by dbSendStatement(), dbHasCompleted() always returns TRUE.

## See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbGetStatement(), dbIsReadOnly(), dbIsValid(), dbQuoteLiteral(), dbQuoteString()

Other DBIResultArrow generics: DBIResultArrow-class, dbBind(), dbClearResult(), dbFetchArrow(), dbFetchArrowChunk(), dbIsValid()

Other data retrieval generics: dbBind(), dbClearResult(), dbFetch(), dbFetchArrow(), dbFetchArrowChunk(), dbGetQuery(), dbGetQueryArrow(), dbSendQuery(), dbSendQueryArrow()

---

# dbHasCompleted,DatabaseConnectorJdbcResult-method

*Completion status*

---

## Description

This method returns if the operation has completed. A SELECT query is completed if all rows have been fetched. A data manipulation statement is always completed.

## Usage

```
## S4 method for signature 'DatabaseConnectorJdbcResult'
dbHasCompleted(res, ...)
```

## Arguments

| | |
|---|---|
| res | An object inheriting from DBIResult. |
| ... | Other arguments passed on to methods. |

## Value

dbHasCompleted() returns a logical scalar. For a query initiated by dbSendQuery() with non-empty result set, dbHasCompleted() returns FALSE initially and TRUE after calling dbFetch() without limit. For a query initiated by dbSendStatement(), dbHasCompleted() always returns TRUE.

## See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbGetStatement(), dbIsReadOnly(), dbIsValid(), dbQuoteLiteral(), dbQuoteString()

Other DBIResultArrow generics: DBIResultArrow-class, dbBind(), dbClearResult(), dbFetchArrow(), dbFetchArrowChunk(), dbIsValid()

Other data retrieval generics: dbBind(), dbClearResult(), dbFetch(), dbFetchArrow(), dbFetchArrowChunk(), dbGetQuery(), dbGetQueryArrow(), dbSendQuery(), dbSendQueryArrow()

---

# dbIsValid,DatabaseConnectorDbiConnection-method

*Is this DBMS object still valid?*

---

## Description

This generic tests whether a database object is still valid (i.e. it hasn't been disconnected or cleared).

## Usage

```
## S4 method for signature 'DatabaseConnectorDbiConnection'
dbIsValid(dbObj, ...)
```

## Arguments

| | |
|---|---|
| dbObj | An object inheriting from DBIObject, i.e. DBIDriver, DBIConnection, or a DBIResult |
| ... | Other arguments to methods. |

## Value

dbIsValid() returns a logical scalar, TRUE if the object specified by dbObj is valid, FALSE otherwise. A DBIConnection object is initially valid, and becomes invalid after disconnecting with dbDisconnect(). For an invalid connection object (e.g., for some drivers if the object is saved to a file and then restored), the method also returns FALSE. A DBIResult object is valid after a call to dbSendQuery(), and stays valid even after all rows have been fetched; only clearing it with dbClearResult() invalidates it. A DBIResult object is also valid after a call to dbSendStatement(), and stays valid after querying the number of rows affected; only clearing it with dbClearResult() invalidates it. If the connection to the database system is dropped (e.g., due to connectivity problems, server failure, etc.), dbIsValid() should return FALSE. This is not tested automatically.

## See Also

Other DBIDriver generics: DBIDriver-class, dbCanConnect(), dbConnect(), dbDataType(), dbDriver(), dbGetInfo(), dbIsReadOnly(), dbListConnections()

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbAppendTableArrow(), dbCreateTable(), dbCreateTableArrow(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbGetQueryArrow(), dbIsReadOnly(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbQuoteIdentifier(), dbReadTable(), dbReadTableArrow(), dbRemoveTable(), dbSendQuery(), dbSendQueryArrow(), dbSendStatement(), dbUnquoteIdentifier(), dbWriteTable(), dbWriteTableArrow()

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbQuoteLiteral(), dbQuoteString()

Other DBIResultArrow generics: DBIResultArrow-class, dbBind(), dbClearResult(), dbFetchArrow(), dbFetchArrowChunk(), dbHasCompleted()

---

# dbIsValid,DatabaseConnectorJdbcConnection-method

*Is this DBMS object still valid?*

---

## Description

This generic tests whether a database object is still valid (i.e. it hasn't been disconnected or cleared).

## Usage

```
## S4 method for signature 'DatabaseConnectorJdbcConnection'
dbIsValid(dbObj, ...)
```

## Arguments

| | |
|---|---|
| dbObj | An object inheriting from DBIObject, i.e. DBIDriver, DBIConnection, or a DBIResult |
| ... | Other arguments to methods. |

## Value

dbIsValid() returns a logical scalar, TRUE if the object specified by dbObj is valid, FALSE otherwise. A DBIConnection object is initially valid, and becomes invalid after disconnecting with dbDisconnect(). For an invalid connection object (e.g., for some drivers if the object is saved to a file and then restored), the method also returns FALSE. A DBIResult object is valid after a call to dbSendQuery(), and stays valid even after all rows have been fetched; only clearing it with dbClearResult() invalidates it. A DBIResult object is also valid after a call to dbSendStatement(), and stays valid after querying the number of rows affected; only clearing it with dbClearResult() invalidates it. If the connection to the database system is dropped (e.g., due to connectivity problems, server failure, etc.), dbIsValid() should return FALSE. This is not tested automatically.

## See Also

Other DBIDriver generics: DBIDriver-class, dbCanConnect(), dbConnect(), dbDataType(), dbDriver(), dbGetInfo(), dbIsReadOnly(), dbListConnections()

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbAppendTableArrow(), dbCreateTable(), dbCreateTableArrow(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbGetQueryArrow(), dbIsReadOnly(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbQuoteIdentifier(), dbReadTable(), dbReadTableArrow(), dbRemoveTable(), dbSendQuery(), dbSendQueryArrow(), dbSendStatement(), dbUnquoteIdentifier(), dbWriteTable(), dbWriteTableArrow()

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbQuoteLiteral(), dbQuoteString()

Other DBIResultArrow generics: DBIResultArrow-class, dbBind(), dbClearResult(), dbFetchArrow(), dbFetchArrowChunk(), dbHasCompleted()

---

# dbListFields,DatabaseConnectorConnection,character-method

*List field names of a remote table*

---

## Description

Returns the field names of a remote table as a character vector.

## Usage

```
## S4 method for signature 'DatabaseConnectorConnection,character'
dbListFields(conn, name, databaseSchema = NULL, ...)
```

## Arguments

| | |
|---|---|
| conn | A DBIConnection object, as returned by dbConnect(). |
| name | The table name, passed on to dbQuoteIdentifier(). Options are:<br>• a character string with the unquoted DBMS table name, e.g. "table_name",<br>• a call to Id() with components to the fully qualified table name, e.g. Id(schema = "my_schema", table = "table_name")<br>• a call to SQL() with the quoted and fully qualified table name given verbatim, e.g. SQL('"my_schema"."table_name"') |
| databaseSchema | The name of the database schema. See details for platform-specific details. |
| ... | Other parameters passed on to methods. |

## Details

The databaseSchema argument is interpreted differently according to the different platforms: SQL Server and PDW: The databaseSchema schema should specify both the database and the schema, e.g. 'my_database.dbo'. Impala: the databaseSchema should specify the database. Oracle: The databaseSchema should specify the Oracle 'user'. All other : The databaseSchema should specify the schema.

## Value

`dbListFields()` returns a character vector that enumerates all fields in the table in the correct order. This also works for temporary tables if supported by the database. The returned names are suitable for quoting with `dbQuoteIdentifier()`.

## See Also

`dbColumnInfo()` to get the type of the fields.

Other DBIConnection generics: `DBIConnection-class`, `dbAppendTable()`, `dbAppendTableArrow()`, `dbCreateTable()`, `dbCreateTableArrow()`, `dbDataType()`, `dbDisconnect()`, `dbExecute()`, `dbExistsTable()`, `dbGetException()`, `dbGetInfo()`, `dbGetQuery()`, `dbGetQueryArrow()`, `dbIsReadOnly()`, `dbIsValid()`, `dbListObjects()`, `dbListResults()`, `dbListTables()`, `dbQuoteIdentifier()`, `dbReadTable()`, `dbReadTableArrow()`, `dbRemoveTable()`, `dbSendQuery()`, `dbSendQueryArrow()`, `dbSendStatement()`, `dbUnquoteIdentifier()`, `dbWriteTable()`, `dbWriteTableArrow()`

---

# dbListTables,DatabaseConnectorConnection-method

*List remote tables*

---

## Description

Returns the unquoted names of remote tables accessible through this connection. This should include views and temporary objects, but not all database backends (in particular **RMariaDB** and **RMySQL**) support this.

## Usage

```
## S4 method for signature 'DatabaseConnectorConnection'
dbListTables(conn, databaseSchema = NULL, ...)
```

## Arguments

| | |
|---|---|
| `conn` | A DBIConnection object, as returned by `dbConnect()`. |
| `databaseSchema` | The name of the database schema. See details for platform-specific details. |
| `...` | Not used |

## Details

The `databaseSchema` argument is interpreted differently according to the different platforms: SQL Server and PDW: The databaseSchema schema should specify both the database and the schema, e.g. 'my_database.dbo'. Impala: the databaseSchema should specify the database. Oracle: The databaseSchema should specify the Oracle 'user'. All other : The databaseSchema should specify the schema.

## Value

dbListTables() returns a character vector that enumerates all tables and views in the database. Tables added with dbWriteTable() are part of the list. As soon a table is removed from the database, it is also removed from the list of database tables.

The same applies to temporary tables if supported by the database.

The returned names are suitable for quoting with dbQuoteIdentifier().

## See Also

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbAppendTableArrow(), dbCreateTable(), dbCreateTableArrow(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbGetQueryArrow(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbQuoteIdentifier(), dbReadTable(), dbReadTableArrow(), dbRemoveTable(), dbSendQuery(), dbSendQueryArrow(), dbSendStatement(), dbUnquoteIdentifier(), dbWriteTable(), dbWriteTableArrow()

---

# dbms *Get the database platform from a connection*

---

## Description

The SqlRender package provides functions that translate SQL from OHDSI-SQL to a target SQL dialect. These function need the name of the database platform to translate to. The dbms function returns the dbms for any DBI connection that can be passed along to SqlRender translation functions (see example).

## Usage

```
dbms(connection)
```

## Arguments

| | |
|---|---|
| connection | The connection to the database server created using either connect() or DBI::dbConnect(). |

## Value

The name of the database (dbms) used by SqlRender

## Examples

```
library(DatabaseConnector)
con <- connect(dbms = "sqlite", server = ":memory:")
dbms(con)
#> [1] "sqlite"
SqlRender::translate("DATEADD(d, 365, dateColumn)", targetDialect = dbms(con))
#> "CAST(STRFTIME('%s', DATETIME(dateColumn, 'unixepoch', (365)||' days')) AS REAL)"
disconnect(con)
```

dbReadTable,DatabaseConnectorConnection,character-method
*Read database tables as data frames*

## Description

Reads a database table to a data frame, optionally converting a column to row names and converting the column names to valid R identifiers. Use dbReadTableArrow() instead to obtain an Arrow object.

## Usage

```
## S4 method for signature 'DatabaseConnectorConnection,character'
dbReadTable(conn, name, databaseSchema = NULL, ...)
```

## Arguments

| | |
|---|---|
| conn | A DBIConnection object, as returned by dbConnect(). |
| name | The table name, passed on to dbQuoteIdentifier(). Options are:<br>• a character string with the unquoted DBMS table name, e.g. "table_name",<br>• a call to Id() with components to the fully qualified table name, e.g. Id(schema = "my_schema", table = "table_name")<br>• a call to SQL() with the quoted and fully qualified table name given verbatim, e.g. SQL('"my_schema"."table_name"') |
| databaseSchema | The name of the database schema. See details for platform-specific details. |
| ... | Other parameters passed on to methods. |

## Details

The databaseSchema argument is interpreted differently according to the different platforms: SQL Server and PDW: The databaseSchema schema should specify both the database and the schema, e.g. 'my_database.dbo'. Impala: the databaseSchema should specify the database. Oracle: The databaseSchema should specify the Oracle 'user'. All other : The databaseSchema should specify the schema.

## Value

dbReadTable() returns a data frame that contains the complete data from the remote table, effectively the result of calling dbGetQuery() with SELECT * FROM <name>.

An empty table is returned as a data frame with zero rows.

The presence of rownames depends on the row.names argument, see sqlColumnToRownames() for details:

- If FALSE or NULL, the returned data frame doesn't have row names.
- If TRUE, a column named "row_names" is converted to row names.

- If NA, a column named "row_names" is converted to row names if it exists, otherwise no translation occurs.
- If a string, this specifies the name of the column in the remote table that contains the row names.

The default is row.names = FALSE.

If the database supports identifiers with special characters, the columns in the returned data frame are converted to valid R identifiers if the check.names argument is TRUE, If check.names = FALSE, the returned table has non-syntactic column names without quotes.

## See Also

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbAppendTableArrow(), dbCreateTable(), dbCreateTableArrow(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbGetQueryArrow(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbQuoteIdentifier(), dbReadTableArrow(), dbRemoveTable(), dbSendQuery(), dbSendQueryArrow(), dbSendStatement(), dbUnquoteIdentifier(), dbWriteTable(), dbWriteTableArrow()

---

# dbRemoveTable,DatabaseConnectorConnection,ANY-method

*Remove a table from the database*

---

## Description

Remove a remote table (e.g., created by dbWriteTable()) from the database.

## Usage

```
## S4 method for signature 'DatabaseConnectorConnection,ANY'
dbRemoveTable(conn, name, databaseSchema = NULL, ...)
```

## Arguments

| | |
|---|---|
| conn | A DBIConnection object, as returned by dbConnect(). |
| name | The table name, passed on to dbQuoteIdentifier(). Options are:<br>• a character string with the unquoted DBMS table name, e.g. "table_name",<br>• a call to Id() with components to the fully qualified table name, e.g. Id(schema = "my_schema", table = "table_name")<br>• a call to SQL() with the quoted and fully qualified table name given verbatim, e.g. SQL('"my_schema"."table_name"') |
| databaseSchema | The name of the database schema. See details for platform-specific details. |
| ... | Other parameters passed on to methods. |

## Details

The databaseSchema argument is interpreted differently according to the different platforms: SQL Server and PDW: The databaseSchema schema should specify both the database and the schema, e.g. 'my_database.dbo'. Impala: the databaseSchema should specify the database. Oracle: The databaseSchema should specify the Oracle 'user'. All other : The databaseSchema should specify the schema.

## Value

`dbRemoveTable()` returns `TRUE`, invisibly.

## See Also

Other DBIConnection generics: `DBIConnection-class`, `dbAppendTable()`, `dbAppendTableArrow()`, `dbCreateTable()`, `dbCreateTableArrow()`, `dbDataType()`, `dbDisconnect()`, `dbExecute()`, `dbExistsTable()`, `dbGetException()`, `dbGetInfo()`, `dbGetQuery()`, `dbGetQueryArrow()`, `dbIsReadOnly()`, `dbIsValid()`, `dbListFields()`, `dbListObjects()`, `dbListResults()`, `dbListTables()`, `dbQuoteIdentifier()`, `dbReadTable()`, `dbReadTableArrow()`, `dbSendQuery()`, `dbSendQueryArrow()`, `dbSendStatement()`, `dbUnquoteIdentifier()`, `dbWriteTable()`, `dbWriteTableArrow()`

---

# dbSendQuery,DatabaseConnectorDbiConnection,character-method

*Execute a query on a given database connection*

---

## Description

The `dbSendQuery()` method only submits and synchronously executes the SQL query to the database engine. It does *not* extract any records — for that you need to use the `dbFetch()` method, and then you must call `dbClearResult()` when you finish fetching the records you need. For interactive use, you should almost always prefer `dbGetQuery()`. Use `dbSendQueryArrow()` or `dbGetQueryArrow()` instead to retrieve the results as an Arrow object.

## Usage

```
## S4 method for signature 'DatabaseConnectorDbiConnection,character'
dbSendQuery(conn, statement, ...)
```

## Arguments

| | |
|---|---|
| `conn` | A DBIConnection object, as returned by `dbConnect()`. |
| `statement` | a character string containing SQL. |
| `...` | Other parameters passed on to methods. |

## Details

This method is for SELECT queries only. Some backends may support data manipulation queries through this method for compatibility reasons. However, callers are strongly encouraged to use dbSendStatement() for data manipulation statements.

The query is submitted to the database server and the DBMS executes it, possibly generating vast amounts of data. Where these data live is driver-specific: some drivers may choose to leave the output on the server and transfer them piecemeal to R, others may transfer all the data to the client – but not necessarily to the memory that R manages. See individual drivers' dbSendQuery() documentation for details.

## Value

dbSendQuery() returns an S4 object that inherits from DBIResult. The result set can be used with dbFetch() to extract records. Once you have finished using a result, make sure to clear it with dbClearResult().

## See Also

For updates: dbSendStatement() and dbExecute().

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbAppendTableArrow(), dbCreateTable(), dbCreateTableArrow(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbGetQueryArrow(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbQuoteIdentifier(), dbReadTable(), dbReadTableArrow(), dbRemoveTable(), dbSendQueryArrow(), dbSendStatement(), dbUnquoteIdentifier(), dbWriteTable(), dbWriteTableArrow()

Other data retrieval generics: dbBind(), dbClearResult(), dbFetch(), dbFetchArrow(), dbFetchArrowChunk(), dbGetQuery(), dbGetQueryArrow(), dbHasCompleted(), dbSendQueryArrow()

---

# dbSendQuery,DatabaseConnectorJdbcConnection,character-method

*Execute a query on a given database connection*

---

## Description

The dbSendQuery() method only submits and synchronously executes the SQL query to the database engine. It does *not* extract any records — for that you need to use the dbFetch() method, and then you must call dbClearResult() when you finish fetching the records you need. For interactive use, you should almost always prefer dbGetQuery(). Use dbSendQueryArrow() or dbGetQueryArrow() instead to retrieve the results as an Arrow object.

## Usage

```
## S4 method for signature 'DatabaseConnectorJdbcConnection,character'
dbSendQuery(conn, statement, ...)
```

### Arguments

| | |
|---|---|
| conn | A DBIConnection object, as returned by dbConnect(). |
| statement | a character string containing SQL. |
| ... | Other parameters passed on to methods. |

### Details

This method is for SELECT queries only. Some backends may support data manipulation queries through this method for compatibility reasons. However, callers are strongly encouraged to use dbSendStatement() for data manipulation statements.

The query is submitted to the database server and the DBMS executes it, possibly generating vast amounts of data. Where these data live is driver-specific: some drivers may choose to leave the output on the server and transfer them piecemeal to R, others may transfer all the data to the client – but not necessarily to the memory that R manages. See individual drivers' dbSendQuery() documentation for details.

### Value

dbSendQuery() returns an S4 object that inherits from DBIResult. The result set can be used with dbFetch() to extract records. Once you have finished using a result, make sure to clear it with dbClearResult().

### See Also

For updates: dbSendStatement() and dbExecute().

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbAppendTableArrow(), dbCreateTable(), dbCreateTableArrow(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbGetQueryArrow(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbQuoteIdentifier(), dbReadTable(), dbReadTableArrow(), dbRemoveTable(), dbSendQueryArrow(), dbSendStatement(), dbUnquoteIdentifier(), dbWriteTable(), dbWriteTableArrow()

Other data retrieval generics: dbBind(), dbClearResult(), dbFetch(), dbFetchArrow(), dbFetchArrowChunk(), dbGetQuery(), dbGetQueryArrow(), dbHasCompleted(), dbSendQueryArrow()

---

## dbSendStatement,DatabaseConnectorConnection,character-method

*Execute a data manipulation statement on a given database connection*

---

### Description

The dbSendStatement() method only submits and synchronously executes the SQL data manipulation statement (e.g., UPDATE, DELETE, INSERT INTO, DROP TABLE, ...) to the database engine. To query the number of affected rows, call dbGetRowsAffected() on the returned result object. You must also call dbClearResult() after that. For interactive use, you should almost always prefer dbExecute().

## Usage

```
## S4 method for signature 'DatabaseConnectorConnection,character'
dbSendStatement(conn, statement, ...)
```

## Arguments

| | |
|---|---|
| conn | A DBIConnection object, as returned by dbConnect(). |
| statement | a character string containing SQL. |
| ... | Other parameters passed on to methods. |

## Details

dbSendStatement() comes with a default implementation that simply forwards to dbSendQuery(), to support backends that only implement the latter.

## Value

dbSendStatement() returns an S4 object that inherits from DBIResult. The result set can be used with dbGetRowsAffected() to determine the number of rows affected by the query. Once you have finished using a result, make sure to clear it with dbClearResult().

## See Also

For queries: dbSendQuery() and dbGetQuery().

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbAppendTableArrow(), dbCreateTable(), dbCreateTableArrow(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbGetQueryArrow(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbQuoteIdentifier(), dbReadTable(), dbReadTableArrow(), dbRemoveTable(), dbSendQuery(), dbSendQueryArrow(), dbUnquoteIdentifier(), dbWriteTable(), dbWriteTableArrow()

Other command execution generics: dbBind(), dbClearResult(), dbExecute(), dbGetRowsAffected()

---

# dbUnloadDriver,DatabaseConnectorDriver-method

*Load and unload database drivers*

---

## Description

These methods are deprecated, please consult the documentation of the individual backends for the construction of driver instances.

dbDriver() is a helper method used to create an new driver object given the name of a database or the corresponding R package. It works through convention: all DBI-extending packages should provide an exported object with the same name as the package. dbDriver() just looks for this object in the right places: if you know what database you are connecting to, you should call the function directly.

dbUnloadDriver() is not implemented for modern backends.

## Usage

```
## S4 method for signature 'DatabaseConnectorDriver'
dbUnloadDriver(drv, ...)
```

## Arguments

| | |
|---|---|
| drv | an object that inherits from DBIDriver as created by dbDriver. |
| ... | any other arguments are passed to the driver drvName. |

## Details

The client part of the database communication is initialized (typically dynamically loading C code, etc.) but note that connecting to the database engine itself needs to be done through calls to dbConnect.

## Value

In the case of dbDriver, an driver object whose class extends DBIDriver. This object may be used to create connections to the actual DBMS engine.

In the case of dbUnloadDriver, a logical indicating whether the operation succeeded or not.

## See Also

Other DBIDriver generics: DBIDriver-class, dbCanConnect(), dbConnect(), dbDataType(), dbGetInfo(), dbIsReadOnly(), dbIsValid(), dbListConnections()

Other DBIDriver generics: DBIDriver-class, dbCanConnect(), dbConnect(), dbDataType(), dbGetInfo(), dbIsReadOnly(), dbIsValid(), dbListConnections()

---

# dbWriteTable,DatabaseConnectorConnection,ANY-method

*Copy data frames to database tables*

---

## Description

Writes, overwrites or appends a data frame to a database table, optionally converting row names to a column and specifying SQL data types for fields.

## Usage

```
## S4 method for signature 'DatabaseConnectorConnection,ANY'
dbWriteTable(
  conn,
  name,
  value,
  databaseSchema = NULL,
  overwrite = FALSE,
```

```
  append = FALSE,
  temporary = FALSE,
  ...
)
```

## Arguments

| | |
|---|---|
| conn | A DBIConnection object, as returned by dbConnect(). |
| name | The table name, passed on to dbQuoteIdentifier(). Options are:<br>• a character string with the unquoted DBMS table name, e.g. "table_name",<br>• a call to Id() with components to the fully qualified table name, e.g. Id(schema = "my_schema", table = "table_name")<br>• a call to SQL() with the quoted and fully qualified table name given verbatim, e.g. SQL('"my_schema"."table_name"') |
| value | A data.frame (or coercible to data.frame). |
| databaseSchema | The name of the database schema. See details for platform-specific details. |
| overwrite | Overwrite an existing table (if exists)? |
| append | Append to existing table? |
| temporary | Should the table created as a temp table? |
| ... | Other parameters passed on to methods. |

## Details

The databaseSchema argument is interpreted differently according to the different platforms: SQL Server and PDW: The databaseSchema schema should specify both the database and the schema, e.g. 'my_database.dbo'. Impala: the databaseSchema should specify the database. Oracle: The databaseSchema should specify the Oracle 'user'. All other : The databaseSchema should specify the schema.

## Value

dbWriteTable() returns TRUE, invisibly.

## See Also

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbAppendTableArrow(), dbCreateTable(), dbCreateTableArrow(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbGetQueryArrow(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbQuoteIdentifier(), dbReadTable(), dbReadTableArrow(), dbRemoveTable(), dbSendQuery(), dbSendQueryArrow(), dbSendStatement(), dbUnquoteIdentifier(), dbWriteTableArrow()

## disconnect *Disconnect from the server*

### Description

Close the connection to the server.

### Usage

```
disconnect(connection)
```

### Arguments

| | |
|---|---|
| connection | The connection to the database server created using either connect() or DBI::dbConnect(). |

### Examples

```
## Not run:
connectionDetails <- createConnectionDetails(
  dbms = "postgresql",
  server = "localhost",
  user = "root",
  password = "blah"
)
conn <- connect(connectionDetails)
count <- querySql(conn, "SELECT COUNT(*) FROM person")
disconnect(conn)

## End(Not run)
```

## downloadJdbcDrivers *Download DatabaseConnector JDBC Jar files*

### Description

Download the DatabaseConnector JDBC drivers from https://ohdsi.github.io/DatabaseConnectorJars/

### Usage

```
downloadJdbcDrivers(
  dbms,
  pathToDriver = Sys.getenv("DATABASECONNECTOR_JAR_FOLDER"),
  method = "auto",
  ...
)
```

## Arguments

| | |
|---|---|
| dbms | The type of DBMS to download Jar files for.<br>• "postgresql" for PostgreSQL<br>• "redshift" for Amazon Redshift<br>• "sql server", "pdw" or "synapse" for Microsoft SQL Server<br>• "oracle" for Oracle<br>• "spark" for Spark<br>• "snowflake" for Snowflake<br>• "bigquery" for Google BigQuery<br>• "iris" for InterSystems IRIS<br>• "all" for all aforementioned platforms |
| pathToDriver | The full path to the folder where the JDBC driver .jar files should be downloaded to. By default the value of the environment variable "DATABASECONNECTOR_JAR_FOLDER" is used. |
| method | The method used for downloading files. See ?download.file for details and options. |
| ... | Further arguments passed on to download.file. |

## Details

The following versions of the JDBC drivers are currently used:

- PostgreSQL: V42.7.3
- RedShift: V2.1.0.9
- SQL Server: V9.2.0
- Oracle: V19.8
- Spark (Databricks): V2.6.36
- Snowflake: V3.24.0
- BigQuery: v1.3.2.1003
- InterSystems IRIS: v3.10.2

## Value

Invisibly returns the destination if the download was successful.

## Examples

```
## Not run:
downloadJdbcDrivers("redshift")

## End(Not run)
```

dropEmulatedTempTables *Drop all emulated temp tables.*

## Description

On some DBMSs, like Oracle and BigQuery, `DatabaseConnector` through `SqlRender` emulates temp tables in a schema provided by the user. Ideally, these tables are deleted by the application / R script creating them, but for various reasons orphan temp tables may remain. This function drops all emulated temp tables created in this session only.

## Usage

```
dropEmulatedTempTables(
  connection,
  tempEmulationSchema = getOption("sqlRenderTempEmulationSchema")
)
```

## Arguments

| | |
|---|---|
| `connection` | The connection to the database server created using either `connect()` or `DBI::dbConnect()`. |
| `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. |

## Value

Invisibly returns the list of deleted emulated temp tables.

executeSql *Execute SQL code*

## Description

This function executes SQL consisting of one or more statements.

## Usage

```
executeSql(
  connection,
  sql,
  profile = FALSE,
  progressBar = !as.logical(Sys.getenv("TESTTHAT", unset = FALSE)),
  reportOverallTime = TRUE,
  errorReportFile = file.path(getwd(), "errorReportSql.txt"),
  runAsBatch = FALSE
)
```

## Arguments

| | |
|---|---|
| connection | The connection to the database server created using either connect() or DBI::dbConnect(). |
| sql | The SQL to be executed |
| profile | When true, each separate statement is written to file prior to sending to the server, and the time taken to execute a statement is displayed. |
| progressBar | When true, a progress bar is shown based on the statements in the SQL code. |
| reportOverallTime | When true, the function will display the overall time taken to execute all statements. |
| errorReportFile | The file where an error report will be written if an error occurs. Defaults to 'errorReportSql.txt' in the current working directory. |
| runAsBatch | When true the SQL statements are sent to the server as a single batch, and executed there. This will be faster if you have many small SQL statements, but there will be no progress bar, and no per-statement error messages. If the database platform does not support batched updates the query is executed without batching. |

## Details

This function splits the SQL in separate statements and sends it to the server for execution. If an error occurs during SQL execution, this error is written to a file to facilitate debugging. Optionally, a progress bar is shown and the total time taken to execute the SQL is displayed. Optionally, each separate SQL statement is written to file, and the execution time per statement is shown to aid in detecting performance issues.

## Examples

```
## Not run:
connectionDetails <- createConnectionDetails(
  dbms = "postgresql",
  server = "localhost",
  user = "root",
  password = "blah",
  schema = "cdm_v4"
)
conn <- connect(connectionDetails)
executeSql(conn, "CREATE TABLE x (k INT); CREATE TABLE y (k INT);")
disconnect(conn)

## End(Not run)
```

## existsTable *Does the table exist?*

### Description

Checks whether a table exists. Accounts for surrounding escape characters. Case insensitive.

### Usage

```
existsTable(connection, databaseSchema, tableName)
```

### Arguments

| | |
|---|---|
| connection | The connection to the database server created using either connect() or DBI::dbConnect(). |
| databaseSchema | The name of the database schema. See details for platform-specific details. |
| tableName | The name of the table to check. |

### Details

The databaseSchema argument is interpreted differently according to the different platforms: SQL Server and PDW: The databaseSchema schema should specify both the database and the schema, e.g. 'my_database.dbo'. Impala: the databaseSchema should specify the database. Oracle: The databaseSchema should specify the Oracle 'user'. All other : The databaseSchema should specify the schema.

### Value

A logical value indicating whether the table exits.

## extractQueryTimes *Extract query times from a* ParallelLogger *log file*

### Description

When using the ParallelLogger default file logger, and using options(LOG_DATABASECONNECTOR_SQL = TRUE), DatabaseConnector will log all SQL sent to the server, and the time to get a response.

This function parses the log file, producing a data frame with time per query.

### Usage

```
extractQueryTimes(logFileName)
```

### Arguments

| | |
|---|---|
| logFileName | Name of the ParallelLogger log file. Assumes the file was created using the default file logger. |

### Value

A data frame with queries and their run times in milliseconds.

### Examples

```
connection <- connect(dbms = "sqlite", server = ":memory:")
logFile <- tempfile(fileext = ".log")
ParallelLogger::addDefaultFileLogger(fileName = logFile, name = "MY_LOGGER")
options(LOG_DATABASECONNECTOR_SQL = TRUE)

executeSql(connection, "CREATE TABLE test (x INT);")
querySql(connection, "SELECT * FROM test;")

extractQueryTimes(logFile)

ParallelLogger::unregisterLogger("MY_LOGGER")
unlink(logFile)
disconnect(connection)
```

---

## getAvailableJavaHeapSpace *Get available Java heap space*

---

### Description

For debugging purposes: get the available Java heap space.

### Usage

```
getAvailableJavaHeapSpace()
```

### Value

The Java heap space (in bytes).

---

## getTableNames *List all tables in a database schema.*

---

### Description

This function returns a list of all tables in a database schema.

### Usage

```
getTableNames(connection, databaseSchema = NULL, cast = "lower")
```

### Arguments

| | |
|---|---|
| connection | The connection to the database server created using either connect() or DBI::dbConnect(). |
| databaseSchema | The name of the database schema. See details for platform-specific details. |
| cast | Should the table names be cast to uppercase or lowercase before being returned? Valid options are "upper" , "lower" (default), "none" (no casting is done) |

### Details

The databaseSchema argument is interpreted differently according to the different platforms: SQL Server and PDW: The databaseSchema schema should specify both the database and the schema, e.g. 'my_database.dbo'. Impala: the databaseSchema should specify the database. Oracle: The databaseSchema should specify the Oracle 'user'. All other : The databaseSchema should specify the schema.

### Value

A character vector of table names.

---

## inDatabaseSchema *Refer to a table in a database schema*

---

### Description

Can be used with dplyr::tbl() to indicate a table in a specific database schema.

### Usage

```
inDatabaseSchema(databaseSchema, table)
```

### Arguments

| | |
|---|---|
| databaseSchema | The name of the database schema. See details for platform-specific details. |
| table | The name of the table in the database schema. |

### Details

The databaseSchema argument is interpreted differently according to the different platforms: SQL Server and PDW: The databaseSchema schema should specify both the database and the schema, e.g. 'my_database.dbo'. Impala: the databaseSchema should specify the database. Oracle: The databaseSchema should specify the Oracle 'user'. All other : The databaseSchema should specify the schema.

### Value

An object representing the table and database schema.

`insertTable` *Insert a table on the server*

## Description

This function sends the data in a data frame to a table on the server. Either a new table is created, or the data is appended to an existing table.

## Usage

```
insertTable(
  connection,
  databaseSchema = NULL,
  tableName,
  data,
  dropTableIfExists = TRUE,
  createTable = TRUE,
  tempTable = FALSE,
  tempEmulationSchema = getOption("sqlRenderTempEmulationSchema"),
  bulkLoad = Sys.getenv("DATABASE_CONNECTOR_BULK_UPLOAD"),
  useMppBulkLoad = Sys.getenv("USE_MPP_BULK_LOAD"),
  progressBar = FALSE,
  camelCaseToSnakeCase = FALSE
)
```

## Arguments

| | |
|---|---|
| `connection` | The connection to the database server created using either `connect()` or `DBI::dbConnect()`. |
| `databaseSchema` | The name of the database schema. See details for platform-specific details. |
| `tableName` | The name of the table where the data should be inserted. |
| `data` | The data frame containing the data to be inserted. |
| `dropTableIfExists` | Drop the table if the table already exists before writing? |
| `createTable` | Create a new table? If false, will append to existing table. |
| `tempTable` | Should the table created as a temp table? |
| `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. |
| `bulkLoad` | If using Redshift, PDW, Hive or Postgres, use more performant bulk loading techniques. Does not work for temp tables (except for HIVE). See Details for requirements for the various platforms. |
| `useMppBulkLoad` | DEPRECATED. Use `bulkLoad` instead. |
| `progressBar` | Show a progress bar when uploading? |

camelCaseToSnakeCase
: If TRUE, the data frame column names are assumed to use camelCase and are converted to snake_case before uploading.

## Details

The databaseSchema argument is interpreted differently according to the different platforms: SQL Server and PDW: The databaseSchema schema should specify both the database and the schema, e.g. 'my_database.dbo'. Impala: the databaseSchema should specify the database. Oracle: The databaseSchema should specify the Oracle 'user'. All other : The databaseSchema should specify the schema.

This function sends the data in a data frame to a table on the server. Either a new table is created, or the data is appended to an existing table. NA values are inserted as null values in the database.

Bulk uploading:

Redshift: The MPP bulk loading relies upon the CloudyR S3 library to test a connection to an S3 bucket using AWS S3 credentials. Credentials are configured directly into the System Environment using the following keys: Sys.setenv("AWS_ACCESS_KEY_ID" = "some_access_key_id", "AWS_SECRET_ACCESS_KEY" = "some_secret_access_key", "AWS_DEFAULT_REGION" = "some_aws_region", "AWS_BUCKET_NAME" = "some_bucket_name", "AWS_OBJECT_KEY" = "some_object_key", "AWS_SSE_TYPE" = "server_side_encryption_type").

Spark (DataBricks): The MPP bulk loading relies upon the AzureStor library to test a connection to an Azure ADLS Gen2 storage container using Azure credentials. Credentials are configured directly into the System Environment using the following keys: Sys.setenv("AZR_STORAGE_ACCOUNT" = "some_azure_storage_account", "AZR_ACCOUNT_KEY" = "some_secret_account_key", "AZR_CONTAINER_NAME" = "some_container_name").

PDW: The MPP bulk loading relies upon the client having a Windows OS and the DWLoader exe installed, and the following permissions granted: –Grant BULK Load permissions - needed at a server level USE master; GRANT ADMINISTER BULK OPERATIONS TO user; –Grant Staging database permissions - we will use the user db. USE scratch; EXEC sp_addrolemember 'db_ddladmin', user; Set the R environment variable DWLOADER_PATH to the location of the binary.

PostgreSQL: Uses the 'psql' executable to upload. Set the POSTGRES_PATH environment variable to the Postgres binary path, e.g. 'C:/Program Files/PostgreSQL/11/bin' on Windows or '/Library/PostgreSQL/16/bin' on MacOs.

## Examples

```
## Not run:
connectionDetails <- createConnectionDetails(
  dbms = "mysql",
  server = "localhost",
  user = "root",
  password = "blah"
)
conn <- connect(connectionDetails)
data <- data.frame(x = c(1, 2, 3), y = c("a", "b", "c"))
insertTable(conn, "my_schema", "my_table", data)
disconnect(conn)
```

```
## bulk data insert with Redshift or PDW
connectionDetails <- createConnectionDetails(
  dbms = "redshift",
  server = "localhost",
  user = "root",
  password = "blah",
  schema = "cdm_v5"
)
conn <- connect(connectionDetails)
data <- data.frame(x = c(1, 2, 3), y = c("a", "b", "c"))
insertTable(
  connection = connection,
  databaseSchema = "scratch",
  tableName = "somedata",
  data = data,
  dropTableIfExists = TRUE,
  createTable = TRUE,
  tempTable = FALSE,
  bulkLoad = TRUE
) # or, Sys.setenv("DATABASE_CONNECTOR_BULK_UPLOAD" = TRUE)

## End(Not run)
```

---

`isSqlReservedWord` *Test a character vector of SQL names for SQL reserved words*

---

## Description

This function checks a character vector against a predefined list of reserved SQL words.

## Usage

```
isSqlReservedWord(sqlNames, warn = FALSE)
```

## Arguments

| | |
|---|---|
| `sqlNames` | A character vector containing table or field names to check. |
| `warn` | (logical) Should a warn be thrown if invalid SQL names are found? |

## Value

A logical vector with length equal to sqlNames that is TRUE for each name that is reserved and FALSE otherwise

## jdbcDrivers *How to download and use JDBC drivers for the various data platforms.*

### Description

Below are instructions for downloading JDBC drivers for the various data platforms. Once downloaded use the `pathToDriver` argument in the `connect()` or `createConnectionDetails()` functions to point to the driver. Alternatively, you can set the 'DATABASECONNECTOR_JAR_FOLDER' environmental variable, for example in your .Renviron file (recommended).

### SQL Server, Oracle, PostgreSQL, PDW, Snowflake, Spark, RedShift, Azure Synapse, BigQuery, InterSystems IRIS

Use the `downloadJdbcDrivers()` function to download these drivers from the OHDSI GitHub pages.

### Netezza

Read the instructions here on how to obtain the Netezza JDBC driver.

### Impala

Go to Cloudera's site, pick your OS version, and click "GET IT NOW!'. Register, and you should be able to download the driver.

### SQLite

For SQLite we actually don't use a JDBC driver. Instead, we use the RSQLite package, which can be installed using `install.packages("RSQLite")`.

## querySql *Retrieve data to a data.frame*

### Description

This function sends SQL to the server, and returns the results.

### Usage

```
querySql(
  connection,
  sql,
  errorReportFile = file.path(getwd(), "errorReportSql.txt"),
  snakeCaseToCamelCase = FALSE,
 integerAsNumeric = getOption("databaseConnectorIntegerAsNumeric", default = TRUE),
 integer64AsNumeric = getOption("databaseConnectorInteger64AsNumeric", default = TRUE)
)
```

## Arguments

| | |
|---|---|
| connection | The connection to the database server created using either connect() or DBI::dbConnect(). |
| sql | The SQL to be send. |
| errorReportFile | The file where an error report will be written if an error occurs. Defaults to 'errorReportSql.txt' in the current working directory. |
| snakeCaseToCamelCase | If true, field names are assumed to use snake_case, and are converted to camelCase. |
| integerAsNumeric | Logical: should 32-bit integers be converted to numeric (double) values? If FALSE 32-bit integers will be represented using R's native Integer class. |
| integer64AsNumeric | Logical: should 64-bit integers be converted to numeric (double) values? If FALSE 64-bit integers will be represented using bit64::integer64. |

## Details

Fields will be automatically converted for improved consistenty in these situations:

- SQLite: Fields with names ending in _date will be converted to DATE fields. Rationale: SQLite does not support DATE fields.
- SQLite: Fields with names ending in _datetime will be converted to POSIXct fields. Rationale: SQLite does not support DATETIME fields.
- BigQuery and Snowflake: Integer fields will be converted to Integer if it fits in an integer, or will remain Integer64 otherwise. Rationale: these platforms do not distinguish between INT and BIGINT.

This function sends the SQL to the server and retrieves the results. If an error occurs during SQL execution, this error is written to a file to facilitate debugging. Null values in the database are converted to NA values in R.

## Value

A data frame.

## Examples

```
## Not run:
connectionDetails <- createConnectionDetails(
  dbms = "postgresql",
  server = "localhost",
  user = "root",
  password = "blah",
  schema = "cdm_v4"
)
conn <- connect(connectionDetails)
count <- querySql(conn, "SELECT COUNT(*) FROM person")
disconnect(conn)
```

```
## End(Not run)
```

---

querySqlToAndromeda *Retrieves data to a local Andromeda object*

---

## Description

This function sends SQL to the server, and returns the results in a local Andromeda object

## Usage

```
querySqlToAndromeda(
  connection,
  sql,
  andromeda,
  andromedaTableName,
  errorReportFile = file.path(getwd(), "errorReportSql.txt"),
  snakeCaseToCamelCase = FALSE,
  appendToTable = FALSE,
 integerAsNumeric = getOption("databaseConnectorIntegerAsNumeric", default = TRUE),
 integer64AsNumeric = getOption("databaseConnectorInteger64AsNumeric", default = TRUE)
)
```

## Arguments

| | |
|---|---|
| connection | The connection to the database server created using either `connect()` or `DBI::dbConnect()`. |
| sql | The SQL to be sent. |
| andromeda | An open Andromeda object, for example as created using `Andromeda::andromeda()`. |
| andromedaTableName | The name of the table in the local Andromeda object where the results of the query will be stored. |
| errorReportFile | The file where an error report will be written if an error occurs. Defaults to 'errorReportSql.txt' in the current working directory. |
| snakeCaseToCamelCase | If true, field names are assumed to use snake_case, and are converted to camelCase. |
| appendToTable | If FALSE, any existing table in the Andromeda with the same name will be replaced with the new data. If TRUE, data will be appended to an existing table, assuming it has the exact same structure. |
| integerAsNumeric | Logical: should 32-bit integers be converted to numeric (double) values? If FALSE 32-bit integers will be represented using R's native `Integer` class. |
| integer64AsNumeric | Logical: should 64-bit integers be converted to numeric (double) values? If FALSE 64-bit integers will be represented using `bit64::integer64`. |

## Details

Retrieves data from the database server and stores it in a local Andromeda object. This allows very large data sets to be retrieved without running out of memory. If an error occurs during SQL execution, this error is written to a file to facilitate debugging. Null values in the database are converted to NA values in R.If a table with the same name already exists in the local Andromeda object it is replaced.

## Value

Invisibly returns the andromeda. The Andromeda object will have a table added with the query results.

## Examples

```
## Not run:
andromeda <- Andromeda::andromeda()
connectionDetails <- createConnectionDetails(
  dbms = "postgresql",
  server = "localhost",
  user = "root",
  password = "blah",
  schema = "cdm_v4"
)
conn <- connect(connectionDetails)
querySqlToAndromeda(
  connection = conn,
  sql = "SELECT * FROM person;",
  andromeda = andromeda,
  andromedaTableName = "foo"
)
disconnect(conn)

andromeda$foo

## End(Not run)
```

---

# renderTranslateExecuteSql

*Render, translate, execute SQL code*

---

## Description

This function renders, translates, and executes SQL consisting of one or more statements.

## Usage

```
renderTranslateExecuteSql(
  connection,
```

```
  sql,
  profile = FALSE,
  progressBar = TRUE,
  reportOverallTime = TRUE,
  errorReportFile = file.path(getwd(), "errorReportSql.txt"),
  runAsBatch = FALSE,
  tempEmulationSchema = getOption("sqlRenderTempEmulationSchema"),
  ...
)
```

## Arguments

| | |
|---|---|
| connection | The connection to the database server created using either connect() or DBI::dbConnect(). |
| sql | The SQL to be executed |
| profile | When true, each separate statement is written to file prior to sending to the server, and the time taken to execute a statement is displayed. |
| progressBar | When true, a progress bar is shown based on the statements in the SQL code. |
| reportOverallTime | When true, the function will display the overall time taken to execute all statements. |
| errorReportFile | The file where an error report will be written if an error occurs. Defaults to 'errorReportSql.txt' in the current working directory. |
| runAsBatch | When true the SQL statements are sent to the server as a single batch, and executed there. This will be faster if you have many small SQL statements, but there will be no progress bar, and no per-statement error messages. If the database platform does not support batched updates the query is executed as ordinarily. |
| 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. |
| ... | Parameters that will be used to render the SQL. |

## Details

This function calls the `render` and `translate` functions in the `SqlRender` package before calling `executeSql()`.

## Examples

```
## Not run:
connectionDetails <- createConnectionDetails(
  dbms = "postgresql",
  server = "localhost",
  user = "root",
  password = "blah",
  schema = "cdm_v4"
)
```

```
conn <- connect(connectionDetails)
renderTranslateExecuteSql(connection,
  sql = "SELECT * INTO #temp FROM @schema.person;",
  schema = "cdm_synpuf"
)
disconnect(conn)

## End(Not run)
```

---

`renderTranslateQueryApplyBatched` *Render, translate, and perform process to batches of data.*

---

## Description

This function renders, and translates SQL, sends it to the server, processes the data in batches with a call back function. Note that this function should perform a row-wise operation. This is designed to work with massive data that won't fit in to memory.

The batch sizes are determined by the java virtual machine and will depend on the data.

## Usage

```
renderTranslateQueryApplyBatched(
  connection,
  sql,
  fun,
  args = list(),
  errorReportFile = file.path(getwd(), "errorReportSql.txt"),
  snakeCaseToCamelCase = FALSE,
  tempEmulationSchema = getOption("sqlRenderTempEmulationSchema"),
 integerAsNumeric = getOption("databaseConnectorIntegerAsNumeric", default = TRUE),
 integer64AsNumeric = getOption("databaseConnectorInteger64AsNumeric", default = TRUE),
  ...
)
```

## Arguments

| | |
|---|---|
| `connection` | The connection to the database server created using either `connect()` or `DBI::dbConnect()`. |
| `sql` | The SQL to be send. |
| `fun` | Function to apply to batch. Must take data.frame and integer position as parameters. |
| `args` | List of arguments to be passed to function call. |
| `errorReportFile` | The file where an error report will be written if an error occurs. Defaults to 'errorReportSql.txt' in the current working directory. |

`snakeCaseToCamelCase`
: If true, field names are assumed to use snake_case, and are converted to camelCase.

`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.

`integerAsNumeric`
: Logical: should 32-bit integers be converted to numeric (double) values? If FALSE 32-bit integers will be represented using R's native `Integer` class.

`integer64AsNumeric`
: Logical: should 64-bit integers be converted to numeric (double) values? If FALSE 64-bit integers will be represented using `bit64::integer64`.

`...`
: Parameters that will be used to render the SQL.

## Details

Fields will be automatically converted for improved consistenty in these situations:

- SQLite: Fields with names ending in `_date` will be converted to DATE fields. Rationale: SQLite does not support DATE fields.
- SQLite: Fields with names ending in `_datetime` will be converted to POSIXct fields. Rationale: SQLite does not support DATETIME fields.
- BigQuery and Snowflake: Integer fields will be converted to Integer if it fits in an integer, or will remain Integer64 otherwise. Rationale: these platforms do not distinguish between INT and BIGINT.

This function calls the `render` and `translate` functions in the `SqlRender` package before calling `querySql()`.

## Value

Invisibly returns a list of outputs from each call to the provided function.

## Examples

```
## Not run:
connectionDetails <- createConnectionDetails(
  dbms = "postgresql",
  server = "localhost",
  user = "root",
  password = "blah",
  schema = "cdm_v4"
)
connection <- connect(connectionDetails)

# First example: write data to a large CSV file:
filepath <- "myBigFile.csv"
writeBatchesToCsv <- function(data, position, ...) {
  write.csv(data, filepath, append = position != 1)
```

```
  return(NULL)
}
renderTranslateQueryApplyBatched(connection,
  "SELECT * FROM @schema.person;",
  schema = "cdm_synpuf",
  fun = writeBatchesToCsv
)

# Second example: write data to Andromeda
# (Alternative to querySqlToAndromeda if some local computation needs to be applied)
bigResults <- Andromeda::andromeda()
writeBatchesToAndromeda <- function(data, position, ...) {
  data$p <- EmpiricalCalibration::computeTraditionalP(data$logRr, data$logSeRr)
  if (position == 1) {
    bigResults$rrs <- data
  } else {
    Andromeda::appendToTable(bigResults$rrs, data)
  }
  return(NULL)
}
sql <- "SELECT target_id, comparator_id, log_rr, log_se_rr FROM @schema.my_results;"
renderTranslateQueryApplyBatched(connection,
  sql,
  fun = writeBatchesToAndromeda,
  schema = "my_results",
  snakeCaseToCamelCase = TRUE
)

disconnect(connection)

## End(Not run)
```

---

renderTranslateQuerySql *Render, translate, and query to data.frame*

---

## Description

This function renders, and translates SQL, sends it to the server, and returns the results as a data.frame.

## Usage

```
renderTranslateQuerySql(
  connection,
  sql,
  errorReportFile = file.path(getwd(), "errorReportSql.txt"),
  snakeCaseToCamelCase = FALSE,
```

```
  tempEmulationSchema = getOption("sqlRenderTempEmulationSchema"),
 integerAsNumeric = getOption("databaseConnectorIntegerAsNumeric", default = TRUE),
 integer64AsNumeric = getOption("databaseConnectorInteger64AsNumeric", default = TRUE),
  ...
)
```

## Arguments

| | |
|---|---|
| `connection` | The connection to the database server created using either `connect()` or `DBI::dbConnect()`. |
| `sql` | The SQL to be send. |
| `errorReportFile` | The file where an error report will be written if an error occurs. Defaults to 'errorReportSql.txt' in the current working directory. |
| `snakeCaseToCamelCase` | If true, field names are assumed to use snake_case, and are converted to camelCase. |
| `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. |
| `integerAsNumeric` | Logical: should 32-bit integers be converted to numeric (double) values? If FALSE 32-bit integers will be represented using R's native `Integer` class. |
| `integer64AsNumeric` | Logical: should 64-bit integers be converted to numeric (double) values? If FALSE 64-bit integers will be represented using `bit64::integer64`. |
| `...` | Parameters that will be used to render the SQL. |

## Details

Fields will be automatically converted for improved consistenty in these situations:

- SQLite: Fields with names ending in `_date` will be converted to DATE fields. Rationale: SQLite does not support DATE fields.
- SQLite: Fields with names ending in `_datetime` will be converted to POSIXct fields. Rationale: SQLite does not support DATETIME fields.
- BigQuery and Snowflake: Integer fields will be converted to Integer if it fits in an integer, or will remain Integer64 otherwise. Rationale: these platforms do not distinguish between INT and BIGINT.

This function calls the `render` and `translate` functions in the `SqlRender` package before calling `querySql()`.

## Value

A data frame.

## Examples

```
## Not run:
connectionDetails <- createConnectionDetails(
  dbms = "postgresql",
  server = "localhost",
  user = "root",
  password = "blah",
  schema = "cdm_v4"
)
conn <- connect(connectionDetails)
persons <- renderTranslatequerySql(conn,
  sql = "SELECT TOP 10 * FROM @schema.person",
  schema = "cdm_synpuf"
)
disconnect(conn)

## End(Not run)
```

---

# renderTranslateQuerySqlToAndromeda

*Render, translate, and query to local Andromeda*

---

## Description

This function renders, and translates SQL, sends it to the server, and returns the results as an Andromeda object

## Usage

```
renderTranslateQuerySqlToAndromeda(
  connection,
  sql,
  andromeda,
  andromedaTableName,
  errorReportFile = file.path(getwd(), "errorReportSql.txt"),
  snakeCaseToCamelCase = FALSE,
  appendToTable = FALSE,
  tempEmulationSchema = getOption("sqlRenderTempEmulationSchema"),
 integerAsNumeric = getOption("databaseConnectorIntegerAsNumeric", default = TRUE),
 integer64AsNumeric = getOption("databaseConnectorInteger64AsNumeric", default = TRUE),
  ...
)
```

## Arguments

| | |
|---|---|
| connection | The connection to the database server created using either connect() or DBI::dbConnect(). |
| sql | The SQL to be send. |

andromeda — An open Andromeda object, for example as created using Andromeda::andromeda().

andromedaTableName — The name of the table in the local Andromeda object where the results of the query will be stored.

errorReportFile — The file where an error report will be written if an error occurs. Defaults to 'errorReportSql.txt' in the current working directory.

snakeCaseToCamelCase — If true, field names are assumed to use snake_case, and are converted to camelCase.

appendToTable — If FALSE, any existing table in the Andromeda with the same name will be replaced with the new data. If TRUE, data will be appended to an existing table, assuming it has the exact same structure.

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.

integerAsNumeric — Logical: should 32-bit integers be converted to numeric (double) values? If FALSE 32-bit integers will be represented using R's native Integer class.

integer64AsNumeric — Logical: should 64-bit integers be converted to numeric (double) values? If FALSE 64-bit integers will be represented using bit64::integer64.

... — Parameters that will be used to render the SQL.

## Details

This function calls the render and translate functions in the SqlRender package before calling querySqlToAndromeda().

## Value

Invisibly returns the andromeda. The Andromeda object will have a table added with the query results.

## Examples

```
## Not run:
connectionDetails <- createConnectionDetails(
  dbms = "postgresql",
  server = "localhost",
  user = "root",
  password = "blah",
  schema = "cdm_v4"
)
conn <- connect(connectionDetails)
renderTranslatequerySqlToAndromeda(conn,
  sql = "SELECT * FROM @schema.person",
```

```
  schema = "cdm_synpuf",
  andromeda = andromeda,
  andromedaTableName = "foo"
)
disconnect(conn)

andromeda$foo

## End(Not run)
```

---

`requiresTempEmulation` *Does the DBMS require temp table emulation?*

---

## Description

Does the DBMS require temp table emulation?

## Usage

```
requiresTempEmulation(dbms)
```

## Arguments

| | |
|---|---|
| dbms | The type of DBMS running on the server. See `connect()` or `createConnectionDetails()` for valid values. |

## Value

TRUE if the DBMS requires temp table emulation, FALSE otherwise.

## Examples

```
requiresTempEmulation("postgresql")
requiresTempEmulation("oracle")
```

# Index