Package 'DBItest'

May 7, 2026

Title Testing DBI Backends

Version 1.8.2

Date 2024-12-07

Description A helper that tests DBI back ends for conformity to the interface.

License LGPL (>= 2.1)

URL https://dbitest.r-dbi.org, https://github.com/r-dbi/DBItest

BugReports https://github.com/r-dbi/DBItest/issues

Depends R (>= 3.2.0)

Imports blob (>= 1.2.0), callr, DBI (>= 1.2.3), desc, hms (>= 0.5.0), lubridate, magrittr, methods, nanoarrow, palmerpenguins, rlang (>= 0.2.0), testthat (>= 2.0.0), utils, withr

Suggests clipr, constructive, debugme, devtools, knitr, lintr, pkgload, rmarkdown, RSQLite

VignetteBuilder knitr

Config/Needs/website r-dbi/dbitemplate

Config/autostyle/scope line_breaks

Config/autostyle/strict false

Config/testthat/edition 3

Config/Needs/check decor

Encoding UTF-8

KeepSource true

RoxygenNote 7.3.2.9000

Collate 'DBItest.R' 'compat-purrr.R' 'context.R' 'dbi.R' 'dummy.R' 'expectations.R' 'generics.R' 'import-dbi.R' 'import-testthat.R' 'run.R' 's4.R' 'spec-getting-started.R' 'spec-compliance-methods.R' 'spec-driver-constructor.R' 'spec-driver-data-type.R' 'spec-connection-data-type.R' 'spec-result-create-table-with-data-type.R'

'spec-driver-connect.R' 'spec-connection-disconnect.R'
'spec-result-send-query.R' 'spec-result-fetch.R'
'spec-result-roundtrip.R' 'spec-result-clear-result.R'
'spec-result-get-query.R' 'spec-result-send-statement.R'
'spec-result-execute.R' 'spec-sql-quote-string.R'
'spec-sql-quote-literal.R' 'spec-sql-quote-identifier.R'
'spec-sql-unquote-identifier.R' 'spec-sql-read-table.R'
'spec-sql-create-table.R' 'spec-sql-append-table.R'
'spec-sql-write-table.R' 'spec-sql-list-tables.R'
'spec-sql-exists-table.R' 'spec-sql-remove-table.R'
'spec-sql-list-objects.R' 'spec-meta-bind-runner.R'
'spec-meta-bind-formals.R' 'spec-meta-bind-expr.R'
'spec-meta-bind.R' 'spec-meta-bind-arrow.R'
'spec-meta-bind-stream.R' 'spec-meta-bind-arrow-stream.R'
'spec-meta-bind-.R' 'spec-meta-is-valid.R'
'spec-meta-has-completed.R' 'spec-meta-get-statement.R'
'spec-meta-get-row-count.R' 'spec-meta-get-rows-affected.R'
'spec-transaction-begin-commit-rollback.R'
'spec-transaction-with-transaction.R'
'spec-arrow-send-query-arrow.R' 'spec-arrow-fetch-arrow.R'
'spec-arrow-fetch-arrow-chunk.R' 'spec-arrow-get-query-arrow.R'
'spec-arrow-read-table-arrow.R'
'spec-arrow-write-table-arrow.R'
'spec-arrow-create-table-arrow.R'
'spec-arrow-append-table-arrow.R' 'spec-arrow-bind.R'
'spec-arrow-roundtrip.R' 'spec-driver-get-info.R'
'spec-connection-get-info.R' 'spec-sql-list-fields.R'
'spec-meta-column-info.R' 'spec-meta-get-info-result.R'
'spec-driver.R' 'spec-connection.R' 'spec-result.R'
'spec-sql.R' 'spec-meta.R' 'spec-arrow.R' 'spec-transaction.R'
'spec-compliance.R' 'spec-stress-connection.R' 'spec-stress.R'
'spec-all.R' 'spec-.R' 'test-all.R' 'test-getting-started.R'
'test-driver.R' 'test-connection.R' 'test-result.R'
'test-sql.R' 'test-meta.R' 'test-transaction.R' 'test-arrow.R'
'test-compliance.R' 'test-stress.R' 'test_backend.R' 'tweaks.R'
'utf8.R' 'utils.R' 'zzz.R'

NeedsCompilation no

Author Kirill Müller [aut, cre] (ORCID:
<https://orcid.org/0000-0002-1416-3412>),
RStudio [cph],
R Consortium [fnd]

Maintainer Kirill Müller <kirill@cynkra.com>

Repository CRAN

Date/Publication 2024-12-07 15:10:01 UTC

Contents

`DBItest-package` *DBItest: Testing DBI Backends*

Description

A helper that tests DBI back ends for conformity to the interface.

Details

The two most important functions are `make_context()` and `test_all()`. The former tells the package how to connect to your DBI backend, the latter executes all tests of the test suite. More fine-grained test functions (all with prefix `test_`) are available.

See the package's vignette for more details.

Author(s)

Kirill Müller

See Also

Useful links:

- https://dbitest.r-dbi.org
- https://github.com/r-dbi/DBItest
- Report bugs at https://github.com/r-dbi/DBItest/issues

make_context *Test contexts*

Description

Create a test context, set and query the default context.

Usage

```
make_context(
  drv,
  connect_args = NULL,
  set_as_default = TRUE,
  tweaks = NULL,
  name = NULL,
  default_skip = NULL
)

set_default_context(ctx)

get_default_context()
```

Arguments

drv	[DBIConnector] An object of class DBI::DBIConnector that describes how to connect to the database.
connect_args	[named list] Deprecated.
set_as_default	[logical(1)] Should the created context be set as default context?
tweaks	[DBItest_tweaks] Tweaks as constructed by the tweaks() function.
name	[character] An optional name of the context which will be used in test messages.
default_skip	[character] Default value of skip argument to test_all() and other testing functions.
ctx	[DBItest_context] A test context.

Value

[DBItest_context]
A test context, for set_default_context the previous default context (invisibly) or NULL.

Examples

```
make_context(
  new(
    "DBIConnector",
    .drv = RSQLite::SQLite(),
    .conn_args = list(dbname = tempfile("DBItest", fileext = ".sqlite"))
  ),
  tweaks = tweaks(
    constructor_relax_args = TRUE,
    placeholder_pattern = c("?", "$1", "$name", ":name"),
    date_cast = function(x) paste0("'", x, "'"),
    time_cast = function(x) paste0("'", x, "'"),
    timestamp_cast = function(x) paste0("'", x, "'"),
    logical_return = function(x) as.integer(x),
    date_typed = FALSE,
    time_typed = FALSE,
    timestamp_typed = FALSE
  ),
  default_skip = c("roundtrip_date", "roundtrip_timestamp")
)
```

`spec_arrow_append_table_arrow`

spec_arrow_append_table_arrow

Description

spec_arrow_append_table_arrow

Value

`dbAppendTableArrow()` returns a scalar numeric.

Failure modes

If the table does not exist, or the new data in `values` is not a data frame or has different column names, an error is raised; the remote table remains unchanged.

An error is raised when calling this method for a closed or invalid connection. An error is also raised if `name` cannot be processed with `DBI::dbQuoteIdentifier()` or if this results in a non-scalar.

Specification

SQL keywords can be used freely in table names, column names, and data. Quotes, commas, spaces, and other special characters such as newlines and tabs, can also be used in the data, and, if the database supports non-syntactic identifiers, also for table names and column names.

The following data types must be supported at least, and be read identically with `DBI::dbReadTable()`:

- integer
- numeric (the behavior for `Inf` and `NaN` is not specified)
- logical
- `NA` as NULL
- 64-bit values (using `"bigint"` as field type); the result can be
  - converted to a numeric, which may lose precision,
  - converted a character vector, which gives the full decimal representation
  - written to another table and read again unchanged
- character (in both UTF-8 and native encodings), supporting empty strings (before and after non-empty strings)
- factor (possibly returned as character)
- objects of type blob::blob (if supported by the database)
- date (if supported by the database; returned as `Date`) also for dates prior to 1970 or 1900 or after 2038
- time (if supported by the database; returned as objects that inherit from `difftime`)
- timestamp (if supported by the database; returned as `POSIXct` respecting the time zone but not necessarily preserving the input time zone), also for timestamps prior to 1970 or 1900 or after 2038 respecting the time zone but not necessarily preserving the input time zone)

Mixing column types in the same table is supported.

The `name` argument is processed as follows, to support databases that allow non-syntactic names for their objects:

- If an unquoted table name as string: `dbAppendTableArrow()` will do the quoting, perhaps by calling `dbQuoteIdentifier(conn, x = name)`
- If the result of a call to `DBI::dbQuoteIdentifier()`: no more quoting is done to support databases that allow non-syntactic names for their objects:

The `value` argument must be a data frame with a subset of the columns of the existing table. The order of the columns does not matter.

See Also

Other Arrow specifications: `spec_arrow_create_table_arrow`, `spec_arrow_fetch_arrow`, `spec_arrow_fetch_arrow_c`
`spec_arrow_get_query_arrow`, `spec_arrow_read_table_arrow`, `spec_arrow_send_query_arrow`,
`spec_arrow_write_table_arrow`, `spec_result_clear_result`

`spec_arrow_create_table_arrow` *spec_arrow_create_table_arrow*

Description

spec_arrow_create_table_arrow

Value

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

Failure modes

If the table exists, an error is raised; the remote table remains unchanged.

An error is raised when calling this method for a closed or invalid connection. An error is also raised if `name` cannot be processed with `DBI::dbQuoteIdentifier()` or if this results in a non-scalar. Invalid values for the `temporary` argument (non-scalars, unsupported data types, `NA`, incompatible values, duplicate names) also raise an error.

Additional arguments

The following arguments are not part of the `dbCreateTableArrow()` generic (to improve compatibility across backends) but are part of the DBI specification:

- `temporary` (default: `FALSE`)

They must be provided as named arguments. See the "Specification" and "Value" sections for details on their usage.

Specification

The `name` argument is processed as follows, to support databases that allow non-syntactic names for their objects:

- If an unquoted table name as string: `dbCreateTableArrow()` will do the quoting, perhaps by calling `dbQuoteIdentifier(conn, x = name)`
- If the result of a call to `DBI::dbQuoteIdentifier()`: no more quoting is done

The `value` argument can be:

- a data frame,
- a nanoarrow array
- a nanoarrow array stream (which will still contain the data after the call)
- a nanoarrow schema

If the `temporary` argument is `TRUE`, the table is not available in a second connection and is gone after reconnecting. Not all backends support this argument. A regular, non-temporary table is visible in a second connection, in a pre-existing connection, and after reconnecting to the database.

SQL keywords can be used freely in table names, column names, and data. Quotes, commas, and spaces can also be used for table names and column names, if the database supports non-syntactic identifiers.

See Also

Other Arrow specifications: `spec_arrow_append_table_arrow`, `spec_arrow_fetch_arrow`, `spec_arrow_fetch_arrow_c`
`spec_arrow_get_query_arrow`, `spec_arrow_read_table_arrow`, `spec_arrow_send_query_arrow`,
`spec_arrow_write_table_arrow`, `spec_result_clear_result`

`spec_arrow_fetch_arrow`

spec_arrow_fetch_arrow

Description

spec_arrow_fetch_arrow

Value

`dbFetchArrow()` always returns an object coercible to 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.

Failure modes

An attempt to fetch from a closed result set raises an error.

Specification

Fetching multi-row queries with one or more columns by default returns the entire result. The object returned by `dbFetchArrow()` can also be passed to `nanoarrow::as_nanoarrow_array_stream()` to create a nanoarrow array stream object that can be used to read the result set in batches. The chunk size is implementation-specific.

See Also

Other Arrow specifications: `spec_arrow_append_table_arrow`, `spec_arrow_create_table_arrow`,
`spec_arrow_fetch_arrow_chunk`, `spec_arrow_get_query_arrow`, `spec_arrow_read_table_arrow`,
`spec_arrow_send_query_arrow`, `spec_arrow_write_table_arrow`, `spec_result_clear_result`

spec_arrow_fetch_arrow_chunk *spec_arrow_fetch_arrow_chunk*

Description

spec_arrow_fetch_arrow_chunk

Value

`dbFetchArrowChunk()` always returns an object coercible to 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.

Failure modes

An attempt to fetch from a closed result set raises an error.

Specification

Fetching multi-row queries with one or more columns returns the next chunk. The size of the chunk is implementation-specific. The object returned by `dbFetchArrowChunk()` can also be passed to `nanoarrow::as_nanoarrow_array()` to create a nanoarrow array object. The chunk size is implementation-specific.

See Also

Other Arrow specifications: `spec_arrow_append_table_arrow`, `spec_arrow_create_table_arrow`, `spec_arrow_fetch_arrow`, `spec_arrow_get_query_arrow`, `spec_arrow_read_table_arrow`, `spec_arrow_send_query_`
`spec_arrow_write_table_arrow`, `spec_result_clear_result`

spec_arrow_get_query_arrow *spec_arrow_get_query_arrow*

Description

spec_arrow_get_query_arrow

Value

`dbGetQueryArrow()` always returns an object coercible to 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.

Failure modes

An error is raised when issuing a query over a closed or invalid connection, if the syntax of the query is invalid, or if the query is not a non-NA string. The object returned by `dbGetQueryArrow()` can also be passed to `nanoarrow::as_nanoarrow_array_stream()` to create a nanoarrow array stream object that can be used to read the result set in batches. The chunk size is implementation-specific.

Additional arguments

The following arguments are not part of the `dbGetQueryArrow()` generic (to improve compatibility across backends) but are part of the DBI specification:

- `params` (default: `NULL`)
- `immediate` (default: `NULL`)

They must be provided as named arguments. See the "Specification" and "Value" sections for details on their usage.

The `param` argument allows passing query parameters, see `DBI::dbBind()` for details.

Specification for the `immediate` argument

The `immediate` argument supports distinguishing between "direct" and "prepared" APIs offered by many database drivers. Passing `immediate = TRUE` leads to immediate execution of the query or statement, via the "direct" API (if supported by the driver). The default `NULL` means that the backend should choose whatever API makes the most sense for the database, and (if relevant) tries the other API if the first attempt fails. A successful second attempt should result in a message that suggests passing the correct `immediate` argument. Examples for possible behaviors:

1. DBI backend defaults to `immediate = TRUE` internally
   (a) A query without parameters is passed: query is executed
   (b) A query with parameters is passed:
      i. `params` not given: rejected immediately by the database because of a syntax error in the query, the backend tries `immediate = FALSE` (and gives a message)
      ii. `params` given: query is executed using `immediate = FALSE`
2. DBI backend defaults to `immediate = FALSE` internally
   (a) A query without parameters is passed:
      i. simple query: query is executed
      ii. "special" query (such as setting a config options): fails, the backend tries `immediate = TRUE` (and gives a message)
   (b) A query with parameters is passed:
      i. `params` not given: waiting for parameters via `DBI::dbBind()`
      ii. `params` given: query is executed

See Also

Other Arrow specifications: `spec_arrow_append_table_arrow`, `spec_arrow_create_table_arrow`, `spec_arrow_fetch_arrow`, `spec_arrow_fetch_arrow_chunk`, `spec_arrow_read_table_arrow`, `spec_arrow_send_query_arrow`, `spec_arrow_write_table_arrow`, `spec_result_clear_result`

spec_arrow_read_table_arrow *spec_arrow_read_table_arrow*

Description

spec_arrow_read_table_arrow

Value

`dbReadTableArrow()` returns an Arrow object that contains the complete data from the remote table, effectively the result of calling `DBI::dbGetQueryArrow()` with `SELECT * FROM <name>`.

An empty table is returned as an Arrow object with zero rows.

Failure modes

An error is raised if the table does not exist.

An error is raised when calling this method for a closed or invalid connection. An error is raised if `name` cannot be processed with `DBI::dbQuoteIdentifier()` or if this results in a non-scalar.

Specification

The `name` argument is processed as follows, to support databases that allow non-syntactic names for their objects:

- If an unquoted table name as string: `dbReadTableArrow()` will do the quoting, perhaps by calling `dbQuoteIdentifier(conn, x = name)`
- If the result of a call to `DBI::dbQuoteIdentifier()`: no more quoting is done

See Also

Other Arrow specifications: `spec_arrow_append_table_arrow`, `spec_arrow_create_table_arrow`, `spec_arrow_fetch_arrow`, `spec_arrow_fetch_arrow_chunk`, `spec_arrow_get_query_arrow`, `spec_arrow_send_query_arrow`, `spec_arrow_write_table_arrow`, `spec_result_clear_result`

spec_arrow_send_query_arrow *spec_result_send_query*

Description

spec_result_send_query

Value

`dbSendQueryArrow()` returns an S4 object that inherits from DBI::DBIResultArrow. The result set can be used with `DBI::dbFetchArrow()` to extract records. Once you have finished using a result, make sure to clear it with `DBI::dbClearResult()`.

Failure modes

An error is raised when issuing a query over a closed or invalid connection, or if the query is not a non-`NA` string. An error is also raised if the syntax of the query is invalid and all query parameters are given (by passing the `params` argument) or the `immediate` argument is set to `TRUE`.

Additional arguments

The following arguments are not part of the `dbSendQueryArrow()` generic (to improve compatibility across backends) but are part of the DBI specification:

- `params` (default: `NULL`)
- `immediate` (default: `NULL`)

They must be provided as named arguments. See the "Specification" sections for details on their usage.

Specification

No warnings occur under normal conditions. When done, the DBIResult object must be cleared with a call to `DBI::dbClearResult()`. Failure to clear the result set leads to a warning when the connection is closed.

If the backend supports only one open result set per connection, issuing a second query invalidates an already open result set and raises a warning. The newly opened result set is valid and must be cleared with `dbClearResult()`.

The `param` argument allows passing query parameters, see `DBI::dbBind()` for details.

Specification for the `immediate` argument

The `immediate` argument supports distinguishing between "direct" and "prepared" APIs offered by many database drivers. Passing `immediate = TRUE` leads to immediate execution of the query or statement, via the "direct" API (if supported by the driver). The default `NULL` means that the backend should choose whatever API makes the most sense for the database, and (if relevant) tries the other API if the first attempt fails. A successful second attempt should result in a message that suggests passing the correct `immediate` argument. Examples for possible behaviors:

1. DBI backend defaults to `immediate = TRUE` internally
   (a) A query without parameters is passed: query is executed
   (b) A query with parameters is passed:
      i. `params` not given: rejected immediately by the database because of a syntax error in the query, the backend tries `immediate = FALSE` (and gives a message)
      ii. `params` given: query is executed using `immediate = FALSE`
2. DBI backend defaults to `immediate = FALSE` internally

(a) A query without parameters is passed:
   i. simple query: query is executed
   ii. "special" query (such as setting a config options): fails, the backend tries `immediate = TRUE` (and gives a message)

(b) A query with parameters is passed:
   i. `params` not given: waiting for parameters via `DBI::dbBind()`
   ii. `params` given: query is executed

See Also

Other Arrow specifications: `spec_arrow_append_table_arrow`, `spec_arrow_create_table_arrow`, `spec_arrow_fetch_arrow`, `spec_arrow_fetch_arrow_chunk`, `spec_arrow_get_query_arrow`, `spec_arrow_read_table_arrow`, `spec_arrow_write_table_arrow`, `spec_result_clear_result`

`spec_arrow_write_table_arrow`
spec_arrow_write_table_arrow

Description

spec_arrow_write_table_arrow

Value

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

Failure modes

If the table exists, and both `append` and `overwrite` arguments are unset, or `append = TRUE` and the data frame with the new data has different column names, an error is raised; the remote table remains unchanged.

An error is raised when calling this method for a closed or invalid connection. An error is also raised if `name` cannot be processed with `DBI::dbQuoteIdentifier()` or if this results in a non-scalar. Invalid values for the additional arguments `overwrite`, `append`, and `temporary` (non-scalars, unsupported data types, `NA`, incompatible values, incompatible columns) also raise an error.

Additional arguments

The following arguments are not part of the `dbWriteTableArrow()` generic (to improve compatibility across backends) but are part of the DBI specification:

- `overwrite` (default: `FALSE`)
- `append` (default: `FALSE`)
- `temporary` (default: `FALSE`)

They must be provided as named arguments. See the "Specification" and "Value" sections for details on their usage.

Specification

The name argument is processed as follows, to support databases that allow non-syntactic names for their objects:

- If an unquoted table name as string: dbWriteTableArrow() will do the quoting, perhaps by calling dbQuoteIdentifier(conn, x = name)
- If the result of a call to DBI::dbQuoteIdentifier(): no more quoting is done

The value argument must be a data frame with a subset of the columns of the existing table if append = TRUE. The order of the columns does not matter with append = TRUE.

If the overwrite argument is TRUE, an existing table of the same name will be overwritten. This argument doesn't change behavior if the table does not exist yet.

If the append argument is TRUE, the rows in an existing table are preserved, and the new data are appended. If the table doesn't exist yet, it is created.

If the temporary argument is TRUE, the table is not available in a second connection and is gone after reconnecting. Not all backends support this argument. A regular, non-temporary table is visible in a second connection, in a pre-existing connection, and after reconnecting to the database.

SQL keywords can be used freely in table names, column names, and data. Quotes, commas, spaces, and other special characters such as newlines and tabs, can also be used in the data, and, if the database supports non-syntactic identifiers, also for table names and column names.

The following data types must be supported at least, and be read identically with DBI::dbReadTable():

- integer
- numeric (the behavior for Inf and NaN is not specified)
- logical
- NA as NULL
- 64-bit values (using "bigint" as field type); the result can be
  - converted to a numeric, which may lose precision,
  - converted a character vector, which gives the full decimal representation
  - written to another table and read again unchanged
- character (in both UTF-8 and native encodings), supporting empty strings before and after a non-empty string
- factor (possibly returned as character)
- objects of type blob::blob (if supported by the database)
- date (if supported by the database; returned as Date), also for dates prior to 1970 or 1900 or after 2038
- time (if supported by the database; returned as objects that inherit from difftime)
- timestamp (if supported by the database; returned as POSIXct respecting the time zone but not necessarily preserving the input time zone), also for timestamps prior to 1970 or 1900 or after 2038 respecting the time zone but not necessarily preserving the input time zone)

Mixing column types in the same table is supported.

See Also

Other Arrow specifications: spec_arrow_append_table_arrow, spec_arrow_create_table_arrow, spec_arrow_fetch_arrow, spec_arrow_fetch_arrow_chunk, spec_arrow_get_query_arrow, spec_arrow_read_table_arrow, spec_arrow_send_query_arrow, spec_result_clear_result

---

`spec_compliance_methods`

spec_compliance_methods

Description

spec_compliance_methods

DBI classes and methods

A backend defines three classes, which are subclasses of DBI::DBIDriver, DBI::DBIConnection, and DBI::DBIResult. The backend provides implementation for all methods of these base classes that are defined but not implemented by DBI. All methods defined in **DBI** are reexported (so that the package can be used without having to attach **DBI**), and have an ellipsis . . . in their formals for extensibility.

`spec_connection_disconnect`

spec_connection_disconnect

Description

spec_connection_disconnect

Value

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

Failure modes

A warning is issued on garbage collection when a connection has been released without calling `dbDisconnect()`, but this cannot be tested automatically. At least one warning is issued immediately when calling `dbDisconnect()` on an already disconnected or invalid connection.

See Also

Other connection specifications: spec_get_info

spec_driver_connect *spec_driver_connect*

Description

spec_driver_connect

Value

`dbConnect()` returns an S4 object that inherits from DBI::DBIConnection. This object is used to communicate with the database engine.

A `format()` method is defined for the connection object. It returns a string that consists of a single line of text.

Specification

DBI recommends using the following argument names for authentication parameters, with `NULL` default:

- `user` for the user name (default: current user)
- `password` for the password
- `host` for the host name (default: local connection)
- `port` for the port number (default: local connection)
- `dbname` for the name of the database on the host, or the database file name

The defaults should provide reasonable behavior, in particular a local connection for `host = NULL`. For some DBMS (e.g., PostgreSQL), this is different to a TCP/IP connection to `localhost`.

In addition, DBI supports the `bigint` argument that governs how 64-bit integer data is returned. The following values are supported:

- `"integer"`: always return as `integer`, silently overflow
- `"numeric"`: always return as `numeric`, silently round
- `"character"`: always return the decimal representation as `character`
- `"integer64"`: return as a data type that can be coerced using `as.integer()` (with warning on overflow), `as.numeric()` and `as.character()`

See Also

Other driver specifications: `spec_driver_constructor`, `spec_driver_data_type`, `spec_get_info`

`spec_driver_constructor` *spec_driver_constructor*

Description

spec_driver_constructor

Construction of the DBIDriver object

The backend must support creation of an instance of its DBI::DBIDriver subclass with a *constructor function*. By default, its name is the package name without the leading 'R' (if it exists), e.g., `SQLite` for the **RSQLite** package. However, backend authors may choose a different name. The constructor must be exported, and it must be a function that is callable without arguments. DBI recommends to define a constructor with an empty argument list.

See Also

Other driver specifications: `spec_driver_connect`, `spec_driver_data_type`, `spec_get_info`

`spec_driver_data_type` *spec_driver_data_type*

Description

spec_driver_data_type

Value

`dbDataType()` returns the SQL type that corresponds to the `obj` argument as a non-empty character string. For data frames, a character vector with one element per column is returned.

Failure modes

An error is raised for invalid values for the `obj` argument such as a `NULL` value.

Specification

The backend can override the `DBI::dbDataType()` generic for its driver class.

This generic expects an arbitrary object as second argument. To query the values returned by the default implementation, run `example(dbDataType, package = "DBI")`. If the backend needs to override this generic, it must accept all basic R data types as its second argument, namely logical, integer, numeric, character, dates (see Dates), date-time (see DateTimeClasses), and difftime. If the database supports blobs, this method also must accept lists of raw vectors, and blob::blob objects. As-is objects (i.e., wrapped by `I()`) must be supported and return the same results as their unwrapped counterparts. The SQL data type for factor and ordered is the same as for character. The behavior for other object types is not specified.

See Also

Other driver specifications: spec_driver_connect, spec_driver_constructor, spec_get_info

---

spec_getting_started *spec_getting_started*

Description

spec_getting_started

Definition

A DBI backend is an R package which imports the **DBI** and **methods** packages. For better or worse, the names of many existing backends start with 'R', e.g., **RSQLite**, **RMySQL**, **RSQLServer**; it is up to the backend author to adopt this convention or not.

---

spec_get_info *spec_driver_get_info*

Description

spec_driver_get_info

spec_connection_get_info

spec_meta_get_info_result

Value

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

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

See Also

Other driver specifications: spec_driver_connect, spec_driver_constructor, spec_driver_data_type

Other connection specifications: spec_connection_disconnect

Other meta specifications: spec_meta_bind, spec_meta_column_info, spec_meta_get_row_count, spec_meta_get_rows_affected, spec_meta_get_statement, spec_meta_has_completed, spec_meta_is_valid

---

spec_meta_bind *spec_meta_bind*

Description

spec_meta_bind

spec_meta_bind

spec_meta_bind

Value

dbBind() returns the result set, invisibly, for queries issued by DBI::dbSendQuery() or DBI::dbSendQueryArrow() and also for data manipulation statements issued by DBI::dbSendStatement().

Specification

DBI clients execute parametrized statements as follows:

1. Call DBI::dbSendQuery(), DBI::dbSendQueryArrow() or DBI::dbSendStatement() with a query or statement that contains placeholders, store the returned DBI::DBIResult object in a variable. Mixing placeholders (in particular, named and unnamed ones) is not recommended. It is good practice to register a call to DBI::dbClearResult() via on.exit() right after calling dbSendQuery() or dbSendStatement() (see the last enumeration item). Until DBI::dbBind() or DBI::dbBindArrow() have been called, the returned result set object has the following behavior:
   - DBI::dbFetch() raises an error (for dbSendQuery() and dbSendQueryArrow())
   - DBI::dbGetRowCount() returns zero (for dbSendQuery() and dbSendQueryArrow())
   - DBI::dbGetRowsAffected() returns an integer NA (for dbSendStatement())

- DBI::dbIsValid() returns TRUE
- DBI::dbHasCompleted() returns FALSE

2. Call DBI::dbBind() or DBI::dbBindArrow():
   - For DBI::dbBind(), the params argument must be a list where all elements have the same lengths and contain values supported by the backend. A data.frame is internally stored as such a list.
   - For DBI::dbBindArrow(), the params argument must be a nanoarrow array stream, with one column per query parameter.
3. Retrieve the data or the number of affected rows from the DBIResult object.
   - For queries issued by dbSendQuery() or dbSendQueryArrow(), call DBI::dbFetch().
   - For statements issued by dbSendStatements(), call DBI::dbGetRowsAffected(). (Execution begins immediately after the DBI::dbBind() call, the statement is processed entirely before the function returns.)
4. Repeat 2. and 3. as necessary.
5. Close the result set via DBI::dbClearResult().

The elements of the params argument do not need to be scalars, vectors of arbitrary length (including length 0) are supported. For queries, calling dbFetch() binding such parameters returns concatenated results, equivalent to binding and fetching for each set of values and connecting via rbind(). For data manipulation statements, dbGetRowsAffected() returns the total number of rows affected if binding non-scalar parameters. dbBind() also accepts repeated calls on the same result set for both queries and data manipulation statements, even if no results are fetched between calls to dbBind(), for both queries and data manipulation statements.

If the placeholders in the query are named, their order in the params argument is not important.

At least the following data types are accepted on input (including NA):

- integer
- numeric
- logical for Boolean values
- character (also with special characters such as spaces, newlines, quotes, and backslashes)
- factor (bound as character, with warning)
- lubridate::Date (also when stored internally as integer)
- lubridate::POSIXct timestamps
- POSIXlt timestamps
- difftime values (also with units other than seconds and with the value stored as integer)
- lists of raw for blobs (with NULL entries for SQL NULL values)
- objects of type blob::blob

Failure modes

Calling dbBind() for a query without parameters raises an error.

Binding too many or not enough values, or parameters with wrong names or unequal length, also raises an error. If the placeholders in the query are named, all parameter values must have names

(which must not be empty or NA), and vice versa, otherwise an error is raised. The behavior for mixing placeholders of different types (in particular mixing positional and named placeholders) is not specified.

Calling dbBind() on a result set already cleared by DBI::dbClearResult() also raises an error.

See Also

Other meta specifications: spec_get_info, spec_meta_column_info, spec_meta_get_row_count, spec_meta_get_rows_affected, spec_meta_get_statement, spec_meta_has_completed, spec_meta_is_valid

Other meta specifications: spec_get_info, spec_meta_column_info, spec_meta_get_row_count, spec_meta_get_rows_affected, spec_meta_get_statement, spec_meta_has_completed, spec_meta_is_valid

Other meta specifications: spec_get_info, spec_meta_column_info, spec_meta_get_row_count, spec_meta_get_rows_affected, spec_meta_get_statement, spec_meta_has_completed, spec_meta_is_valid

---

spec_meta_column_info *spec_meta_column_info*

Description

spec_meta_column_info

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 DBI::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.

Failure modes

An attempt to query columns for a closed result set raises an error.

Specification

A column named row_names is treated like any other column.

The column names are always consistent with the data returned by dbFetch().

If the query returns unnamed columns, non-empty and non-NA names are assigned.

Column names that correspond to SQL or R keywords are left unchanged.

See Also

Other meta specifications: spec_get_info, spec_meta_bind, spec_meta_get_row_count, spec_meta_get_rows_affect
spec_meta_get_statement, spec_meta_has_completed, spec_meta_is_valid

spec_meta_get_rows_affected *spec_meta_get_rows_affected*

Description

spec_meta_get_rows_affected

Value

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

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

Failure modes

Attempting to get the rows affected for a result set cleared with `DBI::dbClearResult()` gives an error.

See Also

Other meta specifications: `spec_get_info`, `spec_meta_bind`, `spec_meta_column_info`, `spec_meta_get_row_count`, `spec_meta_get_statement`, `spec_meta_has_completed`, `spec_meta_is_valid`

spec_meta_get_row_count *spec_meta_get_row_count*

Description

spec_meta_get_row_count

Value

`dbGetRowCount()` returns a scalar number (integer or numeric), the number of rows fetched so far. After calling `DBI::dbSendQuery()`, the row count is initially zero. After a call to `DBI::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 `DBI::dbSendStatement()`, zero is returned before and after calling `dbFetch()`.

Failure modes

Attempting to get the row count for a result set cleared with DBI::dbClearResult() gives an error.

See Also

Other meta specifications: spec_get_info, spec_meta_bind, spec_meta_column_info, spec_meta_get_rows_affected, spec_meta_get_statement, spec_meta_has_completed, spec_meta_is_valid

---

spec_meta_get_statement

spec_meta_get_statement

Description

spec_meta_get_statement

Value

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

Failure modes

Attempting to query the statement for a result set cleared with DBI::dbClearResult() gives an error.

See Also

Other meta specifications: spec_get_info, spec_meta_bind, spec_meta_column_info, spec_meta_get_row_count, spec_meta_get_rows_affected, spec_meta_has_completed, spec_meta_is_valid

---

spec_meta_has_completed

spec_meta_has_completed

Description

spec_meta_has_completed

Value

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

Failure modes

Attempting to query completion status for a result set cleared with DBI::dbClearResult() gives an error.

Specification

The completion status for a query is only guaranteed to be set to FALSE after attempting to fetch past the end of the entire result. Therefore, for a query with an empty result set, the initial return value is unspecified, but the result value is TRUE after trying to fetch only one row.

Similarly, for a query with a result set of length n, the return value is unspecified after fetching n rows, but the result value is TRUE after trying to fetch only one more row.

See Also

Other meta specifications: spec_get_info, spec_meta_bind, spec_meta_column_info, spec_meta_get_row_count, spec_meta_get_rows_affected, spec_meta_get_statement, spec_meta_is_valid

---

spec_meta_is_valid *spec_meta_is_valid*

Description

spec_meta_is_valid

Value

dbIsValid() returns a logical scalar, TRUE if the object specified by dbObj is valid, FALSE otherwise. A DBI::DBIConnection object is initially valid, and becomes invalid after disconnecting with DBI::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 DBI::DBIResult object is valid after a call to DBI::dbSendQuery(), and stays valid even after all rows have been fetched; only clearing it with DBI::dbClearResult() invalidates it. A DBI::DBIResult object is also valid after a call to DBI::dbSendStatement(), and stays valid after querying the number of rows affected; only clearing it with DBI::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 meta specifications: spec_get_info, spec_meta_bind, spec_meta_column_info, spec_meta_get_row_count, spec_meta_get_rows_affected, spec_meta_get_statement, spec_meta_has_completed

---

spec_result_clear_result

spec_result_clear_result

Description

spec_result_clear_result

Value

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

Failure modes

An attempt to close an already closed result set issues a warning for `dbSendQuery()`, `dbSendStatement()`, and `dbSendQueryArrow()`,

Specification

`dbClearResult()` frees all resources associated with retrieving the result of a query or update operation. The DBI backend can expect a call to `dbClearResult()` for each `DBI::dbSendQuery()` or `DBI::dbSendStatement()` call.

See Also

Other result specifications: `spec_result_create_table_with_data_type`, `spec_result_execute`, `spec_result_fetch`, `spec_result_get_query`, `spec_result_roundtrip`, `spec_result_send_query`, `spec_result_send_statement`

Other Arrow specifications: `spec_arrow_append_table_arrow`, `spec_arrow_create_table_arrow`, `spec_arrow_fetch_arrow`, `spec_arrow_fetch_arrow_chunk`, `spec_arrow_get_query_arrow`, `spec_arrow_read_table_arrow`, `spec_arrow_send_query_arrow`, `spec_arrow_write_table_arrow`

---

spec_result_create_table_with_data_type

spec_result_create_table_with_data_type

Description

spec_result_create_table_with_data_type

Specification

All data types returned by `dbDataType()` are usable in an SQL statement of the form `"CREATE TABLE test (a ...)"`.

See Also

Other result specifications: spec_result_clear_result, spec_result_execute, spec_result_fetch, spec_result_get_query, spec_result_roundtrip, spec_result_send_query, spec_result_send_statement

---

`spec_result_execute` *spec_result_execute*

Description

spec_result_execute

Value

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

Failure modes

An error is raised when issuing a statement over a closed or invalid connection, if the syntax of the statement is invalid, or if the statement is not a non-NA string.

Additional arguments

The following arguments are not part of the `dbExecute()` generic (to improve compatibility across backends) but are part of the DBI specification:

- `params` (default: `NULL`)
- `immediate` (default: `NULL`)

They must be provided as named arguments. See the "Specification" sections for details on their usage.

Specification

The `param` argument allows passing query parameters, see `DBI::dbBind()` for details.

Specification for the `immediate` argument

The `immediate` argument supports distinguishing between "direct" and "prepared" APIs offered by many database drivers. Passing `immediate = TRUE` leads to immediate execution of the query or statement, via the "direct" API (if supported by the driver). The default `NULL` means that the backend should choose whatever API makes the most sense for the database, and (if relevant) tries the other API if the first attempt fails. A successful second attempt should result in a message that suggests passing the correct `immediate` argument. Examples for possible behaviors:

1. DBI backend defaults to `immediate = TRUE` internally
   (a) A query without parameters is passed: query is executed
   (b) A query with parameters is passed:

i. `params` not given: rejected immediately by the database because of a syntax error in the query, the backend tries `immediate = FALSE` (and gives a message)

ii. `params` given: query is executed using `immediate = FALSE`

2. DBI backend defaults to `immediate = FALSE` internally

(a) A query without parameters is passed:

i. simple query: query is executed

ii. "special" query (such as setting a config options): fails, the backend tries `immediate = TRUE` (and gives a message)

(b) A query with parameters is passed:

i. `params` not given: waiting for parameters via `DBI::dbBind()`

ii. `params` given: query is executed

See Also

Other result specifications: `spec_result_clear_result`, `spec_result_create_table_with_data_type`, `spec_result_fetch`, `spec_result_get_query`, `spec_result_roundtrip`, `spec_result_send_query`, `spec_result_send_statement`

`spec_result_fetch` *spec_result_fetch*

Description

spec_result_fetch

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.

Failure modes

An attempt to fetch from a closed result set raises an error. If the `n` argument is not an atomic whole number greater or equal to -1 or Inf, an error is raised, but a subsequent call to `dbFetch()` with proper `n` argument succeeds.

Calling `dbFetch()` on a result set from a data manipulation query created by `DBI::dbSendStatement()` can be fetched and return an empty data frame, with a warning.

Specification

Fetching multi-row queries with one or more columns by default returns the entire result. Multi-row queries can also be fetched progressively by passing a whole number (integer or numeric) as the n argument. A value of Inf for the n argument is supported and also returns the full result. If more rows than available are fetched, the result is returned in full without warning. If fewer rows than requested are returned, further fetches will return a data frame with zero rows. If zero rows are fetched, the columns of the data frame are still fully typed. Fetching fewer rows than available is permitted, no warning is issued when clearing the result set.

A column named row_names is treated like any other column.

See Also

Other result specifications: spec_result_clear_result, spec_result_create_table_with_data_type, spec_result_execute, spec_result_get_query, spec_result_roundtrip, spec_result_send_query, spec_result_send_statement

---

spec_result_get_query *spec_result_get_query*

Description

spec_result_get_query

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.

Failure modes

An error is raised when issuing a query over a closed or invalid connection, if the syntax of the query is invalid, or if the query is not a non-NA string. If the n argument is not an atomic whole number greater or equal to -1 or Inf, an error is raised, but a subsequent call to dbGetQuery() with proper n argument succeeds.

Additional arguments

The following arguments are not part of the dbGetQuery() generic (to improve compatibility across backends) but are part of the DBI specification:

- n (default: -1)
- params (default: NULL)
- immediate (default: NULL)

They must be provided as named arguments. See the "Specification" and "Value" sections for details on their usage.

Specification

A column named `row_names` is treated like any other column.

The `n` argument specifies the number of rows to be fetched. If omitted, fetching multi-row queries with one or more columns returns the entire result. A value of Inf for the `n` argument is supported and also returns the full result. If more rows than available are fetched (by passing a too large value for `n`), the result is returned in full without warning. If zero rows are requested, the columns of the data frame are still fully typed. Fetching fewer rows than available is permitted, no warning is issued.

The `param` argument allows passing query parameters, see `DBI::dbBind()` for details.

Specification for the `immediate` argument

The `immediate` argument supports distinguishing between "direct" and "prepared" APIs offered by many database drivers. Passing `immediate = TRUE` leads to immediate execution of the query or statement, via the "direct" API (if supported by the driver). The default `NULL` means that the backend should choose whatever API makes the most sense for the database, and (if relevant) tries the other API if the first attempt fails. A successful second attempt should result in a message that suggests passing the correct `immediate` argument. Examples for possible behaviors:

1. DBI backend defaults to `immediate = TRUE` internally
   (a) A query without parameters is passed: query is executed
   (b) A query with parameters is passed:
      i. `params` not given: rejected immediately by the database because of a syntax error in the query, the backend tries `immediate = FALSE` (and gives a message)
      ii. `params` given: query is executed using `immediate = FALSE`
2. DBI backend defaults to `immediate = FALSE` internally
   (a) A query without parameters is passed:
      i. simple query: query is executed
      ii. "special" query (such as setting a config options): fails, the backend tries `immediate = TRUE` (and gives a message)
   (b) A query with parameters is passed:
      i. `params` not given: waiting for parameters via `DBI::dbBind()`
      ii. `params` given: query is executed

See Also

Other result specifications: `spec_result_clear_result`, `spec_result_create_table_with_data_type`, `spec_result_execute`, `spec_result_fetch`, `spec_result_roundtrip`, `spec_result_send_query`, `spec_result_send_statement`

spec_result_roundtrip *spec_result_roundtrip*

Description

spec_result_roundtrip

Specification

The column types of the returned data frame depend on the data returned:

- integer (or coercible to an integer) for integer values between -2^31 and 2^31 - 1, with NA for SQL NULL values
- numeric for numbers with a fractional component, with NA for SQL NULL values
- logical for Boolean values (some backends may return an integer); with NA for SQL NULL values
- character for text, with NA for SQL NULL values
- lists of raw for blobs with NULL entries for SQL NULL values
- coercible using `as.Date()` for dates, with NA for SQL NULL values (also applies to the return value of the SQL function `current_date`)
- coercible using `hms::as_hms()` for times, with NA for SQL NULL values (also applies to the return value of the SQL function `current_time`)
- coercible using `as.POSIXct()` for timestamps, with NA for SQL NULL values (also applies to the return value of the SQL function `current_timestamp`)

If dates and timestamps are supported by the backend, the following R types are used:

- lubridate::Date for dates (also applies to the return value of the SQL function `current_date`)
- lubridate::POSIXct for timestamps (also applies to the return value of the SQL function `current_timestamp`)

R has no built-in type with lossless support for the full range of 64-bit or larger integers. If 64-bit integers are returned from a query, the following rules apply:

- Values are returned in a container with support for the full range of valid 64-bit values (such as the `integer64` class of the **bit64** package)
- Coercion to numeric always returns a number that is as close as possible to the true value
- Loss of precision when converting to numeric gives a warning
- Conversion to character always returns a lossless decimal representation of the data

See Also

Other result specifications: `spec_result_clear_result`, `spec_result_create_table_with_data_type`, `spec_result_execute`, `spec_result_fetch`, `spec_result_get_query`, `spec_result_send_query`, `spec_result_send_statement`

spec_result_send_query *spec_result_send_query*

Description

spec_result_send_query

Value

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

Failure modes

An error is raised when issuing a query over a closed or invalid connection, or if the query is not a non-`NA` string. An error is also raised if the syntax of the query is invalid and all query parameters are given (by passing the `params` argument) or the `immediate` argument is set to `TRUE`.

Additional arguments

The following arguments are not part of the `dbSendQuery()` generic (to improve compatibility across backends) but are part of the DBI specification:

- `params` (default: `NULL`)
- `immediate` (default: `NULL`)

They must be provided as named arguments. See the "Specification" sections for details on their usage.

Specification

No warnings occur under normal conditions. When done, the DBIResult object must be cleared with a call to `DBI::dbClearResult()`. Failure to clear the result set leads to a warning when the connection is closed.

If the backend supports only one open result set per connection, issuing a second query invalidates an already open result set and raises a warning. The newly opened result set is valid and must be cleared with `dbClearResult()`.

The `param` argument allows passing query parameters, see `DBI::dbBind()` for details.

Specification for the `immediate` argument

The `immediate` argument supports distinguishing between "direct" and "prepared" APIs offered by many database drivers. Passing `immediate = TRUE` leads to immediate execution of the query or statement, via the "direct" API (if supported by the driver). The default `NULL` means that the backend should choose whatever API makes the most sense for the database, and (if relevant) tries

the other API if the first attempt fails. A successful second attempt should result in a message that suggests passing the correct `immediate` argument. Examples for possible behaviors:

1. DBI backend defaults to `immediate = TRUE` internally
   (a) A query without parameters is passed: query is executed
   (b) A query with parameters is passed:
      i. `params` not given: rejected immediately by the database because of a syntax error in the query, the backend tries `immediate = FALSE` (and gives a message)
      ii. `params` given: query is executed using `immediate = FALSE`
2. DBI backend defaults to `immediate = FALSE` internally
   (a) A query without parameters is passed:
      i. simple query: query is executed
      ii. "special" query (such as setting a config options): fails, the backend tries `immediate = TRUE` (and gives a message)
   (b) A query with parameters is passed:
      i. `params` not given: waiting for parameters via `DBI::dbBind()`
      ii. `params` given: query is executed

See Also

Other result specifications: `spec_result_clear_result`, `spec_result_create_table_with_data_type`, `spec_result_execute`, `spec_result_fetch`, `spec_result_get_query`, `spec_result_roundtrip`, `spec_result_send_statement`

`spec_result_send_statement`

spec_result_send_statement

Description

spec_result_send_statement

Value

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

Failure modes

An error is raised when issuing a statement over a closed or invalid connection, or if the statement is not a non-`NA` string. An error is also raised if the syntax of the query is invalid and all query parameters are given (by passing the `params` argument) or the `immediate` argument is set to `TRUE`.

Specification

No warnings occur under normal conditions. When done, the DBIResult object must be cleared with a call to `DBI::dbClearResult()`. Failure to clear the result set leads to a warning when the connection is closed. If the backend supports only one open result set per connection, issuing a second query invalidates an already open result set and raises a warning. The newly opened result set is valid and must be cleared with `dbClearResult()`.

The `param` argument allows passing query parameters, see `DBI::dbBind()` for details.

Additional arguments

The following arguments are not part of the `dbSendStatement()` generic (to improve compatibility across backends) but are part of the DBI specification:

- `params` (default: `NULL`)
- `immediate` (default: `NULL`)

They must be provided as named arguments. See the "Specification" sections for details on their usage.

Specification for the `immediate` argument

The `immediate` argument supports distinguishing between "direct" and "prepared" APIs offered by many database drivers. Passing `immediate = TRUE` leads to immediate execution of the query or statement, via the "direct" API (if supported by the driver). The default `NULL` means that the backend should choose whatever API makes the most sense for the database, and (if relevant) tries the other API if the first attempt fails. A successful second attempt should result in a message that suggests passing the correct `immediate` argument. Examples for possible behaviors:

1. DBI backend defaults to `immediate = TRUE` internally
   (a) A query without parameters is passed: query is executed
   (b) A query with parameters is passed:
      i. `params` not given: rejected immediately by the database because of a syntax error in the query, the backend tries `immediate = FALSE` (and gives a message)
      ii. `params` given: query is executed using `immediate = FALSE`
2. DBI backend defaults to `immediate = FALSE` internally
   (a) A query without parameters is passed:
      i. simple query: query is executed
      ii. "special" query (such as setting a config options): fails, the backend tries `immediate = TRUE` (and gives a message)
   (b) A query with parameters is passed:
      i. `params` not given: waiting for parameters via `DBI::dbBind()`
      ii. `params` given: query is executed

See Also

Other result specifications: `spec_result_clear_result`, `spec_result_create_table_with_data_type`, `spec_result_execute`, `spec_result_fetch`, `spec_result_get_query`, `spec_result_roundtrip`, `spec_result_send_query`

spec_sql_append_table *spec_sql_append_table*

Description

spec_sql_append_table

Value

`dbAppendTable()` returns a scalar numeric.

Failure modes

If the table does not exist, or the new data in `values` is not a data frame or has different column names, an error is raised; the remote table remains unchanged.

An error is raised when calling this method for a closed or invalid connection. An error is also raised if `name` cannot be processed with `DBI::dbQuoteIdentifier()` or if this results in a non-scalar. Invalid values for the `row.names` argument (non-scalars, unsupported data types, `NA`) also raise an error.

Passing a `value` argument different to `NULL` to the `row.names` argument (in particular `TRUE`, `NA`, and a string) raises an error.

Specification

SQL keywords can be used freely in table names, column names, and data. Quotes, commas, spaces, and other special characters such as newlines and tabs, can also be used in the data, and, if the database supports non-syntactic identifiers, also for table names and column names.

The following data types must be supported at least, and be read identically with `DBI::dbReadTable()`:

- integer
- numeric (the behavior for `Inf` and `NaN` is not specified)
- logical
- `NA` as NULL
- 64-bit values (using `"bigint"` as field type); the result can be
  - converted to a numeric, which may lose precision,
  - converted a character vector, which gives the full decimal representation
  - written to another table and read again unchanged
- character (in both UTF-8 and native encodings), supporting empty strings (before and after non-empty strings)
- factor (returned as character, with a warning)
- list of raw (if supported by the database)
- objects of type blob::blob (if supported by the database)

- date (if supported by the database; returned as `Date`) also for dates prior to 1970 or 1900 or after 2038
- time (if supported by the database; returned as objects that inherit from `difftime`)
- timestamp (if supported by the database; returned as `POSIXct` respecting the time zone but not necessarily preserving the input time zone), also for timestamps prior to 1970 or 1900 or after 2038 respecting the time zone but not necessarily preserving the input time zone)

Mixing column types in the same table is supported.

The `name` argument is processed as follows, to support databases that allow non-syntactic names for their objects:

- If an unquoted table name as string: `dbAppendTable()` will do the quoting, perhaps by calling `dbQuoteIdentifier(conn, x = name)`
- If the result of a call to `DBI::dbQuoteIdentifier()`: no more quoting is done to support databases that allow non-syntactic names for their objects:

The `row.names` argument must be `NULL`, the default value. Row names are ignored.

The `value` argument must be a data frame with a subset of the columns of the existing table. The order of the columns does not matter.

See Also

Other sql specifications: `spec_sql_create_table`, `spec_sql_exists_table`, `spec_sql_list_fields`, `spec_sql_list_objects`, `spec_sql_list_tables`, `spec_sql_quote_identifier`, `spec_sql_quote_literal`, `spec_sql_quote_string`, `spec_sql_read_table`, `spec_sql_remove_table`, `spec_sql_unquote_identifier`, `spec_sql_write_table`

`spec_sql_create_table` *spec_sql_create_table*

Description

spec_sql_create_table

Value

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

Failure modes

If the table exists, an error is raised; the remote table remains unchanged.

An error is raised when calling this method for a closed or invalid connection. An error is also raised if `name` cannot be processed with `DBI::dbQuoteIdentifier()` or if this results in a non-scalar. Invalid values for the `row.names` and `temporary` arguments (non-scalars, unsupported data types, `NA`, incompatible values, duplicate names) also raise an error.

Additional arguments

The following arguments are not part of the `dbCreateTable()` generic (to improve compatibility across backends) but are part of the DBI specification:

- `temporary` (default: `FALSE`)

They must be provided as named arguments. See the "Specification" and "Value" sections for details on their usage.

Specification

The `name` argument is processed as follows, to support databases that allow non-syntactic names for their objects:

- If an unquoted table name as string: `dbCreateTable()` will do the quoting, perhaps by calling `dbQuoteIdentifier(conn, x = name)`
- If the result of a call to `DBI::dbQuoteIdentifier()`: no more quoting is done

The `value` argument can be:

- a data frame,
- a named list of SQL types

If the `temporary` argument is `TRUE`, the table is not available in a second connection and is gone after reconnecting. Not all backends support this argument. A regular, non-temporary table is visible in a second connection, in a pre-existing connection, and after reconnecting to the database.

SQL keywords can be used freely in table names, column names, and data. Quotes, commas, and spaces can also be used for table names and column names, if the database supports non-syntactic identifiers.

The `row.names` argument must be missing or `NULL`, the default value. All other values for the `row.names` argument (in particular `TRUE`, `NA`, and a string) raise an error.

See Also

Other sql specifications: `spec_sql_append_table`, `spec_sql_exists_table`, `spec_sql_list_fields`, `spec_sql_list_objects`, `spec_sql_list_tables`, `spec_sql_quote_identifier`, `spec_sql_quote_literal`, `spec_sql_quote_string`, `spec_sql_read_table`, `spec_sql_remove_table`, `spec_sql_unquote_identifier`, `spec_sql_write_table`

`spec_sql_exists_table` *spec_sql_exists_table*

Description

spec_sql_exists_table

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.

Failure modes

An error is raised when calling this method for a closed or invalid connection. An error is also raised if `name` cannot be processed with `DBI::dbQuoteIdentifier()` or if this results in a non-scalar.

Specification

The `name` argument is processed as follows, to support databases that allow non-syntactic names for their objects:

- If an unquoted table name as string: `dbExistsTable()` will do the quoting, perhaps by calling `dbQuoteIdentifier(conn, x = name)`
- If the result of a call to `DBI::dbQuoteIdentifier()`: no more quoting is done

For all tables listed by `DBI::dbListTables()`, `dbExistsTable()` returns `TRUE`.

See Also

Other sql specifications: `spec_sql_append_table`, `spec_sql_create_table`, `spec_sql_list_fields`, `spec_sql_list_objects`, `spec_sql_list_tables`, `spec_sql_quote_identifier`, `spec_sql_quote_literal`, `spec_sql_quote_string`, `spec_sql_read_table`, `spec_sql_remove_table`, `spec_sql_unquote_identifier`, `spec_sql_write_table`

---

spec_sql_list_fields *spec_sql_list_fields*

Description

spec_sql_list_fields

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()`.

Failure modes

If the table does not exist, an error is raised. Invalid types for the `name` argument (e.g., `character` of length not equal to one, or numeric) lead to an error. An error is also raised when calling this method for a closed or invalid connection.

Specification

The name argument can be

- a string
- the return value of DBI::dbQuoteIdentifier()
- a value from the table column from the return value of DBI::dbListObjects() where is_prefix is FALSE

A column named row_names is treated like any other column.

See Also

Other sql specifications: spec_sql_append_table, spec_sql_create_table, spec_sql_exists_table, spec_sql_list_objects, spec_sql_list_tables, spec_sql_quote_identifier, spec_sql_quote_literal, spec_sql_quote_string, spec_sql_read_table, spec_sql_remove_table, spec_sql_unquote_identifier, spec_sql_write_table

---

spec_sql_list_objects *spec_sql_list_objects*

Description

spec_sql_list_objects

Value

dbListObjects() returns a data frame with columns table and is_prefix (in that order), optionally with other columns with a dot (.) prefix. The table column is of type list. Each object in this list is suitable for use as argument in DBI::dbQuoteIdentifier(). The is_prefix column is a logical. This data frame contains one row for each object (schema, table and view) accessible from the prefix (if passed) or from the global namespace (if prefix is omitted). Tables added with DBI::dbWriteTable() are part of the data frame. As soon a table is removed from the database, it is also removed from the data frame of database objects.

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

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

Failure modes

An error is raised when calling this method for a closed or invalid connection.

Specification

The `prefix` column indicates if the `table` value refers to a table or a prefix. For a call with the default `prefix = NULL`, the `table` values that have `is_prefix == FALSE` correspond to the tables returned from `DBI::dbListTables()`,

The `table` object can be quoted with `DBI::dbQuoteIdentifier()`. The result of quoting can be passed to `DBI::dbUnquoteIdentifier()`. (For backends it may be convenient to use the DBI::Id class, but this is not required.)

Values in `table` column that have `is_prefix == TRUE` can be passed as the `prefix` argument to another call to `dbListObjects()`. For the data frame returned from a `dbListObject()` call with the `prefix` argument set, all `table` values where `is_prefix` is `FALSE` can be used in a call to `DBI::dbExistsTable()` which returns `TRUE`.

See Also

Other sql specifications: `spec_sql_append_table`, `spec_sql_create_table`, `spec_sql_exists_table`, `spec_sql_list_fields`, `spec_sql_list_tables`, `spec_sql_quote_identifier`, `spec_sql_quote_literal`, `spec_sql_quote_string`, `spec_sql_read_table`, `spec_sql_remove_table`, `spec_sql_unquote_identifier`, `spec_sql_write_table`

`spec_sql_list_tables` *spec_sql_list_tables*

Description

spec_sql_list_tables

Value

`dbListTables()` returns a character vector that enumerates all tables and views in the database. Tables added with `DBI::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()`.

Failure modes

An error is raised when calling this method for a closed or invalid connection.

See Also

Other sql specifications: `spec_sql_append_table`, `spec_sql_create_table`, `spec_sql_exists_table`, `spec_sql_list_fields`, `spec_sql_list_objects`, `spec_sql_quote_identifier`, `spec_sql_quote_literal`, `spec_sql_quote_string`, `spec_sql_read_table`, `spec_sql_remove_table`, `spec_sql_unquote_identifier`, `spec_sql_write_table`

spec_sql_quote_identifier
spec_sql_quote_identifier

Description

spec_sql_quote_identifier

Value

`dbQuoteIdentifier()` returns an object that can be coerced to character, of the same length as the input. For an empty character vector this function returns a length-0 object. The names of the input argument are preserved in the output. When passing the returned object again to `dbQuoteIdentifier()` as `x` argument, it is returned unchanged. Passing objects of class DBI::SQL should also return them unchanged. (For backends it may be most convenient to return DBI::SQL objects to achieve this behavior, but this is not required.)

Failure modes

An error is raised if the input contains `NA`, but not for an empty string.

Specification

Calling `DBI::dbGetQuery()` for a query of the format `SELECT 1 AS ...` returns a data frame with the identifier, unquoted, as column name. Quoted identifiers can be used as table and column names in SQL queries, in particular in queries like `SELECT 1 AS ...` and `SELECT * FROM (SELECT 1) ...`. The method must use a quoting mechanism that is unambiguously different from the quoting mechanism used for strings, so that a query like `SELECT ... FROM (SELECT 1 AS ...)` throws an error if the column names do not match.

The method can quote column names that contain special characters such as a space, a dot, a comma, or quotes used to mark strings or identifiers, if the database supports this. In any case, checking the validity of the identifier should be performed only when executing a query, and not by `dbQuoteIdentifier()`.

See Also

Other sql specifications: `spec_sql_append_table`, `spec_sql_create_table`, `spec_sql_exists_table`, `spec_sql_list_fields`, `spec_sql_list_objects`, `spec_sql_list_tables`, `spec_sql_quote_literal`, `spec_sql_quote_string`, `spec_sql_read_table`, `spec_sql_remove_table`, `spec_sql_unquote_identifier`, `spec_sql_write_table`

spec_sql_quote_literal *spec_sql_quote_literal*

Description

spec_sql_quote_literal

Value

`dbQuoteLiteral()` returns an object that can be coerced to character, of the same length as the input. For an empty integer, numeric, character, logical, date, time, or blob vector, this function returns a length-0 object.

When passing the returned object again to `dbQuoteLiteral()` as `x` argument, it is returned unchanged. Passing objects of class DBI::SQL should also return them unchanged. (For backends it may be most convenient to return DBI::SQL objects to achieve this behavior, but this is not required.)

Specification

The returned expression can be used in a `SELECT ...` query, and the value of `dbGetQuery(paste0("SELECT ", dbQuoteLiteral(x)))[[1]]` must be equal to `x` for any scalar integer, numeric, string, and logical. If `x` is `NA`, the result must merely satisfy `is.na()`. The literals `"NA"` or `"NULL"` are not treated specially.

`NA` should be translated to an unquoted SQL `NULL`, so that the query `SELECT * FROM (SELECT 1) a WHERE ... IS NULL` returns one row.

Failure modes

Passing a list for the `x` argument raises an error.

See Also

Other sql specifications: `spec_sql_append_table`, `spec_sql_create_table`, `spec_sql_exists_table`, `spec_sql_list_fields`, `spec_sql_list_objects`, `spec_sql_list_tables`, `spec_sql_quote_identifier`, `spec_sql_quote_string`, `spec_sql_read_table`, `spec_sql_remove_table`, `spec_sql_unquote_identifier`, `spec_sql_write_table`

spec_sql_quote_string *spec_sql_quote_string*

Description

spec_sql_quote_string

Value

`dbQuoteString()` returns an object that can be coerced to character, of the same length as the input. For an empty character vector this function returns a length-0 object.

When passing the returned object again to `dbQuoteString()` as `x` argument, it is returned unchanged. Passing objects of class DBI::SQL should also return them unchanged. (For backends it may be most convenient to return DBI::SQL objects to achieve this behavior, but this is not required.)

Specification

The returned expression can be used in a `SELECT ...` query, and for any scalar character `x` the value of `dbGetQuery(paste0("SELECT ", dbQuoteString(x)))[[1]]` must be identical to `x`, even if `x` contains spaces, tabs, quotes (single or double), backticks, or newlines (in any combination) or is itself the result of a `dbQuoteString()` call coerced back to character (even repeatedly). If `x` is `NA`, the result must merely satisfy `is.na()`. The strings `"NA"` or `"NULL"` are not treated specially.

`NA` should be translated to an unquoted SQL `NULL`, so that the query `SELECT * FROM (SELECT 1) a WHERE ... IS NULL` returns one row.

Failure modes

Passing a numeric, integer, logical, or raw vector, or a list for the `x` argument raises an error.

See Also

Other sql specifications: `spec_sql_append_table`, `spec_sql_create_table`, `spec_sql_exists_table`, `spec_sql_list_fields`, `spec_sql_list_objects`, `spec_sql_list_tables`, `spec_sql_quote_identifier`, `spec_sql_quote_literal`, `spec_sql_read_table`, `spec_sql_remove_table`, `spec_sql_unquote_identifier`, `spec_sql_write_table`

spec_sql_read_table *spec_sql_read_table*

Description

spec_sql_read_table

Value

`dbReadTable()` returns a data frame that contains the complete data from the remote table, effectively the result of calling `DBI::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 `DBI::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.

Failure modes

An error is raised if the table does not exist.

An error is raised if `row.names` is `TRUE` and no "row_names" column exists,

An error is raised if `row.names` is set to a string and no corresponding column exists.

An error is raised when calling this method for a closed or invalid connection. An error is raised if `name` cannot be processed with `DBI::dbQuoteIdentifier()` or if this results in a non-scalar. Unsupported values for `row.names` and `check.names` (non-scalars, unsupported data types, `NA` for `check.names`) also raise an error.

Additional arguments

The following arguments are not part of the `dbReadTable()` generic (to improve compatibility across backends) but are part of the DBI specification:

- `row.names` (default: `FALSE`)
- `check.names`

They must be provided as named arguments. See the "Value" section for details on their usage.

Specification

The name argument is processed as follows, to support databases that allow non-syntactic names for their objects:

- If an unquoted table name as string: dbReadTable() will do the quoting, perhaps by calling dbQuoteIdentifier(conn, x = name)
- If the result of a call to DBI::dbQuoteIdentifier(): no more quoting is done

See Also

Other sql specifications: spec_sql_append_table, spec_sql_create_table, spec_sql_exists_table, spec_sql_list_fields, spec_sql_list_objects, spec_sql_list_tables, spec_sql_quote_identifier, spec_sql_quote_literal, spec_sql_quote_string, spec_sql_remove_table, spec_sql_unquote_identifier, spec_sql_write_table

---

spec_sql_remove_table *spec_sql_remove_table*

Description

spec_sql_remove_table

Value

dbRemoveTable() returns TRUE, invisibly.

Failure modes

If the table does not exist, an error is raised. An attempt to remove a view with this function may result in an error.

An error is raised when calling this method for a closed or invalid connection. An error is also raised if name cannot be processed with DBI::dbQuoteIdentifier() or if this results in a non-scalar.

Additional arguments

The following arguments are not part of the dbRemoveTable() generic (to improve compatibility across backends) but are part of the DBI specification:

- temporary (default: FALSE)
- fail_if_missing (default: TRUE)

These arguments must be provided as named arguments.

If temporary is TRUE, the call to dbRemoveTable() will consider only temporary tables. Not all backends support this argument. In particular, permanent tables of the same name are left untouched.

If fail_if_missing is FALSE, the call to dbRemoveTable() succeeds if the table does not exist.

Specification

A table removed by `dbRemoveTable()` doesn't appear in the list of tables returned by `DBI::dbListTables()`, and `DBI::dbExistsTable()` returns `FALSE`. The removal propagates immediately to other connections to the same database. This function can also be used to remove a temporary table.

The `name` argument is processed as follows, to support databases that allow non-syntactic names for their objects:

- If an unquoted table name as string: `dbRemoveTable()` will do the quoting, perhaps by calling `dbQuoteIdentifier(conn, x = name)`
- If the result of a call to `DBI::dbQuoteIdentifier()`: no more quoting is done

See Also

Other sql specifications: `spec_sql_append_table`, `spec_sql_create_table`, `spec_sql_exists_table`, `spec_sql_list_fields`, `spec_sql_list_objects`, `spec_sql_list_tables`, `spec_sql_quote_identifier`, `spec_sql_quote_literal`, `spec_sql_quote_string`, `spec_sql_read_table`, `spec_sql_unquote_identifier`, `spec_sql_write_table`

`spec_sql_unquote_identifier` *spec_sql_unquote_identifier*

Description

spec_sql_unquote_identifier

Value

`dbUnquoteIdentifier()` returns a list of objects of the same length as the input. For an empty vector, this function returns a length-0 object. The names of the input argument are preserved in the output. If `x` is a value returned by `dbUnquoteIdentifier()`, calling `dbUnquoteIdentifier(..., dbQuoteIdentifier(..., x))` returns `list(x)`. If `x` is an object of class DBI::Id, calling `dbUnquoteIdentifier(..., x)` returns `list(x)`. (For backends it may be most convenient to return DBI::Id objects to achieve this behavior, but this is not required.)

Plain character vectors can also be passed to `dbUnquoteIdentifier()`.

Failure modes

An error is raised if a character vectors with a missing value is passed as the `x` argument.

Specification

For any character vector of length one, quoting (with DBI::dbQuoteIdentifier()) then unquoting then quoting the first element is identical to just quoting. This is also true for strings that contain special characters such as a space, a dot, a comma, or quotes used to mark strings or identifiers, if the database supports this.

Unquoting simple strings (consisting of only letters) wrapped with DBI::SQL() and then quoting via DBI::dbQuoteIdentifier() gives the same result as just quoting the string. Similarly, unquoting expressions of the form SQL("schema.table") and then quoting gives the same result as quoting the identifier constructed by Id("schema", "table").

See Also

Other sql specifications: spec_sql_append_table, spec_sql_create_table, spec_sql_exists_table, spec_sql_list_fields, spec_sql_list_objects, spec_sql_list_tables, spec_sql_quote_identifier, spec_sql_quote_literal, spec_sql_quote_string, spec_sql_read_table, spec_sql_remove_table, spec_sql_write_table

---

spec_sql_write_table *spec_sql_write_table*

Description

spec_sql_write_table

Value

dbWriteTable() returns TRUE, invisibly.

Failure modes

If the table exists, and both append and overwrite arguments are unset, or append = TRUE and the data frame with the new data has different column names, an error is raised; the remote table remains unchanged.

An error is raised when calling this method for a closed or invalid connection. An error is also raised if name cannot be processed with DBI::dbQuoteIdentifier() or if this results in a non-scalar. Invalid values for the additional arguments row.names, overwrite, append, field.types, and temporary (non-scalars, unsupported data types, NA, incompatible values, duplicate or missing names, incompatible columns) also raise an error.

Additional arguments

The following arguments are not part of the dbWriteTable() generic (to improve compatibility across backends) but are part of the DBI specification:

- row.names (default: FALSE)
- overwrite (default: FALSE)

- append (default: FALSE)
- field.types (default: NULL)
- temporary (default: FALSE)

They must be provided as named arguments. See the "Specification" and "Value" sections for details on their usage.

Specification

The name argument is processed as follows, to support databases that allow non-syntactic names for their objects:

- If an unquoted table name as string: dbWriteTable() will do the quoting, perhaps by calling dbQuoteIdentifier(conn, x = name)
- If the result of a call to DBI::dbQuoteIdentifier(): no more quoting is done

The value argument must be a data frame with a subset of the columns of the existing table if append = TRUE. The order of the columns does not matter with append = TRUE.

If the overwrite argument is TRUE, an existing table of the same name will be overwritten. This argument doesn't change behavior if the table does not exist yet.

If the append argument is TRUE, the rows in an existing table are preserved, and the new data are appended. If the table doesn't exist yet, it is created.

If the temporary argument is TRUE, the table is not available in a second connection and is gone after reconnecting. Not all backends support this argument. A regular, non-temporary table is visible in a second connection, in a pre-existing connection, and after reconnecting to the database.

SQL keywords can be used freely in table names, column names, and data. Quotes, commas, spaces, and other special characters such as newlines and tabs, can also be used in the data, and, if the database supports non-syntactic identifiers, also for table names and column names.

The following data types must be supported at least, and be read identically with DBI::dbReadTable():

- integer
- numeric (the behavior for Inf and NaN is not specified)
- logical
- NA as NULL
- 64-bit values (using "bigint" as field type); the result can be
  - converted to a numeric, which may lose precision,
  - converted a character vector, which gives the full decimal representation
  - written to another table and read again unchanged
- character (in both UTF-8 and native encodings), supporting empty strings before and after a non-empty string
- factor (returned as character)
- list of raw (if supported by the database)
- objects of type blob::blob (if supported by the database)
- date (if supported by the database; returned as Date), also for dates prior to 1970 or 1900 or after 2038

- time (if supported by the database; returned as objects that inherit from `difftime`)
- timestamp (if supported by the database; returned as `POSIXct` respecting the time zone but not necessarily preserving the input time zone), also for timestamps prior to 1970 or 1900 or after 2038 respecting the time zone but not necessarily preserving the input time zone)

Mixing column types in the same table is supported.

The `field.types` argument must be a named character vector with at most one entry for each column. It indicates the SQL data type to be used for a new column. If a column is missed from `field.types`, the type is inferred from the input data with `DBI::dbDataType()`.

The interpretation of rownames depends on the `row.names` argument, see `DBI::sqlRownamesToColumn()` for details:

- If `FALSE` or `NULL`, row names are ignored.
- If `TRUE`, row names are converted to a column named "row_names", even if the input data frame only has natural row names from 1 to `nrow(...)`.
- If `NA`, a column named "row_names" is created if the data has custom row names, no extra column is created in the case of natural row names.
- If a string, this specifies the name of the column in the remote table that contains the row names, even if the input data frame only has natural row names.

The default is `row.names = FALSE`.

See Also

Other sql specifications: `spec_sql_append_table`, `spec_sql_create_table`, `spec_sql_exists_table`, `spec_sql_list_fields`, `spec_sql_list_objects`, `spec_sql_list_tables`, `spec_sql_quote_identifier`, `spec_sql_quote_literal`, `spec_sql_quote_string`, `spec_sql_read_table`, `spec_sql_remove_table`, `spec_sql_unquote_identifier`

`spec_transaction_begin_commit_rollback`
spec_transaction_begin_commit_rollback

Description

spec_transaction_begin_commit_rollback

Value

`dbBegin()`, `dbCommit()` and `dbRollback()` return `TRUE`, invisibly.

Failure modes

The implementations are expected to raise an error in case of failure, but this is not tested. In any way, all generics throw an error with a closed or invalid connection. In addition, a call to `dbCommit()` or `dbRollback()` without a prior call to `dbBegin()` raises an error. Nested transactions are not supported by DBI, an attempt to call `dbBegin()` twice yields an error.

Specification

Actual support for transactions may vary between backends. A transaction is initiated by a call to `dbBegin()` and committed by a call to `dbCommit()`. Data written in a transaction must persist after the transaction is committed. For example, a record that is missing when the transaction is started but is created during the transaction must exist both during and after the transaction, and also in a new connection.

A transaction can also be aborted with `dbRollback()`. All data written in such a transaction must be removed after the transaction is rolled back. For example, a record that is missing when the transaction is started but is created during the transaction must not exist anymore after the rollback.

Disconnection from a connection with an open transaction effectively rolls back the transaction. All data written in such a transaction must be removed after the transaction is rolled back.

The behavior is not specified if other arguments are passed to these functions. In particular, **RSQLite** issues named transactions with support for nesting if the `name` argument is set.

The transaction isolation level is not specified by DBI.

See Also

Other transaction specifications: `spec_transaction_with_transaction`

`spec_transaction_with_transaction`
spec_transaction_with_transaction

Description

spec_transaction_with_transaction

Value

`dbWithTransaction()` returns the value of the executed code.

Failure modes

Failure to initiate the transaction (e.g., if the connection is closed or invalid of if `DBI::dbBegin()` has been called already) gives an error.

Specification

`dbWithTransaction()` initiates a transaction with `dbBegin()`, executes the code given in the `code` argument, and commits the transaction with `DBI::dbCommit()`. If the code raises an error, the transaction is instead aborted with `DBI::dbRollback()`, and the error is propagated. If the code calls `dbBreak()`, execution of the code stops and the transaction is silently aborted. All side effects caused by the code (such as the creation of new variables) propagate to the calling environment.

See Also

Other transaction specifications: `spec_transaction_begin_commit_rollback`

test_all *Run all tests*

Description

`test_all()` calls all tests defined in this package (see the section "Tests" below). This function supports running only one test by setting an environment variable, e.g., set the `DBITEST_ONLY_RESULT` to a nonempty value to run only `test_result()`.

`test_some()` allows testing one or more tests.

Usage

```
test_all(skip = NULL, run_only = NULL, ctx = get_default_context())

test_some(test, ctx = get_default_context())
```

Arguments

`skip`	`[character()]` A vector of regular expressions to match against test names; skip test if matching any. The regular expressions are matched against the entire test name minus a possible suffix `_N` where `N` is a number. For example, `skip = "exists_table"` will skip both `"exists_table_1"` and `"exists_table_2"`.
`run_only`	`[character()]` A vector of regular expressions to match against test names; run only these tests. The regular expressions are matched against the entire test name.
`ctx`	`[DBItest_context]` A test context as created by `make_context()`.
`test`	`[character]` A character vector of regular expressions describing the tests to run. The regular expressions are matched against the entire test name.

Details

Internally `^` and `$` are used as prefix and suffix around the regular expressions passed in the `skip` and `run_only` arguments.

Tests

This function runs the following tests, except the stress tests:

`test_getting_started()`: Getting started with testing

`test_driver()`: Test the "Driver" class

`test_connection()`: Test the "Connection" class

`test_result()`: Test the "Result" class

`test_sql()`: Test SQL methods

test_meta(): Test metadata functions

test_transaction(): Test transaction functions

test_arrow(): Test Arrow methods

test_compliance(): Test full compliance to DBI

test_stress(): Stress tests (not tested with test_all)

---

test_arrow *Test Arrow methods*

Description

Test Arrow methods

Usage

```
test_arrow(skip = NULL, run_only = NULL, ctx = get_default_context())
```

Arguments

skip	[character()] A vector of regular expressions to match against test names; skip test if matching any. The regular expressions are matched against the entire test name minus a possible suffix _N where N is a number. For example, skip = "exists_table" will skip both "exists_table_1" and "exists_table_2".
run_only	[character()] A vector of regular expressions to match against test names; run only these tests. The regular expressions are matched against the entire test name.
ctx	[DBItest_context] A test context as created by make_context().

See Also

Other tests: test_compliance(), test_connection(), test_driver(), test_getting_started(), test_meta(), test_result(), test_sql(), test_stress(), test_transaction()

test_compliance *Test full compliance to DBI*

Description

Test full compliance to DBI

Usage

```
test_compliance(skip = NULL, run_only = NULL, ctx = get_default_context())
```

Arguments

skip	[character()] A vector of regular expressions to match against test names; skip test if matching any. The regular expressions are matched against the entire test name minus a possible suffix _N where N is a number. For example, skip = "exists_table" will skip both "exists_table_1" and "exists_table_2".
run_only	[character()] A vector of regular expressions to match against test names; run only these tests. The regular expressions are matched against the entire test name.
ctx	[DBItest_context] A test context as created by make_context().

See Also

Other tests: test_arrow(), test_connection(), test_driver(), test_getting_started(), test_meta(), test_result(), test_sql(), test_stress(), test_transaction()

test_connection *Test the "Connection" class*

Description

Test the "Connection" class

Usage

```
test_connection(skip = NULL, run_only = NULL, ctx = get_default_context())
```

Arguments

skip	[character()] A vector of regular expressions to match against test names; skip test if matching any. The regular expressions are matched against the entire test name minus a possible suffix _N where N is a number. For example, skip = "exists_table" will skip both "exists_table_1" and "exists_table_2".
run_only	[character()] A vector of regular expressions to match against test names; run only these tests. The regular expressions are matched against the entire test name.
ctx	[DBItest_context] A test context as created by make_context().

See Also

Other tests: test_arrow(), test_compliance(), test_driver(), test_getting_started(), test_meta(), test_result(), test_sql(), test_stress(), test_transaction()

---

test_driver *Test the "Driver" class*

Description

Test the "Driver" class

Usage

```
test_driver(skip = NULL, run_only = NULL, ctx = get_default_context())
```

Arguments

skip	[character()] A vector of regular expressions to match against test names; skip test if matching any. The regular expressions are matched against the entire test name minus a possible suffix _N where N is a number. For example, skip = "exists_table" will skip both "exists_table_1" and "exists_table_2".
run_only	[character()] A vector of regular expressions to match against test names; run only these tests. The regular expressions are matched against the entire test name.
ctx	[DBItest_context] A test context as created by make_context().

See Also

Other tests: test_arrow(), test_compliance(), test_connection(), test_getting_started(), test_meta(), test_result(), test_sql(), test_stress(), test_transaction()

test_getting_started *Getting started with testing*

Description

Tests very basic features of a DBI driver package, to support testing and test-first development right from the start.

Usage

```
test_getting_started(skip = NULL, run_only = NULL, ctx = get_default_context())
```

Arguments

skip	[character()] A vector of regular expressions to match against test names; skip test if matching any. The regular expressions are matched against the entire test name minus a possible suffix _N where N is a number. For example, skip = "exists_table" will skip both "exists_table_1" and "exists_table_2".
run_only	[character()] A vector of regular expressions to match against test names; run only these tests. The regular expressions are matched against the entire test name.
ctx	[DBItest_context] A test context as created by make_context().

See Also

Other tests: test_arrow(), test_compliance(), test_connection(), test_driver(), test_meta(), test_result(), test_sql(), test_stress(), test_transaction()

test_meta *Test metadata functions*

Description

Test metadata functions

Usage

```
test_meta(skip = NULL, run_only = NULL, ctx = get_default_context())
```

Arguments

skip	[character()] A vector of regular expressions to match against test names; skip test if matching any. The regular expressions are matched against the entire test name minus a possible suffix _N where N is a number. For example, skip = "exists_table" will skip both "exists_table_1" and "exists_table_2".
run_only	[character()] A vector of regular expressions to match against test names; run only these tests. The regular expressions are matched against the entire test name.
ctx	[DBItest_context] A test context as created by make_context().

See Also

Other tests: test_arrow(), test_compliance(), test_connection(), test_driver(), test_getting_started(), test_result(), test_sql(), test_stress(), test_transaction()

---

test_result *Test the "Result" class*

Description

Test the "Result" class

Usage

```
test_result(skip = NULL, run_only = NULL, ctx = get_default_context())
```

Arguments

skip	[character()] A vector of regular expressions to match against test names; skip test if matching any. The regular expressions are matched against the entire test name minus a possible suffix _N where N is a number. For example, skip = "exists_table" will skip both "exists_table_1" and "exists_table_2".
run_only	[character()] A vector of regular expressions to match against test names; run only these tests. The regular expressions are matched against the entire test name.
ctx	[DBItest_context] A test context as created by make_context().

See Also

Other tests: test_arrow(), test_compliance(), test_connection(), test_driver(), test_getting_started(), test_meta(), test_sql(), test_stress(), test_transaction()

test_sql *Test SQL methods*

Description

Test SQL methods

Usage

```
test_sql(skip = NULL, run_only = NULL, ctx = get_default_context())
```

Arguments

skip	[character()] A vector of regular expressions to match against test names; skip test if matching any. The regular expressions are matched against the entire test name minus a possible suffix _N where N is a number. For example, skip = "exists_table" will skip both "exists_table_1" and "exists_table_2".
run_only	[character()] A vector of regular expressions to match against test names; run only these tests. The regular expressions are matched against the entire test name.
ctx	[DBItest_context] A test context as created by make_context().

See Also

Other tests: test_arrow(), test_compliance(), test_connection(), test_driver(), test_getting_started(), test_meta(), test_result(), test_stress(), test_transaction()

test_transaction *Test transaction functions*

Description

Test transaction functions

Usage

```
test_transaction(skip = NULL, run_only = NULL, ctx = get_default_context())
```

Arguments

skip	[character()] A vector of regular expressions to match against test names; skip test if matching any. The regular expressions are matched against the entire test name minus a possible suffix _N where N is a number. For example, skip = "exists_table" will skip both "exists_table_1" and "exists_table_2".
run_only	[character()] A vector of regular expressions to match against test names; run only these tests. The regular expressions are matched against the entire test name.
ctx	[DBItest_context] A test context as created by make_context().

See Also

Other tests: test_arrow(), test_compliance(), test_connection(), test_driver(), test_getting_started(), test_meta(), test_result(), test_sql(), test_stress()

---

tweaks *Tweaks for DBI tests*

Description

The tweaks are a way to control the behavior of certain tests. Currently, you need to search the **DBItest** source code to understand which tests are affected by which tweaks. This function is usually called to set the tweaks argument in a make_context() call.

Usage

```
tweaks(
  ...,
  constructor_name = NULL,
  constructor_relax_args = FALSE,
  strict_identifier = FALSE,
  omit_blob_tests = FALSE,
  current_needs_parens = FALSE,
  union = function(x) paste(x, collapse = " UNION "),
  placeholder_pattern = NULL,
  logical_return = identity,
  date_cast = function(x) paste0("date('", x, "')"),
  time_cast = function(x) paste0("time('", x, "')"),
  timestamp_cast = function(x) paste0("timestamp('", x, "')"),
  blob_cast = identity,
  date_typed = TRUE,
  time_typed = TRUE,
  timestamp_typed = TRUE,
  temporary_tables = TRUE,
```

```
  list_temporary_tables = TRUE,
  allow_na_rows_affected = FALSE,
  is_null_check = function(x) paste0("(", x, " IS NULL)"),
 create_table_as = function(table_name, query) paste0("CREATE TABLE ", table_name,
    " AS ", query),
  dbitest_version = "1.7.1"
)
```

Arguments

`...`	`[any]` Unknown tweaks are accepted, with a warning. The ellipsis also makes sure that you only can pass named arguments.
`constructor_name`	`[character(1)]` Name of the function that constructs the `Driver` object.
`constructor_relax_args`	`[logical(1)]` If `TRUE`, allow a driver constructor with default values for all arguments; otherwise, require a constructor with empty argument list (default).
`strict_identifier`	`[logical(1)]` Set to `TRUE` if the DBMS does not support arbitrarily-named identifiers even when quoting is used.
`omit_blob_tests`	`[logical(1)]` Set to `TRUE` if the DBMS does not support a `BLOB` data type.
`current_needs_parens`	`[logical(1)]` Set to `TRUE` if the SQL functions `current_date`, `current_time`, and `current_timestamp` require parentheses.
`union`	`[function(character)]` Function that combines several subqueries into one so that the resulting query returns the concatenated results of the subqueries
`placeholder_pattern`	`[character]` A pattern for placeholders used in `DBI::dbBind()`, e.g., `"?"`, `"$1"`, or `":name"`. See `make_placeholder_fun()` for details.
`logical_return`	`[function(logical)]` A vectorized function that converts logical values to the data type returned by the DBI backend.
`date_cast`	`[function(character)]` A vectorized function that creates an SQL expression for coercing a string to a date value.
`time_cast`	`[function(character)]` A vectorized function that creates an SQL expression for coercing a string to a time value.

`timestamp_cast` `[function(character)]`
A vectorized function that creates an SQL expression for coercing a string to a timestamp value.

`blob_cast` `[function(character)]`
A vectorized function that creates an SQL expression for coercing a string to a blob value.

`date_typed` `[logical(1L)]`
Set to `FALSE` if the DBMS doesn't support a dedicated type for dates.

`time_typed` `[logical(1L)]`
Set to `FALSE` if the DBMS doesn't support a dedicated type for times.

`timestamp_typed`
`[logical(1L)]`
Set to `FALSE` if the DBMS doesn't support a dedicated type for timestamps.

`temporary_tables`
`[logical(1L)]`
Set to `FALSE` if the DBMS doesn't support temporary tables.

`list_temporary_tables`
`[logical(1L)]`
Set to `FALSE` if the DBMS doesn't support listing temporary tables.

`allow_na_rows_affected`
`[logical(1L)]`
Set to `TRUE` to allow `DBI::dbGetRowsAffected()` to return `NA`.

`is_null_check` `[function(character)]`
A vectorized function that creates an SQL expression for checking if a value is `NULL`.

`create_table_as`
`[function(character(1), character(1))]`
A function that creates an SQL expression for creating a table from an SQL expression.

`dbitest_version`
`[character(1)]`
Compatible DBItest version, default: "1.7.1".

Examples

```
## Not run:
make_context(..., tweaks = tweaks(strict_identifier = TRUE))

## End(Not run)
```

Index