

# Package ‘DBItest’

January 25, 2018

**Title** Testing 'DBI' Back Ends

**Version** 1.5-2

**Date** 2018-01-26

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

**Depends** R (>= 3.0.0)

**Imports** blob, DBI (>= 0.4-9), desc, hms, methods, R6, testthat (>= 1.0.2), withr

**Suggests** knitr, lintr, rmarkdown

**License** LGPL (>= 2)

**LazyData** true

**Encoding** UTF-8

**BugReports** <https://github.com/rstats-db/DBItest/issues>

**RoxygenNote** 6.0.1

**VignetteBuilder** knitr

**Collate** 'DBItest.R' 'context.R' 'expectations.R' 'import-dbi.R' 'import-testthat.R' 'run.R' 's4.R' 'spec.R' 'spec-getting-started.R' 'spec-compliance-methods.R' 'spec-driver-constructor.R' 'spec-driver-class.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-identifier.R' 'spec-sql-read-table.R' 'spec-sql-write-table.R' 'spec-sql-list-tables.R' 'spec-sql-exists-table.R' 'spec-sql-remove-table.R' 'spec-meta-bind-runner.R' 'spec-meta-bind-tester-extra.R' 'spec-meta-bind.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-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-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-compliance.R' 'test-stress.R' 'tweaks.R' 'utf8.R'  
 'utils.R'

**NeedsCompilation** no

**Author** Kirill Müller [aut, cre],  
 RStudio [cph],  
 R Consortium [fnd]

**Maintainer** Kirill Müller <kr1mlr+r@mailbox.org>

**Repository** CRAN

**Date/Publication** 2018-01-25 19:54:58 UTC

## R topics documented:

|   |    |
|---|----|
| DBItest-package . . . . .                         | 3  |
| DBIspec . . . . .                                 | 4  |
| make_context . . . . .                            | 4  |
| spec_connection_disconnect . . . . .              | 5  |
| spec_driver_connect . . . . .                     | 6  |
| spec_driver_data_type . . . . .                   | 6  |
| spec_meta_bind . . . . .                          | 7  |
| spec_meta_get_rows_affected . . . . .             | 8  |
| spec_meta_get_row_count . . . . .                 | 9  |
| spec_meta_get_statement . . . . .                 | 9  |
| spec_meta_has_completed . . . . .                 | 10 |
| spec_meta_is_valid . . . . .                      | 10 |
| spec_result_clear_result . . . . .                | 11 |
| spec_result_create_table_with_data_type . . . . . | 11 |
| spec_result_execute . . . . .                     | 11 |
| spec_result_fetch . . . . .                       | 12 |
| spec_result_get_query . . . . .                   | 12 |
| spec_result_roundtrip . . . . .                   | 13 |
| spec_result_send_query . . . . .                  | 14 |
| spec_result_send_statement . . . . .              | 15 |
| spec_sql_exists_table . . . . .                   | 15 |
| spec_sql_list_tables . . . . .                    | 16 |
| spec_sql_quote_identifier . . . . .               | 16 |

|  |           |
|--|-----------|
| spec_sql_quote_string . . . . .                  | 17        |
| spec_sql_read_table . . . . .                    | 18        |
| spec_sql_remove_table . . . . .                  | 19        |
| spec_sql_write_table . . . . .                   | 20        |
| spec_transaction_begin_commit_rollback . . . . . | 22        |
| spec_transaction_with_transaction . . . . .      | 22        |
| test_all . . . . .                               | 23        |
| test_compliance . . . . .                        | 24        |
| test_connection . . . . .                        | 24        |
| test_data_type . . . . .                         | 25        |
| test_driver . . . . .                            | 25        |
| test_getting_started . . . . .                   | 26        |
| test_meta . . . . .                              | 27        |
| test_result . . . . .                            | 27        |
| test_sql . . . . .                               | 28        |
| test_stress . . . . .                            | 28        |
| test_transaction . . . . .                       | 29        |
| tweaks . . . . .                                 | 29        |
| <b>Index</b>                                     | <b>32</b> |

---

|                 |   |
|-----------------|---|
| DBItest-package | <i>DBItest: Testing 'DBI' Back Ends</i> |
|-----------------|---|

---

## 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:

- Report bugs at <https://github.com/rstats-db/DBItest/issues>

---

 DBIspec

*DBI specification*


---

### Description

Placeholder page.

### 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.

### DBI classes and methods

A backend defines three classes, which are subclasses of **DBIDriver**, **DBIConnection**, and **DBIResult**. The backend provides implementation for all methods of these base classes that are defined but not implemented by DBI. All methods have an ellipsis . . . in their formal.

### Construction of the DBIDriver object

The backend must support creation of an instance of its **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.

---

 make\_context

*Test contexts*


---

### Description

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

### Usage

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

```
set_default_context(ctx)
```

```
get_default_context()
```

**Arguments**

|                |   |
|----------------|---|
| drv            | [DBIDriver]<br>An expression that constructs a DBI driver, like SQLite().           |
| connect_args   | [named list]<br>Connection arguments (names and values).                            |
| set_as_default | [logical(1)]<br>Should the created context be set as default context?               |
| tweaks         | [DBItest_tweaks]<br>Tweaks as constructed by the <code>tweaks()</code> function.    |
| name           | [character]<br>An optional name of the context which will be used in test messages. |
| ctx            | [DBItest_context]<br>A test context.  |

**Value**

[DBItest\_context]  
A test context, for `set_default_context` the previous default context (invisibly) or NULL.

---

spec\_connection\_disconnect  
*spec\_connection\_disconnect*

---

**Description**

spec\_connection\_disconnect

**Value**

dbDisconnect() returns TRUE, invisibly.

**Specification**

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

spec\_driver\_connect    *spec\_driver\_connect*

---

### Description

spec\_driver\_connect

### Value

dbConnect() returns an S4 object that inherits from [DBIConnection](#). This object is used to communicate with the database engine.

### 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.

---

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

## Specification

The backend can override the `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.

---

|                |                       |
|----------------|-----------------------|
| spec_meta_bind | <i>spec_meta_bind</i> |
|----------------|-----------------------|

---

## Description

spec\_meta\_bind

spec\_meta\_bind

## Value

`dbBind()` returns the result set, invisibly, for queries issued by `dbSendQuery()` and also for data manipulation statements issued by `dbSendStatement()`. 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 `dbClearResult()` also raises an error.

## Specification

DBI clients execute parametrized statements as follows:

1. Call `dbSendQuery()` or `dbSendStatement()` with a query or statement that contains placeholders, store the returned `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 `dbClearResult()` via `on.exit()` right after calling `dbSendQuery()` or `dbSendStatement()` (see the last enumeration item). Until `dbBind()` has been called, the returned result set object has the following behavior:
  - `dbFetch()` raises an error (for `dbSendQuery()`)
  - `dbGetRowCount()` returns zero (for `dbSendQuery()`)
  - `dbGetRowsAffected()` returns an integer NA (for `dbSendStatement()`)
  - `dbIsValid()` returns TRUE
  - `dbHasCompleted()` returns FALSE

2. Construct a list with parameters that specify actual values for the placeholders. The list must be named or unnamed, depending on the kind of placeholders used. Named values are matched to named parameters, unnamed values are matched by position in the list of parameters. All elements in this list must have the same lengths and contain values supported by the backend; a `data.frame` is internally stored as such a list. The parameter list is passed to a call to `dbBind()` on the `DBIResult` object.
3. Retrieve the data or the number of affected rows from the `DBIResult` object.
  - For queries issued by `dbSendQuery()`, call `dbFetch()`.
  - For statements issued by `dbSendStatements()`, call `dbGetRowsAffected()`. (Execution begins immediately after the `dbBind()` call, the statement is processed entirely before the function returns.)
4. Repeat 2. and 3. as necessary.
5. Close the result set via `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()`.

At least the following data types are accepted:

- `integer`
- `numeric`
- `logical` for Boolean values (some backends may return an integer)
- `NA`
- `character`
- `factor` (bound as character, with warning)
- `Date`
- `POSIXct` timestamps
- `POSIXlt` timestamps
- lists of `raw` for blobs (with `NULL` entries for SQL `NULL` values)
- objects of type `blob::blob`

---

`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 `dbSendStatement()`. The value is available directly after the call and does not change after calling `dbFetch()`. For queries issued with `dbSendQuery()`, zero is returned before and after the call to `dbFetch()`. Attempting to get the rows affected for a result set cleared with `dbClearResult()` gives an error.

---

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 `dbSendQuery()`, the row count is initially zero. After a call to `dbFetch()` without limit, the row count matches the total number of rows returned. Fetching a limited number of rows increases the number of rows by the number of rows returned, even if fetching past the end of the result set. For queries with an empty result set, zero is returned even after fetching. For data manipulation statements issued with `dbSendStatement()`, zero is returned before and after calling `dbFetch()`. Attempting to get the row count for a result set cleared with `dbClearResult()` gives an error.

---

spec\_meta\_get\_statement

*spec\_meta\_get\_statement*

---

**Description**

spec\_meta\_get\_statement

**Value**

dbGetStatement() returns a string, the query used in either `dbSendQuery()` or `dbSendStatement()`. Attempting to query the statement for a result set cleared with `dbClearResult()` gives an error.

---

 spec\_meta\_has\_completed

*spec\_meta\_has\_completed*


---

### Description

spec\_meta\_has\_completed

### Value

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

---

 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 `DBIConnection` object is initially valid, and becomes invalid after disconnecting with `dbDisconnect()`. A `DBIResult` object is valid after a call to `dbSendQuery()`, and stays valid even after all rows have been fetched; only clearing it with `dbClearResult()` invalidates it. A `DBIResult` object is also valid after a call to `dbSendStatement()`, and stays valid after querying the number of rows affected; only clearing it with `dbClearResult()` invalidates it. If the connection to the database system is dropped (e.g., due to connectivity problems, server failure, etc.), `dbIsValid()` should return FALSE. This is not tested automatically.

---

spec\_result\_clear\_result  
*spec\_result\_clear\_result*

---

**Description**

spec\_result\_clear\_result

**Value**

dbClearResult() returns TRUE, invisibly, for result sets obtained from both dbSendQuery() and dbSendStatement(). An attempt to close an already closed result set issues a warning in both cases.

**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 dbSendQuery() or dbSendStatement() call.

---

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 ...)".

---

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. 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 argument is not part of the `dbExecute()` generic (to improve compatibility across backends) but is part of the DBI specification:

- `params` (TBD)

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

---

|                                |                          |
|--------------------------------|--------------------------|
| <code>spec_result_fetch</code> | <i>spec_result_fetch</i> |
|--------------------------------|--------------------------|

---

**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. 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 `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.

---

|                                    |                              |
|------------------------------------|------------------------------|
| <code>spec_result_get_query</code> | <i>spec_result_get_query</i> |
|------------------------------------|------------------------------|

---

**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. 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 (TBD)

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

**Specification**

Fetching multi-row queries with one or more columns by default 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, the result is returned in full without warning. 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.

A column named row\_names is treated like any other column.

---

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](#) for integer values between  $-2^{31}$  and  $2^{31} - 1$
- [numeric](#) for numbers with a fractional component
- [logical](#) for Boolean values (some backends may return an integer)
- [character](#) for text
- lists of [raw](#) for blobs (with NULL entries for SQL NULL values)
- coercible using [as.Date\(\)](#) for dates (also applies to the return value of the SQL function current\_date)

- coercible using `hms::as.hms()` for times (also applies to the return value of the SQL function `current_time`)
- coercible using `as.POSIXct()` for timestamps (also applies to the return value of the SQL function `current_timestamp`)
- `NA` for SQL NULL values

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

- `Date` for dates (also applies to the return value of the SQL function `current_date`)
- `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

---

spec\_result\_send\_query

*spec\_result\_send\_query*

---

## Description

spec\_result\_send\_query

## Value

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

## Specification

No warnings occur under normal conditions. When done, the `DBIResult` object must be cleared with a call to `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()`.

---

spec\_result\_send\_statement  
*spec\_result\_send\_statement*

---

**Description**

spec\_result\_send\_statement

**Value**

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

**Specification**

No warnings occur under normal conditions. When done, the DBIResult object must be cleared with a call to [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\(\)](#).

---

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.

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 [dbQuoteIdentifier\(\)](#) or if this results in a non-scalar.

**Additional arguments**

TBD: temporary = NA

This must be provided as named argument. See the "Specification" 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: `dbExistsTable()` will do the quoting, perhaps by calling `dbQuoteIdentifier(conn, x = name)`
- If the result of a call to `dbQuoteIdentifier()`: no more quoting is done

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

---

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 `dbWriteTable()` are part of the list, including temporary tables if supported by the database. As soon a table is removed from the database, it is also removed from the list of database tables.

The returned names are suitable for quoting with `dbQuoteIdentifier()`. An error is raised when calling this method for a closed or invalid connection.

**Additional arguments**

TBD: temporary = NA

This must be provided as named argument. See the "Specification" section for details on their usage.

---

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

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



**Specification**

Calling `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()`.

---

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 [SQL](#) should also return them unchanged. (For backends it may be most convenient to return [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.

---

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 `dbGetQuery()` with `SELECT * FROM <name>`. An error is raised if the table does not exist. An empty table is returned as a data frame with zero rows.

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

- If FALSE or NULL, the returned data frame doesn't have row names.
- If TRUE, a column named "row\_names" is converted to row names, an error is raised if no such column exists.
- 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, an error is raised if no such column exists.

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, otherwise non-syntactic column names can be returned unquoted.

An error is raised when calling this method for a closed or invalid connection. An error is raised if name cannot be processed with `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`
- `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 `dbQuoteIdentifier()`: no more quoting is done

---

spec\_sql\_remove\_table *spec\_sql\_remove\_table*

---

**Description**

spec\_sql\_remove\_table

**Value**

`dbRemoveTable()` returns TRUE, invisibly. 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 `dbQuoteIdentifier()` or if this results in a non-scalar.

**Specification**

A table removed by `dbRemoveTable()` doesn't appear in the list of tables returned by `dbListTables()`, and `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 `dbQuoteIdentifier()`: no more quoting is done

---

spec\_sql\_write\_table *spec\_sql\_write\_table*

---

### Description

spec\_sql\_write\_table

### Value

dbWriteTable() returns TRUE, invisibly. 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 `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: NA)
- 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 dbQuoteIdentifier(): no more quoting is done

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 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 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 `dbReadTable()`:

- integer
- numeric (also with Inf and NaN values, the latter are translated to NA)
- logical
- NA as NULL
- 64-bit values (using "bigint" as field type); the result can be converted to a numeric, which may lose precision,
- character (in both UTF-8 and native encodings), supporting empty strings
- 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)
- time (if supported by the database; returned as objects that inherit from `difftime`)
- timestamp (if supported by the database; returned as `POSIXct` with time zone support)

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.

The interpretation of `rownames` depends on the `row.names` argument, see `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.

---

```
spec_transaction_begin_commit_rollback
    spec_transaction_begin_commit_rollback
```

---

**Description**

spec\_transaction\_begin\_commit\_rollback

**Value**

dbBegin(), dbCommit() and dbRollback() return TRUE, invisibly. 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 table that is missing when the transaction is started but is created and populated during the transaction must exist and contain the data added there 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 table 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.

---

```
spec_transaction_with_transaction
    spec_transaction_with_transaction
```

---

**Description**

spec\_transaction\_with\_transaction

**Value**

dbWithTransaction() returns the value of the executed code. Failure to initiate the transaction (e.g., if the connection is closed or invalid or if 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 dbCommit(). If the code raises an error, the transaction is instead aborted with 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.

---

|          |                      |
|----------|----------------------|
| test_all | <i>Run all tests</i> |
|----------|----------------------|

---

**Description**

test\_all() calls all tests defined in this package (see the section "Tests" below).

test\_some() allows testing one or more tests, it works by constructing the skip argument using negative lookaheads.

**Usage**

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

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

**Arguments**

|      |  |
|------|--|
| skip | [character()]<br>A vector of regular expressions to match against test names; skip test if matching any. |
| ctx  | [DBITestContext]<br>A test context as created by <a href="#">make_context()</a> .                        |
| test | [character]<br>A character vector of regular expressions describing the tests to run.                    |

**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\\_compliance\(\)](#): Test full compliance to DBI

[test\\_stress\(\)](#): Stress tests (not tested with test\_all)

---

|                 |                                    |
|-----------------|------------------------------------|
| test_compliance | <i>Test full compliance to DBI</i> |
|-----------------|------------------------------------|

---

**Description**

Test full compliance to DBI

**Usage**

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

**Arguments**

|      |  |
|------|--|
| skip | [character()]<br>A vector of regular expressions to match against test names; skip test if matching any. |
| ctx  | [DBITest_context]<br>A test context as created by <a href="#">make_context()</a> .                       |

**See Also**

Other tests: [test\\_connection](#), [test\\_driver](#), [test\\_getting\\_started](#), [test\\_meta](#), [test\\_result](#), [test\\_sql](#), [test\\_stress](#), [test\\_transaction](#)

---

|                 |                                    |
|-----------------|------------------------------------|
| test_connection | <i>Test the "Connection" class</i> |
|-----------------|------------------------------------|

---

**Description**

Test the "Connection" class

**Usage**

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

**Arguments**

|      |  |
|------|--|
| skip | [character()]<br>A vector of regular expressions to match against test names; skip test if matching any. |
| ctx  | [DBITest_context]<br>A test context as created by <a href="#">make_context()</a> .                       |

**See Also**

Other tests: [test\\_compliance](#), [test\\_driver](#), [test\\_getting\\_started](#), [test\\_meta](#), [test\\_result](#), [test\\_sql](#), [test\\_stress](#), [test\\_transaction](#)



---

|                |                       |
|----------------|-----------------------|
| test_data_type | <i>test_data_type</i> |
|----------------|-----------------------|

---

**Description**

test\_data\_type

**Usage**

```
test_data_type(ctx, dbObj)
```

**Arguments**

ctx, dbObj      Arguments to internal test function

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

**Specification**

The backend can override the `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.

---

|             |                                |
|-------------|--------------------------------|
| test_driver | <i>Test the "Driver" class</i> |
|-------------|--------------------------------|

---

**Description**

Test the "Driver" class

**Usage**

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

**Arguments**

|      |  |
|------|--|
| skip | [character()]<br>A vector of regular expressions to match against test names; skip test if matching any. |
| ctx  | [DBITest_context]<br>A test context as created by <code>make_context()</code> .                          |

**See Also**

Other tests: [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, ctx = get_default_context())
```

**Arguments**

|      |  |
|------|--|
| skip | [character()]<br>A vector of regular expressions to match against test names; skip test if matching any. |
| ctx  | [DBITest_context]<br>A test context as created by <code>make_context()</code> .                          |

**See Also**

Other tests: [test\\_compliance](#), [test\\_connection](#), [test\\_driver](#), [test\\_meta](#), [test\\_result](#), [test\\_sql](#), [test\\_stress](#), [test\\_transaction](#)

---

|           |                                |
|-----------|--------------------------------|
| test_meta | <i>Test metadata functions</i> |
|-----------|--------------------------------|

---

**Description**

Test metadata functions

**Usage**

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

**Arguments**

|      |  |
|------|--|
| skip | [character()]<br>A vector of regular expressions to match against test names; skip test if matching any. |
| ctx  | [DBITest_context]<br>A test context as created by <a href="#">make_context()</a> .                       |

**See Also**

Other tests: [test\\_compliance](#), [test\\_connection](#), [test\\_driver](#), [test\\_getting\\_started](#), [test\\_result](#), [test\\_sql](#), [test\\_stress](#), [test\\_transaction](#)

---

|             |                                |
|-------------|--------------------------------|
| test_result | <i>Test the "Result" class</i> |
|-------------|--------------------------------|

---

**Description**

Test the "Result" class

**Usage**

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

**Arguments**

|      |  |
|------|--|
| skip | [character()]<br>A vector of regular expressions to match against test names; skip test if matching any. |
| ctx  | [DBITest_context]<br>A test context as created by <a href="#">make_context()</a> .                       |

**See Also**

Other tests: [test\\_compliance](#), [test\\_connection](#), [test\\_driver](#), [test\\_getting\\_started](#), [test\\_meta](#), [test\\_sql](#), [test\\_stress](#), [test\\_transaction](#)

---

|          |                         |
|----------|-------------------------|
| test_sql | <i>Test SQL methods</i> |
|----------|-------------------------|

---

**Description**

Test SQL methods

**Usage**

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

**Arguments**

|      |  |
|------|--|
| skip | [character()]<br>A vector of regular expressions to match against test names; skip test if matching any. |
| ctx  | [DBITest_context]<br>A test context as created by <a href="#">make_context()</a> .                       |

**See Also**

Other tests: [test\\_compliance](#), [test\\_connection](#), [test\\_driver](#), [test\\_getting\\_started](#), [test\\_meta](#), [test\\_result](#), [test\\_stress](#), [test\\_transaction](#)

---

|             |                     |
|-------------|---------------------|
| test_stress | <i>Stress tests</i> |
|-------------|---------------------|

---

**Description**

Stress tests

**Usage**

```
test_stress(skip = NULL, ctx = get_default_context())
```

**Arguments**

|      |  |
|------|--|
| skip | [character()]<br>A vector of regular expressions to match against test names; skip test if matching any. |
| ctx  | [DBITest_context]<br>A test context as created by <a href="#">make_context()</a> .                       |

**See Also**

Other tests: [test\\_compliance](#), [test\\_connection](#), [test\\_driver](#), [test\\_getting\\_started](#), [test\\_meta](#), [test\\_result](#), [test\\_sql](#), [test\\_transaction](#)

---

|                  |                                   |
|------------------|-----------------------------------|
| test_transaction | <i>Test transaction functions</i> |
|------------------|-----------------------------------|

---

### Description

Test transaction functions

### Usage

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

### Arguments

|      |  |
|------|--|
| skip | [character()]<br>A vector of regular expressions to match against test names; skip test if matching any. |
| ctx  | [DBITest_context]<br>A test context as created by <code>make_context()</code> .                          |

### See Also

Other tests: [test\\_compliance](#), [test\\_connection](#), [test\\_driver](#), [test\\_getting\\_started](#), [test\\_meta](#), [test\\_result](#), [test\\_sql](#), [test\\_stress](#)

---

|        |                             |
|--------|-----------------------------|
| tweaks | <i>Tweaks for DBI tests</i> |
|--------|-----------------------------|

---

### Description

TBD.

### 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, "')"), date_typed = TRUE, time_typed = TRUE,
  timestamp_typed = TRUE, temporary_tables = TRUE)
```

**Arguments**

|                        |  |
|------------------------|--|
| ...                    | [any]<br>Unknown tweaks are accepted, with a warning. The ellipsis also asserts that all arguments are named.  |
| constructor_name       | [character(1)]<br>Name of the function that constructs the Driver object.  |
| constructor_relax_args | [logical(1)]<br>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)]<br>Set to TRUE if the DBMS does not support arbitrarily-named identifiers even when quoting is used.  |
| omit_blob_tests        | [logical(1)]<br>Set to TRUE if the DBMS does not support a BLOB data type.   |
| current_needs_parens   | [logical(1)]<br>Set to TRUE if the SQL functions <code>current_date</code> , <code>current_time</code> , and <code>current_timestamp</code> require parentheses. |
| union                  | [function(character)]<br>Function that combines several subqueries into one so that the resulting query returns the concatenated results of the subqueries       |
| placeholder_pattern    | [character]<br>A pattern for placeholders used in <code>dbBind()</code> , e.g., "?", "\$1", or ":name". See <a href="#">make_placeholder_fun()</a> for details.  |
| logical_return         | [function(logical)]<br>A vectorized function that converts logical values to the data type returned by the DBI backend.  |
| date_cast              | [function(character)]<br>A vectorized function that creates an SQL expression for coercing a string to a date value.   |
| time_cast              | [function(character)]<br>A vectorized function that creates an SQL expression for coercing a string to a time value.   |
| timestamp_cast         | [function(character)]<br>A vectorized function that creates an SQL expression for coercing a string to a timestamp value.  |
| date_typed             | [logical(1L)]<br>Set to FALSE if the DBMS doesn't support a dedicated type for dates.  |
| time_typed             | [logical(1L)]<br>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.

# Index

## \*Topic **datasets**

- DBISpec, 4
- as.Date(), 13
- as.POSIXct(), 14
- blob::blob, 7, 8, 21, 25
- character, 7, 8, 13, 16, 17, 25
- data.frame, 8, 12, 13
- Date, 8, 14
- Dates, 7, 25
- DateTimeClasses, 7, 25
- dbBegin(), 22
- dbBind(), 30
- dbClearResult(), 7–10, 14, 15
- dbCommit(), 23
- dbDataType(), 7, 25
- dbDisconnect(), 10
- dbExistsTable(), 19
- dbFetch(), 7–10, 14
- dbGetQuery(), 17, 18
- dbGetRowCount(), 7
- dbGetRowsAffected(), 7, 8, 15
- dbHasCompleted(), 7
- DBIConnection, 4, 6, 10
- DBIDriver, 4
- DBIResult, 4, 7, 10, 14, 15
- DBISpec, 4
- dbIsValid(), 7
- DBItest (DBItest-package), 3
- DBItest-package, 3
- dbListTables(), 16, 19
- dbQuoteIdentifier(), 15, 16, 18–20
- dbReadTable(), 21
- dbRollback(), 23
- dbSendQuery(), 7, 9–11
- dbSendStatement(), 7, 9–12
- dbWriteTable(), 16
- difftime, 7, 25
- factor, 7, 8, 25
- get\_default\_context (make\_context), 4
- hms::as.hms(), 14
- I(), 7, 25
- Inf, 12, 13
- integer, 7, 8, 12, 13, 25
- is.na(), 17
- logical, 7, 8, 13, 25
- make\_context, 4
- make\_context(), 3, 23, 24, 26–29
- make\_placeholder\_fun(), 30
- NA, 8, 14
- numeric, 7, 8, 12, 13, 25
- on.exit(), 7
- ordered, 7, 25
- POSIXct, 8, 14
- POSIXlt, 8
- raw, 7, 8, 13, 25
- rbind(), 8
- rownames, 18, 21
- set\_default\_context (make\_context), 4
- spec\_connection\_disconnect, 5
- spec\_driver\_connect, 6
- spec\_driver\_data\_type, 6
- spec\_meta\_bind, 7
- spec\_meta\_get\_row\_count, 9
- spec\_meta\_get\_rows\_affected, 8
- spec\_meta\_get\_statement, 9
- spec\_meta\_has\_completed, 10
- spec\_meta\_is\_valid, 10



`spec_result_clear_result`, 11  
`spec_result_create_table_with_data_type`,  
11  
`spec_result_execute`, 11  
`spec_result_fetch`, 12  
`spec_result_get_query`, 12  
`spec_result_roundtrip`, 13  
`spec_result_send_query`, 14  
`spec_result_send_statement`, 15  
`spec_sql_exists_table`, 15  
`spec_sql_list_tables`, 16  
`spec_sql_quote_identifier`, 16  
`spec_sql_quote_string`, 17  
`spec_sql_read_table`, 18  
`spec_sql_remove_table`, 19  
`spec_sql_write_table`, 20  
`spec_transaction_begin_commit_rollback`,  
22  
`spec_transaction_with_transaction`, 22  
SQL, 16, 17  
`sqlColumnToRownames()`, 18  
`sqlRownamesToColumn()`, 21

`test_all`, 23  
`test_all()`, 3  
`test_compliance`, 24, 24, 26–29  
`test_compliance()`, 23  
`test_connection`, 24, 24, 26–29  
`test_connection()`, 23  
`test_data_type`, 25  
`test_driver`, 24, 25, 26–29  
`test_driver()`, 23  
`test_getting_started`, 24, 26, 26, 27–29  
`test_getting_started()`, 23  
`test_meta`, 24, 26, 27, 27, 28, 29  
`test_meta()`, 23  
`test_result`, 24, 26, 27, 27, 28, 29  
`test_result()`, 23  
`test_some (test_all)`, 23  
`test_sql`, 24, 26–28, 28, 29  
`test_sql()`, 23  
`test_stress`, 24, 26–28, 28, 29  
`test_stress()`, 23  
`test_transaction`, 24, 26–28, 29  
`test_transaction()`, 23  
`tweaks`, 29  
`tweaks()`, 5