Package 'DBI'

February 25, 2026

Title R Database Interface

Version 1.3.0

Date 2026-02-11

Description A database interface definition for communication between R and relational database management systems. All classes in this package are virtual and need to be extended by the various R/DBMS implementations.

License LGPL (>= 2.1)

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

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

Depends methods, R (>= 3.0.0)

Suggests arrow, blob, callr, covr, DBItest (>= 1.8.2), dbplyr, downlit, dplyr, glue, hms, knitr, magrittr, nanoarrow (>= 0.3.0.1), otel, otelsdk, RMariaDB, rmarkdown, rprojroot, RSQLite (>= 1.1-2), testthat (>= 3.0.0), vctrs, xml2

VignetteBuilder knitr

Config/autostyle/scope line_breaks

Config/autostyle/strict false

Config/Needs/check r-dbi/DBItest

Config/Needs/website r-dbi/DBItest, r-dbi/dbitemplate, adbi, AzureKusto, bigrquery, DatabaseConnector, dittodb, duckdb, implyr, lazysf, odbc, pool, RAthena, IMSMWU/RClickhouse, RH2, RJDBC, RMariaDB, RMySQL, RPostgres, RPostgreSQL, RPresto, RSQLite, sergeant, sparklyr, withr

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.3.9000

NeedsCompilation no

Author R Special Interest Group on Databases (R-SIG-DB) [aut],
Hadley Wickham [aut],
Kirill Müller [aut, cre] (ORCID:
<https://orcid.org/0000-0002-1416-3412>),
R Consortium [fnd]

Maintainer Kirill Müller <kirill@cynkra.com>

Repository CRAN

Date/Publication 2026-02-25 06:31:27 UTC

Contents

DBI-package . . . 3
.SQL92Keywords . . . 4
dbAppendTable . . . 5
dbAppendTableArrow . . . 7
dbBegin . . . 9
dbBind . . . 11
dbCanConnect . . . 16
dbClearResult . . . 17
dbColumnInfo . . . 19
dbConnect . . . 21
dbCreateTable . . . 22
dbCreateTableArrow . . . 24
dbDataType . . . 26
dbDisconnect . . . 28
dbExecute . . . 29
dbExistsTable . . . 32
dbFetch . . . 33
dbFetchArrow . . . 36
dbFetchArrowChunk . . . 38
dbGetConnectArgs . . . 39
dbGetInfo . . . 40
dbGetQuery . . . 41
dbGetQueryArrow . . . 44
dbGetRowCount . . . 46
dbGetRowsAffected . . . 47
dbGetStatement . . . 49
dbHasCompleted . . . 50
DBIConnection-class . . . 52
DBIConnector-class . . . 53
DBIDriver-class . . . 53
DBIObject-class . . . 54
DBIResult-class . . . 55
DBIResultArrow-class . . . 55
dbIsReadOnly . . . 56
dbIsValid . . . 57
dbListFields . . . 58

dbListObjects 59
dbListTables 61
dbQuoteIdentifier 62
dbQuoteLiteral 63
dbQuoteString 65
dbReadTable 66
dbReadTableArrow 68
dbRemoveTable 70
dbSendQuery 72
dbSendQueryArrow 75
dbSendStatement 78
dbUnquoteIdentifier 81
dbWithTransaction 82
dbWriteTable 84
dbWriteTableArrow 87
Id-class 90
rownames 91
SQL 92
sqlAppendTable 93
sqlCreateTable 94
sqlData 95
sqlInterpolate 96

Index **99**

`DBI-package` *DBI: R Database Interface*

Description

DBI defines an interface for communication between R and relational database management systems. All classes in this package are virtual and need to be extended by the various R/DBMS implementations (so-called *DBI backends*).

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 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.

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.

Author(s)

Maintainer: Kirill Müller <kirill@cynkra.com> (ORCID)

Authors:

- R Special Interest Group on Databases (R-SIG-DB)
- Hadley Wickham

Other contributors:

- R Consortium [funder]

See Also

Important generics: `dbConnect()`, `dbGetQuery()`, `dbReadTable()`, `dbWriteTable()`, `dbDisconnect()`

Formal specification (currently work in progress and incomplete): `vignette("spec", package = "DBI")`

Examples

```
RSQLite::SQLite()
```

.SQL92Keywords *Keywords according to the SQL-92 standard*

Description

A character vector of SQL-92 keywords, uppercase.

Usage

```
.SQL92Keywords
```

Format

An object of class `character` of length 220.

Examples

```
"SELECT" %in% .SQL92Keywords
```

dbAppendTable *Insert rows into a table*

Description

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

Usage

```
dbAppendTable(conn, name, value, ..., row.names = NULL)
```

Arguments

conn	A DBI::DBIConnection object, as returned by dbConnect().
name	The table name, passed on to dbQuoteIdentifier(). Options are: • a character string with the unquoted DBMS table name, e.g. "table_name", • a call to Id() with components to the fully qualified table name, e.g. Id(schema = "my_schema", table = "table_name") • a call to SQL() with the quoted and fully qualified table name given verbatim, e.g. SQL('"my_schema"."table_name"')
value	A data.frame (or coercible to data.frame).
...	Other parameters passed on to methods.
row.names	Must be NULL.

Details

Backends compliant to ANSI SQL 99 which use ? as a placeholder for prepared queries don't need to override it. Backends with a different SQL syntax which use ? as a placeholder for prepared queries can override sqlAppendTable(). Other backends (with different placeholders or with entirely different ways to create tables) need to override the dbAppendTable() method.

The row.names argument is not supported by this method. Process the values with sqlRownamesToColumn() before calling this method.

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")
dbCreateTable(con, "iris", iris)
dbAppendTable(con, "iris", iris)
dbReadTable(con, "iris")
dbDisconnect(con)
```

---

dbAppendTableArrow *Insert rows into a table from an Arrow stream*

Description

[Experimental]

The dbAppendTableArrow() method assumes that the table has been created beforehand, e.g. with dbCreateTableArrow(). The default implementation calls dbAppendTable() for each chunk of the stream. Use dbAppendTable() to append data from a data.frame.

Usage

```
dbAppendTableArrow(conn, name, value, ...)
```

Arguments

conn	A DBI::DBIConnection object, as returned by dbConnect().
name	The table name, passed on to dbQuoteIdentifier(). Options are: • a character string with the unquoted DBMS table name, e.g. "table_name", • a call to Id() with components to the fully qualified table name, e.g. Id(schema = "my_schema", table = "table_name") • a call to SQL() with the quoted and fully qualified table name given verbatim, e.g. SQL('"my_schema"."table_name"')
value	An object coercible with nanoarrow::as_nanoarrow_array_stream().
...	Other parameters passed on to methods.

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")
dbCreateTableArrow(con, "iris", iris[0, ])
dbAppendTableArrow(con, "iris", iris[1:5, ])
dbReadTable(con, "iris")
dbDisconnect(con)
```

---

dbBegin *Begin/commit/rollback SQL transactions*

Description

A transaction encapsulates several SQL statements in an atomic unit. It is initiated with dbBegin() and either made persistent with dbCommit() or undone with dbRollback(). In any case, the DBMS guarantees that either all or none of the statements have a permanent effect. This helps ensuring consistency of write operations to multiple tables.

Usage

```
dbBegin(conn, ...)

dbCommit(conn, ...)

dbRollback(conn, ...)
```

Arguments

conn	A DBI::DBIConnection object, as returned by dbConnect().
...	Other parameters passed on to methods.

Details

Not all database engines implement transaction management, in which case these methods should not be implemented for the specific DBIConnection subclass.

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

Self-contained transactions: `dbWithTransaction()`

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "cash", data.frame(amount = 100))
dbWriteTable(con, "account", data.frame(amount = 2000))

# All operations are carried out as logical unit:
dbBegin(con)
withdrawal <- 300
dbExecute(con, "UPDATE cash SET amount = amount + ?", list(withdrawal))
dbExecute(con, "UPDATE account SET amount = amount - ?", list(withdrawal))
dbCommit(con)

dbReadTable(con, "cash")
dbReadTable(con, "account")

# Rolling back after detecting negative value on account:
dbBegin(con)
withdrawal <- 5000
dbExecute(con, "UPDATE cash SET amount = amount + ?", list(withdrawal))
dbExecute(con, "UPDATE account SET amount = amount - ?", list(withdrawal))
if (dbReadTable(con, "account")$amount >= 0) {
```

```
  dbCommit(con)
} else {
  dbRollback(con)
}

dbReadTable(con, "cash")
dbReadTable(con, "account")

dbDisconnect(con)
```

---

dbBind *Bind values to a parameterized/prepared statement*

Description

For parametrized or prepared statements, the dbSendQuery(), dbSendQueryArrow(), and dbSendStatement() functions can be called with statements that contain placeholders for values. The dbBind() and dbBindArrow() functions bind these placeholders to actual values, and are intended to be called on the result set before calling dbFetch() or dbFetchArrow(). The values are passed to dbBind() as lists or data frames, and to dbBindArrow() as a stream created by nanoarrow::as_nanoarrow_array_stream().

[Experimental]

dbBindArrow() is experimental, as are the other *Arrow functions. dbSendQuery() is compatible with dbBindArrow(), and dbSendQueryArrow() is compatible with dbBind().

Usage

```
dbBind(res, params, ...)

dbBindArrow(res, params, ...)
```

Arguments

res	An object inheriting from DBI::DBIResult.
params	For dbBind(), a list of values, named or unnamed, or a data frame, with one element/column per query parameter. For dbBindArrow(), values as a nanoarrow stream, with one column per query parameter.
...	Other arguments passed on to methods.

Details

DBI supports parametrized (or prepared) queries and statements via the dbBind() and dbBindArrow() generics. Parametrized queries are different from normal queries in that they allow an arbitrary number of placeholders, which are later substituted by actual values. Parametrized queries (and statements) serve two purposes:

- The same query can be executed more than once with different values. The DBMS may cache intermediate information for the query, such as the execution plan, and execute it faster.

- Separation of query syntax and parameters protects against SQL injection.

The placeholder format is currently not specified by **DBI**; in the future, a uniform placeholder syntax may be supported. Consult the backend documentation for the supported formats. For automated testing, backend authors specify the placeholder syntax with the `placeholder_pattern` tweak. Known examples are:

- `?` (positional matching in order of appearance) in **RMariaDB** and **RSQLite**
- `$1` (positional matching by index) in **RPostgres** and **RSQLite**
- `:name` and `$name` (named matching) in **RSQLite**

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()`.

The data retrieval flow

This section gives a complete overview over the flow for the execution of queries that return tabular data as data frames.

Most of this flow, except repeated calling of `dbBind()` or `dbBindArrow()`, is implemented by `dbGetQuery()`, which should be sufficient unless you want to access the results in a paged way or you have a parameterized query that you want to reuse. This flow requires an active connection established by `dbConnect()`. See also `vignette("dbi-advanced")` for a walkthrough.

1. Use `dbSendQuery()` to create a result set object of class DBIResult.
2. Optionally, bind query parameters with `dbBind()` or `dbBindArrow()`. This is required only if the query contains placeholders such as `?` or `$1`, depending on the database backend.
3. Optionally, use `dbColumnInfo()` to retrieve the structure of the result set without retrieving actual data.
4. Use `dbFetch()` to get the entire result set, a page of results, or the remaining rows. Fetching zero rows is also possible to retrieve the structure of the result set as a data frame. This step can be called multiple times. Only forward paging is supported, you need to cache previous pages if you need to navigate backwards.
5. Use `dbHasCompleted()` to tell when you're done. This method returns `TRUE` if no more rows are available for fetching.
6. Repeat the last four steps as necessary.
7. Use `dbClearResult()` to clean up the result set object. This step is mandatory even if no rows have been fetched or if an error has occurred during the processing. It is good practice to use `on.exit()` or `withr::defer()` to ensure that this step is always executed.

The data retrieval flow for Arrow streams

This section gives a complete overview over the flow for the execution of queries that return tabular data as an Arrow stream.

Most of this flow, except repeated calling of `dbBindArrow()` or `dbBind()`, is implemented by `dbGetQueryArrow()`, which should be sufficient unless you have a parameterized query that you want to reuse. This flow requires an active connection established by `dbConnect()`. See also `vignette("dbi-advanced")` for a walkthrough.

1. Use dbSendQueryArrow() to create a result set object of class DBIResultArrow.
2. Optionally, bind query parameters with dbBindArrow() or dbBind(). This is required only if the query contains placeholders such as ? or $1, depending on the database backend.
3. Use dbFetchArrow() to get a data stream.
4. Repeat the last two steps as necessary.
5. Use dbClearResult() to clean up the result set object. This step is mandatory even if no rows have been fetched or if an error has occurred during the processing. It is good practice to use on.exit() or withr::defer() to ensure that this step is always executed.

The command execution flow

This section gives a complete overview over the flow for the execution of SQL statements that have side effects such as stored procedures, inserting or deleting data, or setting database or connection options. Most of this flow, except repeated calling of dbBindArrow(), is implemented by dbExecute(), which should be sufficient for non-parameterized queries. This flow requires an active connection established by dbConnect(). See also vignette("dbi-advanced") for a walk-through.

1. Use dbSendStatement() to create a result set object of class DBIResult. For some queries you need to pass immediate = TRUE.
2. Optionally, bind query parameters withdbBind() or dbBindArrow(). This is required only if the query contains placeholders such as ? or $1, depending on the database backend.
3. Optionally, use dbGetRowsAffected() to retrieve the number of rows affected by the query.
4. Repeat the last two steps as necessary.
5. Use dbClearResult() to clean up the result set object. This step is mandatory even if no rows have been fetched or if an error has occurred during the processing. It is good practice to use on.exit() or withr::defer() to ensure that this step is always executed.

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.

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)
- Date (also when stored internally as integer)
- 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

See Also

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

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

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

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

Examples

```
# Data frame flow:
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "iris", iris)

# Using the same query for different values
iris_result <- dbSendQuery(con, "SELECT * FROM iris WHERE [Petal.Width] > ?")
dbBind(iris_result, list(2.3))
dbFetch(iris_result)
dbBind(iris_result, list(3))
dbFetch(iris_result)
dbClearResult(iris_result)

# Executing the same statement with different values at once
iris_result <- dbSendStatement(con, "DELETE FROM iris WHERE [Species] = $species")
dbBind(iris_result, list(species = c("setosa", "versicolor", "unknown")))
dbGetRowsAffected(iris_result)
dbClearResult(iris_result)

nrow(dbReadTable(con, "iris"))

dbDisconnect(con)



# Arrow flow:
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "iris", iris)

# Using the same query for different values
iris_result <- dbSendQueryArrow(con, "SELECT * FROM iris WHERE [Petal.Width] > ?")
dbBindArrow(
  iris_result,
  nanoarrow::as_nanoarrow_array_stream(data.frame(2.3, fix.empty.names = FALSE))
)
as.data.frame(dbFetchArrow(iris_result))
```

```
dbBindArrow(
  iris_result,
  nanoarrow::as_nanoarrow_array_stream(data.frame(3, fix.empty.names = FALSE))
)
as.data.frame(dbFetchArrow(iris_result))
dbClearResult(iris_result)

# Executing the same statement with different values at once
iris_result <- dbSendStatement(con, "DELETE FROM iris WHERE [Species] = $species")
dbBindArrow(iris_result, nanoarrow::as_nanoarrow_array_stream(data.frame(
  species = c("setosa", "versicolor", "unknown")
)))
dbGetRowsAffected(iris_result)
dbClearResult(iris_result)

nrow(dbReadTable(con, "iris"))

dbDisconnect(con)
```

`dbCanConnect` *Check if a connection to a DBMS can be established*

Description

Like `dbConnect()`, but only checks validity without actually returning a connection object. The default implementation opens a connection and disconnects on success, but individual backends might implement a lighter-weight check.

Usage

```
dbCanConnect(drv, ...)
```

Arguments

`drv`	An object that inherits from DBI::DBIDriver, or an existing DBI::DBIConnection object (in order to clone an existing connection).
`...`	Authentication arguments needed by the DBMS instance; these typically include `user`, `password`, `host`, `port`, `dbname`, etc. For details see the appropriate `DBIDriver`.

Value

A scalar logical. If `FALSE`, the `"reason"` attribute indicates a reason for failure.

See Also

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

Examples

```
# SQLite only needs a path to the database. (Here, ":memory:" is a special
# path that creates an in-memory database.) Other database drivers
# will require more details (like user, password, host, port, etc.)
dbCanConnect(RSQLite::SQLite(), ":memory:")
```

`dbClearResult` *Clear a result set*

Description

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

Usage

```
dbClearResult(res, ...)
```

Arguments

`res`	An object inheriting from DBI::DBIResult.
`...`	Other arguments passed on to methods.

Value

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

The data retrieval flow

This section gives a complete overview over the flow for the execution of queries that return tabular data as data frames.

Most of this flow, except repeated calling of `dbBind()` or `dbBindArrow()`, is implemented by `dbGetQuery()`, which should be sufficient unless you want to access the results in a paged way or you have a parameterized query that you want to reuse. This flow requires an active connection established by `dbConnect()`. See also `vignette("dbi-advanced")` for a walkthrough.

1. Use `dbSendQuery()` to create a result set object of class DBIResult.
2. Optionally, bind query parameters with `dbBind()` or `dbBindArrow()`. This is required only if the query contains placeholders such as `?` or `$1`, depending on the database backend.
3. Optionally, use `dbColumnInfo()` to retrieve the structure of the result set without retrieving actual data.
4. Use `dbFetch()` to get the entire result set, a page of results, or the remaining rows. Fetching zero rows is also possible to retrieve the structure of the result set as a data frame. This step can be called multiple times. Only forward paging is supported, you need to cache previous pages if you need to navigate backwards.

5. Use dbHasCompleted() to tell when you're done. This method returns TRUE if no more rows are available for fetching.
6. Repeat the last four steps as necessary.
7. Use dbClearResult() to clean up the result set object. This step is mandatory even if no rows have been fetched or if an error has occurred during the processing. It is good practice to use on.exit() or withr::defer() to ensure that this step is always executed.

The command execution flow

This section gives a complete overview over the flow for the execution of SQL statements that have side effects such as stored procedures, inserting or deleting data, or setting database or connection options. Most of this flow, except repeated calling of dbBindArrow(), is implemented by dbExecute(), which should be sufficient for non-parameterized queries. This flow requires an active connection established by dbConnect(). See also vignette("dbi-advanced") for a walk-through.

1. Use dbSendStatement() to create a result set object of class DBIResult. For some queries you need to pass immediate = TRUE.
2. Optionally, bind query parameters withdbBind() or dbBindArrow(). This is required only if the query contains placeholders such as ? or $1, depending on the database backend.
3. Optionally, use dbGetRowsAffected() to retrieve the number of rows affected by the query.
4. Repeat the last two steps as necessary.
5. Use dbClearResult() to clean up the result set object. This step is mandatory even if no rows have been fetched or if an error has occurred during the processing. It is good practice to use on.exit() or withr::defer() to ensure that this step is always executed.

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

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

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

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

rs <- dbSendQuery(con, "SELECT 1")
print(dbFetch(rs))

dbClearResult(rs)
dbDisconnect(con)
```

---

dbColumnInfo *Information about result types*

Description

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

Usage

```
dbColumnInfo(res, ...)
```

Arguments

res	An object inheriting from DBI::DBIResult.
...	Other arguments passed on to methods.

Value

`dbColumnInfo()` returns a data frame with at least two columns `"name"` and `"type"` (in that order) (and optional columns that start with a dot). The `"name"` and `"type"` columns contain the names and types of the R columns of the data frame that is returned from `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.

The data retrieval flow

This section gives a complete overview over the flow for the execution of queries that return tabular data as data frames.

Most of this flow, except repeated calling of `dbBind()` or `dbBindArrow()`, is implemented by `dbGetQuery()`, which should be sufficient unless you want to access the results in a paged way or you have a parameterized query that you want to reuse. This flow requires an active connection established by `dbConnect()`. See also `vignette("dbi-advanced")` for a walkthrough.

1. Use `dbSendQuery()` to create a result set object of class DBIResult.

2. Optionally, bind query parameters with dbBind() or dbBindArrow(). This is required only if the query contains placeholders such as ? or $1, depending on the database backend.
3. Optionally, use dbColumnInfo() to retrieve the structure of the result set without retrieving actual data.
4. Use dbFetch() to get the entire result set, a page of results, or the remaining rows. Fetching zero rows is also possible to retrieve the structure of the result set as a data frame. This step can be called multiple times. Only forward paging is supported, you need to cache previous pages if you need to navigate backwards.
5. Use dbHasCompleted() to tell when you're done. This method returns TRUE if no more rows are available for fetching.
6. Repeat the last four steps as necessary.
7. Use dbClearResult() to clean up the result set object. This step is mandatory even if no rows have been fetched or if an error has occurred during the processing. It is good practice to use on.exit() or withr::defer() to ensure that this step is always executed.

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

rs <- dbSendQuery(con, "SELECT 1 AS a, 2 AS b")
dbColumnInfo(rs)
dbFetch(rs)

dbClearResult(rs)
dbDisconnect(con)
```

dbConnect *Create a connection to a DBMS*

Description

Connect to a DBMS going through the appropriate authentication procedure. Some implementations may allow you to have multiple connections open, so you may invoke this function repeatedly assigning its output to different objects. The authentication mechanism is left unspecified, so check the documentation of individual drivers for details. Use `dbCanConnect()` to check if a connection can be established.

Usage

```
dbConnect(drv, ...)
```

Arguments

`drv`	An object that inherits from DBI::DBIDriver, or an existing DBI::DBIConnection object (in order to clone an existing connection).
`...`	Authentication arguments needed by the DBMS instance; these typically include `user`, `password`, `host`, `port`, `dbname`, etc. For details see the appropriate `DBIDriver`.

Value

`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

dbDisconnect() to disconnect from a database.

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

Other DBIConnector generics: DBIConnector-class, dbDataType(), dbGetConnectArgs(), dbIsReadOnly()

Examples

```
# SQLite only needs a path to the database. (Here, ":memory:" is a special
# path that creates an in-memory database.) Other database drivers
# will require more details (like user, password, host, port, etc.)
con <- dbConnect(RSQLite::SQLite(), ":memory:")
con

dbListTables(con)

dbDisconnect(con)

# Bad, for subtle reasons:
# This code fails when RSQLite isn't loaded yet,
# because dbConnect() doesn't know yet about RSQLite.
dbListTables(con <- dbConnect(RSQLite::SQLite(), ":memory:"))
```

---

dbCreateTable *Create a table in the database*

Description

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

Usage

```
dbCreateTable(conn, name, fields, ..., row.names = NULL, temporary = FALSE)
```

Arguments

conn	A DBI::DBIConnection object, as returned by dbConnect().
name	The table name, passed on to dbQuoteIdentifier(). Options are: • a character string with the unquoted DBMS table name, e.g. "table_name",

- a call to `Id()` with components to the fully qualified table name, e.g. `Id(schema = "my_schema", table = "table_name")`
- a call to `SQL()` with the quoted and fully qualified table name given verbatim, e.g. `SQL('"my_schema"."table_name"')`

`fields` Either a character vector or a data frame.

A named character vector: Names are column names, values are types. Names are escaped with `dbQuoteIdentifier()`. Field types are unescaped.

A data frame: field types are generated using `dbDataType()`.

`...` Other parameters passed on to methods.

`row.names` Must be `NULL`.

`temporary` If `TRUE`, will generate a temporary table.

Details

Backends compliant to ANSI SQL 99 don't need to override it. Backends with a different SQL syntax can override `sqlCreateTable()`, backends with entirely different ways to create tables need to override this method.

The `row.names` argument is not supported by this method. Process the values with `sqlRownamesToColumn()` before calling this method.

The argument order is different from the `sqlCreateTable()` method, the latter will be adapted in a later release of DBI.

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")
dbCreateTable(con, "iris", iris)
dbReadTable(con, "iris")
dbDisconnect(con)
```

---

dbCreateTableArrow *Create a table in the database based on an Arrow object*

Description

[Experimental]

The default dbCreateTableArrow() method determines the R data types of the Arrow schema associated with the Arrow object, and calls dbCreateTable(). Backends that implement dbAppendTableArrow() should typically also implement this generic. Use dbCreateTable() to create a table from the column types as defined in a data frame.

Usage

```
dbCreateTableArrow(conn, name, value, ..., temporary = FALSE)
```

Arguments

conn	A DBI::DBIConnection object, as returned by dbConnect().
name	The table name, passed on to dbQuoteIdentifier(). Options are: • a character string with the unquoted DBMS table name, e.g. "table_name", • a call to Id() with components to the fully qualified table name, e.g. Id(schema = "my_schema", table = "table_name") • a call to SQL() with the quoted and fully qualified table name given verbatim, e.g. SQL('"my_schema"."table_name"')
value	An object for which a schema can be determined via nanoarrow::infer_nanoarrow_schema().
...	Other parameters passed on to methods.
temporary	If TRUE, will generate a temporary table.

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")
ptype <- data.frame(a = numeric())
dbCreateTableArrow(con, "df", nanoarrow::infer_nanoarrow_schema(ptype))
dbReadTable(con, "df")
dbDisconnect(con)
```

`dbDataType` *Determine the SQL data type of an object*

Description

Returns an SQL string that describes the SQL data type to be used for an object. The default implementation of this generic determines the SQL type of an R object according to the SQL 92 specification, which may serve as a starting point for driver implementations. DBI also provides an implementation for data.frame which will return a character vector giving the type for each column in the dataframe.

Usage

```
dbDataType(dbObj, obj, ...)
```

Arguments

dbObj	A object inheriting from DBI::DBIDriver or DBI::DBIConnection
obj	An R object whose SQL type we want to determine.
...	Other arguments passed on to methods.

Details

The data types supported by databases are different than the data types in R, but the mapping between the primitive types is straightforward:

- Any of the many fixed and varying length character types are mapped to character vectors
- Fixed-precision (non-IEEE) numbers are mapped into either numeric or integer vectors.

Notice that many DBMS do not follow IEEE arithmetic, so there are potential problems with under/overflows and loss of precision.

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.

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

See Also

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

Other DBIConnection generics: `DBIConnection-class`, `dbAppendTable()`, `dbAppendTableArrow()`, `dbCreateTable()`, `dbCreateTableArrow()`, `dbDisconnect()`, `dbExecute()`, `dbExistsTable()`, `dbGetException()`, `dbGetInfo()`, `dbGetQuery()`, `dbGetQueryArrow()`, `dbIsReadOnly()`, `dbIsValid()`, `dbListFields()`, `dbListObjects()`, `dbListResults()`, `dbListTables()`, `dbQuoteIdentifier()`,

dbReadTable(), dbReadTableArrow(), dbRemoveTable(), dbSendQuery(), dbSendQueryArrow(), dbSendStatement(), dbUnquoteIdentifier(), dbWriteTable(), dbWriteTableArrow()

Other DBIConnector generics: DBIConnector-class, dbConnect(), dbGetConnectArgs(), dbIsReadOnly()

Examples

```
dbDataType(ANSI(), 1:5)
dbDataType(ANSI(), 1)
dbDataType(ANSI(), TRUE)
dbDataType(ANSI(), Sys.Date())
dbDataType(ANSI(), Sys.time())
dbDataType(ANSI(), Sys.time() - as.POSIXct(Sys.Date()))
dbDataType(ANSI(), c("x", "abc"))
dbDataType(ANSI(), list(raw(10), raw(20)))
dbDataType(ANSI(), I(3))

dbDataType(ANSI(), iris)


con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbDataType(con, 1:5)
dbDataType(con, 1)
dbDataType(con, TRUE)
dbDataType(con, Sys.Date())
dbDataType(con, Sys.time())
dbDataType(con, Sys.time() - as.POSIXct(Sys.Date()))
dbDataType(con, c("x", "abc"))
dbDataType(con, list(raw(10), raw(20)))
dbDataType(con, I(3))

dbDataType(con, iris)

dbDisconnect(con)
```

---

dbDisconnect *Disconnect (close) a connection*

Description

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

Usage

```
dbDisconnect(conn, ...)
```

Arguments

conn	A DBI::DBIConnection object, as returned by dbConnect().
...	Other parameters passed on to methods.

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")
dbDisconnect(con)
```

---

dbExecute *Change database state*

Description

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

Usage

```
dbExecute(conn, statement, ...)
```

Arguments

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

Details

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

Value

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

Implementation notes

Subclasses should override this method only if they provide some sort of performance optimization.

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

For queries: `dbSendQuery()` and `dbGetQuery()`.

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

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "cars", head(cars, 3))
dbReadTable(con, "cars")   # there are 3 rows
dbExecute(
  con,
  "INSERT INTO cars (speed, dist) VALUES (1, 1), (2, 2), (3, 3)"
)
dbReadTable(con, "cars")   # there are now 6 rows

# Pass values using the param argument:
dbExecute(
  con,
  "INSERT INTO cars (speed, dist) VALUES (?, ?)",
  params = list(4:7, 5:8)
)
dbReadTable(con, "cars")   # there are now 10 rows

dbDisconnect(con)
```

dbExistsTable *Does a table exist?*

Description

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

Usage

```
dbExistsTable(conn, name, ...)
```

Arguments

conn	A DBI::DBIConnection object, as returned by dbConnect().
name	The table name, passed on to dbQuoteIdentifier(). Options are: • a character string with the unquoted DBMS table name, e.g. "table_name", • a call to Id() with components to the fully qualified table name, e.g. Id(schema = "my_schema", table = "table_name") • a call to SQL() with the quoted and fully qualified table name given verbatim, e.g. SQL('"my_schema"."table_name"')
...	Other parameters passed on to methods.

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbExistsTable(con, "iris")
dbWriteTable(con, "iris", iris)
dbExistsTable(con, "iris")

dbDisconnect(con)
```

---

dbFetch *Fetch records from a previously executed query*

Description

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

Usage

```
dbFetch(res, n = -1, ...)

fetch(res, n = -1, ...)
```

Arguments

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

Details

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

Value

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

The data retrieval flow

This section gives a complete overview over the flow for the execution of queries that return tabular data as data frames.

Most of this flow, except repeated calling of dbBind() or dbBindArrow(), is implemented by dbGetQuery(), which should be sufficient unless you want to access the results in a paged way or you have a parameterized query that you want to reuse. This flow requires an active connection established by dbConnect(). See also vignette("dbi-advanced") for a walkthrough.

1. Use dbSendQuery() to create a result set object of class DBIResult.
2. Optionally, bind query parameters with dbBind() or dbBindArrow(). This is required only if the query contains placeholders such as ? or $1, depending on the database backend.
3. Optionally, use dbColumnInfo() to retrieve the structure of the result set without retrieving actual data.
4. Use dbFetch() to get the entire result set, a page of results, or the remaining rows. Fetching zero rows is also possible to retrieve the structure of the result set as a data frame. This step can be called multiple times. Only forward paging is supported, you need to cache previous pages if you need to navigate backwards.
5. Use dbHasCompleted() to tell when you're done. This method returns TRUE if no more rows are available for fetching.
6. Repeat the last four steps as necessary.
7. Use dbClearResult() to clean up the result set object. This step is mandatory even if no rows have been fetched or if an error has occurred during the processing. It is good practice to use on.exit() or withr::defer() to ensure that this step is always executed.

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.

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:

- 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

See Also

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

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

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars)

# Fetch all results
rs <- dbSendQuery(con, "SELECT * FROM mtcars WHERE cyl = 4")
dbFetch(rs)
dbClearResult(rs)

# Fetch in chunks
rs <- dbSendQuery(con, "SELECT * FROM mtcars")
while (!dbHasCompleted(rs)) {
  chunk <- dbFetch(rs, 10)
  print(nrow(chunk))
}

dbClearResult(rs)
dbDisconnect(con)
```

`dbFetchArrow` *Fetch records from a previously executed query as an Arrow object*

Description

[Experimental]

Fetch the result set and return it as an Arrow object. Use `dbFetchArrowChunk()` to fetch results in chunks.

Usage

```
dbFetchArrow(res, ...)
```

Arguments

`res`	An object inheriting from DBI::DBIResultArrow, created by `dbSendQueryArrow()`.
`...`	Other arguments passed on to methods.

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.

The data retrieval flow for Arrow streams

This section gives a complete overview over the flow for the execution of queries that return tabular data as an Arrow stream.

Most of this flow, except repeated calling of `dbBindArrow()` or `dbBind()`, is implemented by `dbGetQueryArrow()`, which should be sufficient unless you have a parameterized query that you want to reuse. This flow requires an active connection established by `dbConnect()`. See also `vignette("dbi-advanced")` for a walkthrough.

1. Use `dbSendQueryArrow()` to create a result set object of class DBIResultArrow.
2. Optionally, bind query parameters with `dbBindArrow()` or `dbBind()`. This is required only if the query contains placeholders such as `?` or `$1`, depending on the database backend.
3. Use `dbFetchArrow()` to get a data stream.
4. Repeat the last two steps as necessary.
5. Use `dbClearResult()` to clean up the result set object. This step is mandatory even if no rows have been fetched or if an error has occurred during the processing. It is good practice to use `on.exit()` or `withr::defer()` to ensure that this step is always executed.

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

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

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

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars)

# Fetch all results
rs <- dbSendQueryArrow(con, "SELECT * FROM mtcars WHERE cyl = 4")
as.data.frame(dbFetchArrow(rs))
dbClearResult(rs)

dbDisconnect(con)
```

dbFetchArrowChunk *Fetch the next batch of records from a previously executed query as an Arrow object*

Description

[Experimental]

Fetch the next chunk of the result set and return it as an Arrow object. The chunk size is implementation-specific. Use `dbFetchArrow()` to fetch all results.

Usage

```
dbFetchArrowChunk(res, ...)
```

Arguments

`res`	An object inheriting from DBI::DBIResultArrow, created by `dbSendQueryArrow()`.
`...`	Other arguments passed on to methods.

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.

The data retrieval flow for Arrow streams

This section gives a complete overview over the flow for the execution of queries that return tabular data as an Arrow stream.

Most of this flow, except repeated calling of `dbBindArrow()` or `dbBind()`, is implemented by `dbGetQueryArrow()`, which should be sufficient unless you have a parameterized query that you want to reuse. This flow requires an active connection established by `dbConnect()`. See also `vignette("dbi-advanced")` for a walkthrough.

1. Use `dbSendQueryArrow()` to create a result set object of class DBIResultArrow.
2. Optionally, bind query parameters with `dbBindArrow()` or `dbBind()`. This is required only if the query contains placeholders such as ? or $1, depending on the database backend.
3. Use `dbFetchArrow()` to get a data stream.
4. Repeat the last two steps as necessary.
5. Use `dbClearResult()` to clean up the result set object. This step is mandatory even if no rows have been fetched or if an error has occurred during the processing. It is good practice to use `on.exit()` or `withr::defer()` to ensure that this step is always executed.

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

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

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

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars)

# Fetch all results
rs <- dbSendQueryArrow(con, "SELECT * FROM mtcars WHERE cyl = 4")
dbHasCompleted(rs)
as.data.frame(dbFetchArrowChunk(rs))
dbHasCompleted(rs)
as.data.frame(dbFetchArrowChunk(rs))
dbClearResult(rs)

dbDisconnect(con)
```

---

dbGetConnectArgs *Get connection arguments*

Description

Returns the arguments stored in a DBIConnector object for inspection, optionally evaluating them. This function is called by `dbConnect()` and usually does not need to be called directly.

Usage

```
dbGetConnectArgs(drv, eval = TRUE, ...)
```

Arguments

`drv`	A object inheriting from DBIConnector.
`eval`	Set to `FALSE` to return the functions that generate the argument instead of evaluating them.
`...`	Other arguments passed on to methods. Not otherwise used.

See Also

Other DBIConnector generics: DBIConnector-class, dbConnect(), dbDataType(), dbIsReadOnly()

Examples

```
cnr <- new("DBIConnector",
  .drv = RSQLite::SQLite(),
  .conn_args = list(dbname = ":memory:", password = function() "supersecret")
)
dbGetConnectArgs(cnr)
dbGetConnectArgs(cnr, eval = FALSE)
```

---

dbGetInfo *Get DBMS metadata*

Description

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

Usage

```
dbGetInfo(dbObj, ...)
```

Arguments

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

Value

For objects of class 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().

Implementation notes

The default implementation for DBIResult objects constructs such a list from the return values of the corresponding methods, dbGetStatement(), dbGetRowCount(), dbGetRowsAffected(), and dbHasCompleted().

See Also

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

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

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

Examples

```
dbGetInfo(RSQLite::SQLite())
```

---

dbGetQuery *Retrieve results from a query*

Description

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

Usage

```
dbGetQuery(conn, statement, ...)
```

Arguments

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

Details

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

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

Value

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

Implementation notes

Subclasses should override this method only if they provide some sort of performance optimization.

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

For updates: `dbSendStatement()` and `dbExecute()`.

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

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars)
dbGetQuery(con, "SELECT * FROM mtcars")
dbGetQuery(con, "SELECT * FROM mtcars", n = 6)

# Pass values using the param argument:
# (This query runs eight times, once for each different
# parameter. The resulting rows are combined into a single
# data frame.)
dbGetQuery(
  con,
  "SELECT COUNT(*) FROM mtcars WHERE cyl = ?",
  params = list(1:8)
)

dbDisconnect(con)
```

---

dbGetQueryArrow *Retrieve results from a query as an Arrow object*

Description

[Experimental]

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

Usage

```
dbGetQueryArrow(conn, statement, ...)
```

Arguments

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

Details

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

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

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.

Implementation notes

Subclasses should override this method only if they provide some sort of performance optimization.

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

For updates: `dbSendStatement()` and `dbExecute()`.

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

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

Examples

```
# Retrieve data as arrow table
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars)
dbGetQueryArrow(con, "SELECT * FROM mtcars")

dbDisconnect(con)
```

`dbGetRowCount` *The number of rows fetched so far*

Description

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

Usage

```
dbGetRowCount(res, ...)
```

Arguments

`res`	An object inheriting from DBI::DBIResult.
`...`	Other arguments passed on to methods.

Value

dbGetRowCount() returns a scalar number (integer or numeric), the number of rows fetched so far. After calling 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 DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteLiteral(), dbQuoteString()

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars)
rs <- dbSendQuery(con, "SELECT * FROM mtcars")

dbGetRowCount(rs)
ret1 <- dbFetch(rs, 10)
dbGetRowCount(rs)
ret2 <- dbFetch(rs)
dbGetRowCount(rs)
nrow(ret1) + nrow(ret2)

dbClearResult(rs)
dbDisconnect(con)
```

---

dbGetRowsAffected *The number of rows affected*

Description

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

Usage

```
dbGetRowsAffected(res, ...)
```

Arguments

res	An object inheriting from DBI::DBIResult.
...	Other arguments passed on to methods.

Value

dbGetRowsAffected() returns a scalar number (integer or numeric), the number of rows affected by a data manipulation statement issued with 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.

The command execution flow

This section gives a complete overview over the flow for the execution of SQL statements that have side effects such as stored procedures, inserting or deleting data, or setting database or connection options. Most of this flow, except repeated calling of dbBindArrow(), is implemented by dbExecute(), which should be sufficient for non-parameterized queries. This flow requires an active connection established by dbConnect(). See also vignette("dbi-advanced") for a walk-through.

1. Use dbSendStatement() to create a result set object of class DBIResult. For some queries you need to pass immediate = TRUE.
2. Optionally, bind query parameters withdbBind() or dbBindArrow(). This is required only if the query contains placeholders such as ? or $1, depending on the database backend.
3. Optionally, use dbGetRowsAffected() to retrieve the number of rows affected by the query.
4. Repeat the last two steps as necessary.
5. Use dbClearResult() to clean up the result set object. This step is mandatory even if no rows have been fetched or if an error has occurred during the processing. It is good practice to use on.exit() or withr::defer() to ensure that this step is always executed.

Failure modes

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

See Also

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

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars)
rs <- dbSendStatement(con, "DELETE FROM mtcars")
dbGetRowsAffected(rs)
nrow(mtcars)

dbClearResult(rs)
dbDisconnect(con)
```

---

dbGetStatement *Get the statement associated with a result set*

Description

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

Usage

```
dbGetStatement(res, ...)
```

Arguments

`res`	An object inheriting from DBI::DBIResult.
`...`	Other arguments passed on to methods.

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars)
rs <- dbSendQuery(con, "SELECT * FROM mtcars")
dbGetStatement(rs)

dbClearResult(rs)
dbDisconnect(con)
```

---

dbHasCompleted *Completion status*

Description

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

Usage

```
dbHasCompleted(res, ...)
```

Arguments

res	An object inheriting from DBI::DBIResult.
...	Other arguments passed on to methods.

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.

The data retrieval flow

This section gives a complete overview over the flow for the execution of queries that return tabular data as data frames.

Most of this flow, except repeated calling of dbBind() or dbBindArrow(), is implemented by dbGetQuery(), which should be sufficient unless you want to access the results in a paged way or you have a parameterized query that you want to reuse. This flow requires an active connection established by dbConnect(). See also vignette("dbi-advanced") for a walkthrough.

1. Use dbSendQuery() to create a result set object of class DBIResult.
2. Optionally, bind query parameters with dbBind() or dbBindArrow(). This is required only if the query contains placeholders such as ? or $1, depending on the database backend.

3. Optionally, use dbColumnInfo() to retrieve the structure of the result set without retrieving actual data.
4. Use dbFetch() to get the entire result set, a page of results, or the remaining rows. Fetching zero rows is also possible to retrieve the structure of the result set as a data frame. This step can be called multiple times. Only forward paging is supported, you need to cache previous pages if you need to navigate backwards.
5. Use dbHasCompleted() to tell when you're done. This method returns TRUE if no more rows are available for fetching.
6. Repeat the last four steps as necessary.
7. Use dbClearResult() to clean up the result set object. This step is mandatory even if no rows have been fetched or if an error has occurred during the processing. It is good practice to use on.exit() or withr::defer() to ensure that this step is always executed.

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

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

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars)
rs <- dbSendQuery(con, "SELECT * FROM mtcars")

dbHasCompleted(rs)
ret1 <- dbFetch(rs, 10)
dbHasCompleted(rs)
ret2 <- dbFetch(rs)
dbHasCompleted(rs)
```

```
dbClearResult(rs)
dbDisconnect(con)
```

DBIConnection-class *DBIConnection class*

Description

This virtual class encapsulates the connection to a DBMS, and it provides access to dynamic queries, result sets, DBMS session management (transactions), etc.

Implementation note

Individual drivers are free to implement single or multiple simultaneous connections.

See Also

Other DBI classes: DBIConnector-class, DBIDriver-class, DBIObject-class, DBIResult-class, DBIResultArrow-class

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")
con
dbDisconnect(con)
## Not run:
con <- dbConnect(RPostgreSQL::PostgreSQL(), "username", "password")
con
dbDisconnect(con)

## End(Not run)
```

DBIConnector-class *DBIConnector class*

Description

Wraps objects of the DBIDriver class to include connection options. The purpose of this class is to store both the driver and the connection options. A database connection can be established with a call to `dbConnect()`, passing only that object without additional arguments.

Details

To prevent leakage of passwords and other credentials, this class supports delayed evaluation. All arguments can optionally be a function (callable without arguments). In such a case, the function is evaluated transparently when connecting in `dbGetConnectArgs()`.

See Also

Other DBI classes: `DBIConnection-class`, `DBIDriver-class`, `DBIObject-class`, `DBIResult-class`, `DBIResultArrow-class`

Other DBIConnector generics: `dbConnect()`, `dbDataType()`, `dbGetConnectArgs()`, `dbIsReadOnly()`

Examples

```
# Create a connector:
cnr <- new("DBIConnector",
  .drv = RSQLite::SQLite(),
  .conn_args = list(dbname = ":memory:")
)
cnr

# Establish a connection through this connector:
con <- dbConnect(cnr)
con

# Access the database through this connection:
dbGetQuery(con, "SELECT 1 AS a")
dbDisconnect(con)
```

DBIDriver-class *DBIDriver class*

Description

Base class for all DBMS drivers (e.g., RSQLite, MySQL, PostgreSQL). The virtual class `DBIDriver` defines the operations for creating connections and defining data type mappings. Actual driver classes, for instance `RPostgres`, `RMariaDB`, etc. implement these operations in a DBMS-specific manner.

See Also

Other DBI classes: DBIConnection-class, DBIConnector-class, DBIObject-class, DBIResult-class, DBIResultArrow-class

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

DBIObject-class *DBIObject class*

Description

Base class for all other DBI classes (e.g., drivers, connections). This is a virtual Class: No objects may be created from it.

Details

More generally, the DBI defines a very small set of classes and generics that allows users and applications access DBMS with a common interface. The virtual classes are DBIDriver that individual drivers extend, DBIConnection that represent instances of DBMS connections, and DBIResult that represent the result of a DBMS statement. These three classes extend the basic class of DBIObject, which serves as the root or parent of the class hierarchy.

Implementation notes

An implementation MUST provide methods for the following generics:

- dbGetInfo().

It MAY also provide methods for:

- summary(). Print a concise description of the object. The default method invokes dbGetInfo(dbObj) and prints the name-value pairs one per line. Individual implementations may tailor this appropriately.

See Also

Other DBI classes: DBIConnection-class, DBIConnector-class, DBIDriver-class, DBIResult-class, DBIResultArrow-class

Examples

```
drv <- RSQLite::SQLite()
con <- dbConnect(drv)

rs <- dbSendQuery(con, "SELECT 1")
is(drv, "DBIObject")   ## True
is(con, "DBIObject")   ## True
is(rs, "DBIObject")
```

```
dbClearResult(rs)
dbDisconnect(con)
```

DBIResult-class *DBIResult class*

Description

This virtual class describes the result and state of execution of a DBMS statement (any statement, query or non-query). The result set keeps track of whether the statement produces output how many rows were affected by the operation, how many rows have been fetched (if statement is a query), whether there are more rows to fetch, etc.

Implementation notes

Individual drivers are free to allow single or multiple active results per connection.

The default show method displays a summary of the query using other DBI generics.

See Also

Other DBI classes: DBIConnection-class, DBIConnector-class, DBIDriver-class, DBIObject-class, DBIResultArrow-class

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

DBIResultArrow-class *DBIResultArrow class*

Description

[Experimental]

This virtual class describes the result and state of execution of a DBMS statement (any statement, query or non-query) for returning data as an Arrow object.

Implementation notes

Individual drivers are free to allow single or multiple active results per connection.

The default show method displays a summary of the query using other DBI generics.

See Also

Other DBI classes: DBIConnection-class, DBIConnector-class, DBIDriver-class, DBIObject-class, DBIResult-class

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

`dbIsReadOnly` *Is this DBMS object read only?*

Description

This generic tests whether a database object is read only.

Usage

```
dbIsReadOnly(dbObj, ...)
```

Arguments

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

See Also

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

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

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

Other DBIConnector generics: DBIConnector-class, dbConnect(), dbDataType(), dbGetConnectArgs()

Examples

```
dbIsReadOnly(ANSI())
```

dbIsValid *Is this DBMS object still valid?*

Description

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

Usage

```
dbIsValid(dbObj, ...)
```

Arguments

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

Value

`dbIsValid()` returns a logical scalar, `TRUE` if the object specified by `dbObj` is valid, `FALSE` otherwise. A 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 DBIDriver generics: `DBIDriver-class`, `dbCanConnect()`, `dbConnect()`, `dbDataType()`, `dbDriver()`, `dbGetInfo()`, `dbIsReadOnly()`, `dbListConnections()`

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

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

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

Examples

```
dbIsValid(RSQLite::SQLite())

con <- dbConnect(RSQLite::SQLite(), ":memory:")
dbIsValid(con)

rs <- dbSendQuery(con, "SELECT 1")
dbIsValid(rs)

dbClearResult(rs)
dbIsValid(rs)

dbDisconnect(con)
dbIsValid(con)
```

---

dbListFields *List field names of a remote table*

Description

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

Usage

```
dbListFields(conn, name, ...)
```

Arguments

conn	A DBI::DBIConnection object, as returned by dbConnect().
name	The table name, passed on to dbQuoteIdentifier(). Options are: • a character string with the unquoted DBMS table name, e.g. "table_name", • a call to Id() with components to the fully qualified table name, e.g. Id(schema = "my_schema", table = "table_name") • a call to SQL() with the quoted and fully qualified table name given verbatim, e.g. SQL('"my_schema"."table_name"')
...	Other parameters passed on to methods.

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

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

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars)
dbListFields(con, "mtcars")

dbDisconnect(con)
```

---

dbListObjects *List remote objects*

Description

Returns the names of remote objects accessible through this connection as a data frame. This should include temporary objects, but not all database backends (in particular **RMariaDB** and **RMySQL**) support this. Compared to `dbListTables()`, this method also enumerates tables and views in schemas, and returns fully qualified identifiers to access these objects. This allows exploration of all database objects available to the current user, including those that can only be accessed by giving the full namespace.

Usage

```
dbListObjects(conn, prefix = NULL, ...)
```

Arguments

conn	A DBI::DBIConnection object, as returned by dbConnect().
prefix	A fully qualified path in the database's namespace, or NULL. This argument will be processed with dbUnquoteIdentifier(). If given the method will return all objects accessible through this prefix.
...	Other parameters passed on to methods.

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbListObjects(con)
dbWriteTable(con, "mtcars", mtcars)
dbListObjects(con)

dbDisconnect(con)
```

`dbListTables` *List remote tables*

Description

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

Usage

```
dbListTables(conn, ...)
```

Arguments

conn	A DBI::DBIConnection object, as returned by `dbConnect()`.
...	Other parameters passed on to methods.

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbListTables(con)
dbWriteTable(con, "mtcars", mtcars)
dbListTables(con)

dbDisconnect(con)
```

`dbQuoteIdentifier` *Quote identifiers*

Description

Call this method to generate a string that is suitable for use in a query as a column or table name, to make sure that you generate valid SQL and protect against SQL injection attacks. The inverse operation is `dbUnquoteIdentifier()`.

Usage

```
dbQuoteIdentifier(conn, x, ...)
```

Arguments

`conn`	A DBI::DBIConnection object, as returned by `dbConnect()`.
`x`	A character vector, SQL or Id object to quote as identifier.
`...`	Other arguments passed on to methods.

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

Examples

```
# Quoting ensures that arbitrary input is safe for use in a query
name <- "Robert'); DROP TABLE Students;--"
dbQuoteIdentifier(ANSI(), name)

# Use Id() to specify other components such as the schema
id_name <- Id(schema = "schema_name", table = "table_name")
id_name
dbQuoteIdentifier(ANSI(), id_name)

# SQL vectors are always passed through as is
var_name <- SQL("select")
var_name
dbQuoteIdentifier(ANSI(), var_name)

# This mechanism is used to prevent double escaping
dbQuoteIdentifier(ANSI(), dbQuoteIdentifier(ANSI(), name))
```

---

dbQuoteLiteral *Quote literal values*

Description

Call these methods to generate a string that is suitable for use in a query as a literal value of the correct type, to make sure that you generate valid SQL and protect against SQL injection attacks.

Usage

```
dbQuoteLiteral(conn, x, ...)
```

Arguments

conn	A DBI::DBIConnection object, as returned by dbConnect().
x	A vector to quote as string.
...	Other arguments passed on to methods.

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.)

Failure modes

Passing a list for the x argument raises an error.

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.

See Also

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

Examples

```
# Quoting ensures that arbitrary input is safe for use in a query
name <- "Robert'); DROP TABLE Students;--"
dbQuoteLiteral(ANSI(), name)

# NAs become NULL
dbQuoteLiteral(ANSI(), c(1:3, NA))

# Logicals become integers by default
dbQuoteLiteral(ANSI(), c(TRUE, FALSE, NA))
```

```
# Raw vectors become hex strings by default
dbQuoteLiteral(ANSI(), list(as.raw(1:3), NULL))

# SQL vectors are always passed through as is
var_name <- SQL("select")
var_name
dbQuoteLiteral(ANSI(), var_name)

# This mechanism is used to prevent double escaping
dbQuoteLiteral(ANSI(), dbQuoteLiteral(ANSI(), name))
```

---

dbQuoteString *Quote literal strings*

Description

Call this method to generate a string that is suitable for use in a query as a string literal, to make sure that you generate valid SQL and protect against SQL injection attacks.

Usage

```
dbQuoteString(conn, x, ...)
```

Arguments

conn	A DBI::DBIConnection object, as returned by dbConnect().
x	A character vector to quote as string.
...	Other arguments passed on to methods.

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.)

Failure modes

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

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.

See Also

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

Examples

```
# Quoting ensures that arbitrary input is safe for use in a query
name <- "Robert'); DROP TABLE Students;--"
dbQuoteString(ANSI(), name)

# NAs become NULL
dbQuoteString(ANSI(), c("x", NA))

# SQL vectors are always passed through as is
var_name <- SQL("select")
var_name
dbQuoteString(ANSI(), var_name)

# This mechanism is used to prevent double escaping
dbQuoteString(ANSI(), dbQuoteString(ANSI(), name))
```

---

dbReadTable *Read database tables as data frames*

Description

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

Usage

```
dbReadTable(conn, name, ...)
```

Arguments

conn	A DBI::DBIConnection object, as returned by dbConnect().
name	The table name, passed on to dbQuoteIdentifier(). Options are: • a character string with the unquoted DBMS table name, e.g. "table_name", • a call to Id() with components to the fully qualified table name, e.g. Id(schema = "my_schema", table = "table_name") • a call to SQL() with the quoted and fully qualified table name given verbatim, e.g. SQL('"my_schema"."table_name"')
...	Other parameters passed on to methods.

Details

This function returns a data frame. Use dbReadTableArrow() to obtain an Arrow object.

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars[1:10, ])
dbReadTable(con, "mtcars")

dbDisconnect(con)
```

---

dbReadTableArrow *Read database tables as Arrow objects*

Description

[Experimental]

Reads a database table as an Arrow object. Use dbReadTable() instead to obtain a data frame.

Usage

```
dbReadTableArrow(conn, name, ...)
```

Arguments

conn	A DBI::DBIConnection object, as returned by dbConnect().
name	The table name, passed on to dbQuoteIdentifier(). Options are: • a character string with the unquoted DBMS table name, e.g. "table_name", • a call to Id() with components to the fully qualified table name, e.g. Id(schema = "my_schema", table = "table_name") • a call to SQL() with the quoted and fully qualified table name given verbatim, e.g. SQL('"my_schema"."table_name"')
...	Other parameters passed on to methods.

Details

This function returns an Arrow object. Convert it to a data frame with as.data.frame() or use dbReadTable() to obtain a data frame.

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

Examples

```
# Read data as Arrow table
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars[1:10, ])
dbReadTableArrow(con, "mtcars")

dbDisconnect(con)
```

---

dbRemoveTable *Remove a table from the database*

Description

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

Usage

```
dbRemoveTable(conn, name, ...)
```

Arguments

conn	A DBI::DBIConnection object, as returned by `dbConnect()`.
name	The table name, passed on to `dbQuoteIdentifier()`. Options are: • a character string with the unquoted DBMS table name, e.g. `"table_name"`, • a call to `Id()` with components to the fully qualified table name, e.g. `Id(schema = "my_schema", table = "table_name")` • a call to `SQL()` with the quoted and fully qualified table name given verbatim, e.g. `SQL('"my_schema"."table_name"')`
...	Other parameters passed on to methods.

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbExistsTable(con, "iris")
dbWriteTable(con, "iris", iris)
dbExistsTable(con, "iris")
dbRemoveTable(con, "iris")
dbExistsTable(con, "iris")

dbDisconnect(con)
```

dbSendQuery *Execute a query on a given database connection*

Description

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

Usage

```
dbSendQuery(conn, statement, ...)
```

Arguments

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

Details

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

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

Value

dbSendQuery() returns an S4 object that inherits from 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().

The data retrieval flow

This section gives a complete overview over the flow for the execution of queries that return tabular data as data frames.

Most of this flow, except repeated calling of dbBind() or dbBindArrow(), is implemented by dbGetQuery(), which should be sufficient unless you want to access the results in a paged way or you have a parameterized query that you want to reuse. This flow requires an active connection established by dbConnect(). See also vignette("dbi-advanced") for a walkthrough.

1. Use dbSendQuery() to create a result set object of class DBIResult.
2. Optionally, bind query parameters with dbBind() or dbBindArrow(). This is required only if the query contains placeholders such as ? or $1, depending on the database backend.
3. Optionally, use dbColumnInfo() to retrieve the structure of the result set without retrieving actual data.
4. Use dbFetch() to get the entire result set, a page of results, or the remaining rows. Fetching zero rows is also possible to retrieve the structure of the result set as a data frame. This step can be called multiple times. Only forward paging is supported, you need to cache previous pages if you need to navigate backwards.
5. Use dbHasCompleted() to tell when you're done. This method returns TRUE if no more rows are available for fetching.
6. Repeat the last four steps as necessary.
7. Use dbClearResult() to clean up the result set object. This step is mandatory even if no rows have been fetched or if an error has occurred during the processing. It is good practice to use on.exit() or withr::defer() to ensure that this step is always executed.

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

For updates: `dbSendStatement()` and `dbExecute()`.

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

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars)
rs <- dbSendQuery(con, "SELECT * FROM mtcars WHERE cyl = 4")
dbFetch(rs)
dbClearResult(rs)

# Pass one set of values with the param argument:
rs <- dbSendQuery(
  con,
```

```
  "SELECT * FROM mtcars WHERE cyl = ?",
  params = list(4L)
)
dbFetch(rs)
dbClearResult(rs)

# Pass multiple sets of values with dbBind():
rs <- dbSendQuery(con, "SELECT * FROM mtcars WHERE cyl = ?")
dbBind(rs, list(6L))
dbFetch(rs)
dbBind(rs, list(8L))
dbFetch(rs)
dbClearResult(rs)

dbDisconnect(con)
```

---

dbSendQueryArrow *Execute a query on a given database connection for retrieval via Arrow*

Description

[Experimental]

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

Usage

```
dbSendQueryArrow(conn, statement, ...)
```

Arguments

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

Details

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

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()`.

The data retrieval flow for Arrow streams

This section gives a complete overview over the flow for the execution of queries that return tabular data as an Arrow stream.

Most of this flow, except repeated calling of `dbBindArrow()` or `dbBind()`, is implemented by `dbGetQueryArrow()`, which should be sufficient unless you have a parameterized query that you want to reuse. This flow requires an active connection established by `dbConnect()`. See also `vignette("dbi-advanced")` for a walkthrough.

1. Use `dbSendQueryArrow()` to create a result set object of class DBIResultArrow.
2. Optionally, bind query parameters with `dbBindArrow()` or `dbBind()`. This is required only if the query contains placeholders such as `?` or `$1`, depending on the database backend.
3. Use `dbFetchArrow()` to get a data stream.
4. Repeat the last two steps as necessary.
5. Use `dbClearResult()` to clean up the result set object. This step is mandatory even if no rows have been fetched or if an error has occurred during the processing. It is good practice to use `on.exit()` or `withr::defer()` to ensure that this step is always executed.

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

For updates: `dbSendStatement()` and `dbExecute()`.

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

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

Examples

```
# Retrieve data as arrow table
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars)
rs <- dbSendQueryArrow(con, "SELECT * FROM mtcars WHERE cyl = 4")
dbFetchArrow(rs)
dbClearResult(rs)

dbDisconnect(con)
```

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

Description

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

Usage

```
dbSendStatement(conn, statement, ...)
```

Arguments

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

Details

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

Value

dbSendStatement() returns an S4 object that inherits from 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().

The command execution flow

This section gives a complete overview over the flow for the execution of SQL statements that have side effects such as stored procedures, inserting or deleting data, or setting database or connection options. Most of this flow, except repeated calling of dbBindArrow(), is implemented by dbExecute(), which should be sufficient for non-parameterized queries. This flow requires an active connection established by dbConnect(). See also vignette("dbi-advanced") for a walk-through.

1. Use dbSendStatement() to create a result set object of class DBIResult. For some queries you need to pass immediate = TRUE.
2. Optionally, bind query parameters withdbBind() or dbBindArrow(). This is required only if the query contains placeholders such as ? or $1, depending on the database backend.
3. Optionally, use dbGetRowsAffected() to retrieve the number of rows affected by the query.

4. Repeat the last two steps as necessary.
5. Use dbClearResult() to clean up the result set object. This step is mandatory even if no rows have been fetched or if an error has occurred during the processing. It is good practice to use on.exit() or withr::defer() to ensure that this step is always executed.

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.

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

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

For queries: `dbSendQuery()` and `dbGetQuery()`.

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

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "cars", head(cars, 3))

rs <- dbSendStatement(
  con,
  "INSERT INTO cars (speed, dist) VALUES (1, 1), (2, 2), (3, 3)"
)
dbHasCompleted(rs)
dbGetRowsAffected(rs)
dbClearResult(rs)
dbReadTable(con, "cars")   # there are now 6 rows

# Pass one set of values directly using the param argument:
rs <- dbSendStatement(
  con,
  "INSERT INTO cars (speed, dist) VALUES (?, ?)",
  params = list(4L, 5L)
)
dbClearResult(rs)

# Pass multiple sets of values using dbBind():
rs <- dbSendStatement(
  con,
  "INSERT INTO cars (speed, dist) VALUES (?, ?)"
)
dbBind(rs, list(5:6, 6:7))
dbBind(rs, list(7L, 8L))
dbClearResult(rs)
dbReadTable(con, "cars")   # there are now 10 rows
```

```
dbDisconnect(con)
```

`dbUnquoteIdentifier` *Unquote identifiers*

Description

Call this method to convert a SQL object created by `dbQuoteIdentifier()` back to a list of Id objects.

Usage

```
dbUnquoteIdentifier(conn, x, ...)
```

Arguments

`conn`	A DBI::DBIConnection object, as returned by `dbConnect()`.
`x`	An SQL or Id object.
`...`	Other arguments passed on to methods.

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

Examples

```
# Unquoting allows to understand the structure of a
# possibly complex quoted identifier
dbUnquoteIdentifier(
  ANSI(),
  SQL(c('"Catalog"."Schema"."Table"', '"Schema"."Table"', '"UnqualifiedTable"'))
)

# The returned object is always a list,
# also for Id objects
dbUnquoteIdentifier(ANSI(), Id("Catalog", "Schema", "Table"))

# Quoting and unquoting are inverses
dbQuoteIdentifier(
  ANSI(),
  dbUnquoteIdentifier(ANSI(), SQL("UnqualifiedTable"))[[1]]
)

dbQuoteIdentifier(
  ANSI(),
  dbUnquoteIdentifier(ANSI(), Id("Schema", "Table"))[[1]]
)
```

---

dbWithTransaction *Self-contained SQL transactions*

Description

Given that transactions are implemented, this function allows you to pass in code that is run in a transaction. The default method of dbWithTransaction() calls dbBegin() before executing the code, and dbCommit() after successful completion, or dbRollback() in case of an error. The advantage is that you don't have to remember to do dbBegin() and dbCommit() or dbRollback() – that is all taken care of. The special function dbBreak() allows an early exit with rollback, it can be called only inside dbWithTransaction().

Usage

```
dbWithTransaction(conn, code, ...)

dbBreak()
```

Arguments

conn	A DBI::DBIConnection object, as returned by dbConnect().
code	An arbitrary block of R code.
...	Other parameters passed on to methods.

Details

DBI implements dbWithTransaction(), backends should need to override this generic only if they implement specialized handling.

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 or 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.

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "cash", data.frame(amount = 100))
dbWriteTable(con, "account", data.frame(amount = 2000))

# All operations are carried out as logical unit:
dbWithTransaction(
  con,
  {
    withdrawal <- 300
    dbExecute(con, "UPDATE cash SET amount = amount + ?", list(withdrawal))
    dbExecute(con, "UPDATE account SET amount = amount - ?", list(withdrawal))
  }
)

# The code is executed as if in the current environment:
withdrawal

# The changes are committed to the database after successful execution:
dbReadTable(con, "cash")
dbReadTable(con, "account")
```

```
# Rolling back with dbBreak():
dbWithTransaction(
  con,
  {
    withdrawal <- 5000
    dbExecute(con, "UPDATE cash SET amount = amount + ?", list(withdrawal))
    dbExecute(con, "UPDATE account SET amount = amount - ?", list(withdrawal))
    if (dbReadTable(con, "account")$amount < 0) {
      dbBreak()
    }
  }
)

# These changes were not committed to the database:
dbReadTable(con, "cash")
dbReadTable(con, "account")

dbDisconnect(con)
```

`dbWriteTable` *Copy data frames to database tables*

Description

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

Usage

```
dbWriteTable(conn, name, value, ...)
```

Arguments

`conn`	A DBI::DBIConnection object, as returned by `dbConnect()`.
`name`	The table name, passed on to `dbQuoteIdentifier()`. Options are: • a character string with the unquoted DBMS table name, e.g. `"table_name"`, • a call to `Id()` with components to the fully qualified table name, e.g. `Id(schema = "my_schema", table = "table_name")` • a call to `SQL()` with the quoted and fully qualified table name given verbatim, e.g. `SQL('"my_schema"."table_name"')`
`value`	A data.frame (or coercible to data.frame).
`...`	Other parameters passed on to methods.

Details

This function expects a data frame. Use `dbWriteTableArrow()` to write an Arrow object.

This function is useful if you want to create and load a table at the same time. Use `dbAppendTable()` or `dbAppendTableArrow()` for appending data to an existing table, `dbCreateTable()` or `dbCreateTableArrow()` for creating a table, and `dbExistsTable()` and `dbRemoveTable()` for overwriting tables.

DBI only standardizes writing data frames with `dbWriteTable()`. Some backends might implement methods that can consume CSV files or other data formats. For details, see the documentation for the individual methods.

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars[1:5, ])
dbReadTable(con, "mtcars")

dbWriteTable(con, "mtcars", mtcars[6:10, ], append = TRUE)
dbReadTable(con, "mtcars")

dbWriteTable(con, "mtcars", mtcars[1:10, ], overwrite = TRUE)
dbReadTable(con, "mtcars")

# No row names
dbWriteTable(con, "mtcars", mtcars[1:10, ], overwrite = TRUE, row.names = FALSE)
dbReadTable(con, "mtcars")
```

---

dbWriteTableArrow *Copy Arrow objects to database tables*

Description

[Experimental]

Writes, overwrites or appends an Arrow object to a database table.

Usage

```
dbWriteTableArrow(conn, name, value, ...)
```

Arguments

conn	A DBI::DBIConnection object, as returned by dbConnect().
name	The table name, passed on to dbQuoteIdentifier(). Options are: • a character string with the unquoted DBMS table name, e.g. "table_name", • a call to Id() with components to the fully qualified table name, e.g. Id(schema = "my_schema", table = "table_name") • a call to SQL() with the quoted and fully qualified table name given verbatim, e.g. SQL('"my_schema"."table_name"')
value	An nanoarray stream, or an object coercible to a nanoarray stream with nanoarrow::as_nanoarrow_arra
...	Other parameters passed on to methods.

Details

This function expects an Arrow object. Convert a data frame to an Arrow object with nanoarrow::as_nanoarrow_array_st or use dbWriteTable() to write a data frame.

This function is useful if you want to create and load a table at the same time. Use dbAppendTableArrow() for appending data to an existing table, dbCreateTableArrow() for creating a table and specifying field types, and dbRemoveTable() for overwriting tables.

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

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTableArrow(con, "mtcars", nanoarrow::as_nanoarrow_array_stream(mtcars[1:5, ]))
dbReadTable(con, "mtcars")

dbDisconnect(con)
```

Id-class *Refer to a table nested in a hierarchy (e.g. within a schema)*

Description

Objects of class Id have a single slot name, which is a character vector. The dbQuoteIdentifier() method converts Id objects to strings. Support for Id objects depends on the database backend.

They can be used in the following methods as name or table argument:

- dbCreateTable()
- dbAppendTable()
- dbReadTable()
- dbWriteTable()
- dbExistsTable()
- dbRemoveTable()

Objects of this class are also returned from dbListObjects().

Usage

```
Id(...)
```

Arguments

...	Components of the hierarchy, e.g. cluster, catalog, schema, or table, depending on the database backend. For more on these concepts, see https://stackoverflow.com/questions/7022755/

Examples

```
# Identifies a table in a specific schema:
Id("dbo", "Customer")
# You can name the components if you want, but it's not needed
Id(table = "Customer", schema = "dbo")

# Create a SQL expression for an identifier:
dbQuoteIdentifier(ANSI(), Id("nycflights13", "flights"))

# Write a table in a specific schema:
## Not run:
dbWriteTable(con, Id("myschema", "mytable"), data.frame(a = 1))

## End(Not run)
```

`rownames` *Convert row names back and forth between columns*

Description

These functions provide a reasonably automatic way of preserving the row names of data frame during back-and-forth translation to an SQL table. By default, row names will be converted to an explicit column called "row_names", and any query returning a column called "row_names" will have those automatically set as row names. These methods are mostly useful for backend implementers.

Usage

```
sqlRownamesToColumn(df, row.names = NA)

sqlColumnToRownames(df, row.names = NA)
```

Arguments

`df`	A data frame
`row.names`	Either `TRUE`, `FALSE`, `NA` or a string. If `TRUE`, always translate row names to a column called "row_names". If `FALSE`, never translate row names. If `NA`, translate rownames only if they're a character vector. A string is equivalent to `TRUE`, but allows you to override the default name. For backward compatibility, `NULL` is equivalent to `FALSE`.

Examples

```
# If have row names
sqlRownamesToColumn(head(mtcars))
sqlRownamesToColumn(head(mtcars), FALSE)
```

```
sqlRownamesToColumn(head(mtcars), "ROWNAMES")

# If don't have
sqlRownamesToColumn(head(iris))
sqlRownamesToColumn(head(iris), TRUE)
sqlRownamesToColumn(head(iris), "ROWNAMES")
```

SQL *SQL quoting*

Description

This set of classes and generics make it possible to flexibly deal with SQL escaping needs. By default, any user supplied input to a query should be escaped using either `dbQuoteIdentifier()` or `dbQuoteString()` depending on whether it refers to a table or variable name, or is a literal string. These functions may return an object of the `SQL` class, which tells DBI functions that a character string does not need to be escaped anymore, to prevent double escaping. The `SQL` class has associated the `SQL()` constructor function.

Usage

```
SQL(x, ..., names = NULL)
```

Arguments

`x`	A character vector to label as being escaped SQL.
`...`	Other arguments passed on to methods. Not otherwise used.
`names`	Names for the returned object, must have the same length as `x`.

Value

An object of class `SQL`.

Implementation notes

DBI provides default generics for SQL-92 compatible quoting. If the database uses a different convention, you will need to provide your own methods. Note that because of the way that S4 dispatch finds methods and because SQL inherits from character, if you implement (e.g.) a method for `dbQuoteString(MyConnection, character)`, you will also need to implement `dbQuoteString(MyConnection, SQL)` - this should simply return `x` unchanged.

Examples

```
dbQuoteIdentifier(ANSI(), "SELECT")
dbQuoteString(ANSI(), "SELECT")

# SQL vectors are always passed through as is
var_name <- SQL("SELECT")
```

```
var_name

dbQuoteIdentifier(ANSI(), var_name)
dbQuoteString(ANSI(), var_name)

# This mechanism is used to prevent double escaping
dbQuoteString(ANSI(), dbQuoteString(ANSI(), "SELECT"))
```

`sqlAppendTable` *Compose query to insert rows into a table*

Description

`sqlAppendTable()` generates a single SQL string that inserts a data frame into an existing table. `sqlAppendTableTemplate()` generates a template suitable for use with `dbBind()`. The default methods are ANSI SQL 99 compliant. These methods are mostly useful for backend implementers.

Usage

```
sqlAppendTable(con, table, values, row.names = NA, ...)

sqlAppendTableTemplate(
  con,
  table,
  values,
  row.names = NA,
  prefix = "?",
  ...,
  pattern = ""
)
```

Arguments

`con`	A database connection.
`table`	The table name, passed on to `dbQuoteIdentifier()`. Options are: • a character string with the unquoted DBMS table name, e.g. `"table_name"`, • a call to `Id()` with components to the fully qualified table name, e.g. `Id(schema = "my_schema", table = "table_name")` • a call to `SQL()` with the quoted and fully qualified table name given verbatim, e.g. `SQL('"my_schema"."table_name"')`
`values`	A data frame. Factors will be converted to character vectors. Character vectors will be escaped with `dbQuoteString()`.
`row.names`	Either `TRUE`, `FALSE`, `NA` or a string. If `TRUE`, always translate row names to a column called "row_names". If `FALSE`, never translate row names. If `NA`, translate rownames only if they're a character vector.

	A string is equivalent to TRUE, but allows you to override the default name. For backward compatibility, NULL is equivalent to FALSE.
...	Other arguments used by individual methods.
prefix	Parameter prefix to use for placeholders.
pattern	Parameter pattern to use for placeholders: • "": no pattern • "1": position • anything else: field name

Details

The row.names argument must be passed explicitly in order to avoid a compatibility warning. The default will be changed in a later release.

Examples

```
sqlAppendTable(ANSI(), "iris", head(iris))

sqlAppendTable(ANSI(), "mtcars", head(mtcars))
sqlAppendTable(ANSI(), "mtcars", head(mtcars), row.names = FALSE)
sqlAppendTableTemplate(ANSI(), "iris", iris)

sqlAppendTableTemplate(ANSI(), "mtcars", mtcars)
sqlAppendTableTemplate(ANSI(), "mtcars", mtcars, row.names = FALSE)
```

---

sqlCreateTable *Compose query to create a simple table*

Description

Exposes an interface to simple CREATE TABLE commands. The default method is ANSI SQL 99 compliant. This method is mostly useful for backend implementers.

Usage

```
sqlCreateTable(con, table, fields, row.names = NA, temporary = FALSE, ...)
```

Arguments

con	A database connection.
table	The table name, passed on to dbQuoteIdentifier(). Options are: • a character string with the unquoted DBMS table name, e.g. "table_name", • a call to Id() with components to the fully qualified table name, e.g. Id(schema = "my_schema", table = "table_name") • a call to SQL() with the quoted and fully qualified table name given verbatim, e.g. SQL('"my_schema"."table_name"')

`fields`	Either a character vector or a data frame. A named character vector: Names are column names, values are types. Names are escaped with `dbQuoteIdentifier()`. Field types are unescaped. A data frame: field types are generated using `dbDataType()`.
`row.names`	Either `TRUE`, `FALSE`, `NA` or a string. If `TRUE`, always translate row names to a column called "row_names". If `FALSE`, never translate row names. If `NA`, translate rownames only if they're a character vector. A string is equivalent to `TRUE`, but allows you to override the default name. For backward compatibility, `NULL` is equivalent to `FALSE`.
`temporary`	If `TRUE`, will generate a temporary table.
`...`	Other arguments used by individual methods.

Details

The `row.names` argument must be passed explicitly in order to avoid a compatibility warning. The default will be changed in a later release.

Examples

```
sqlCreateTable(ANSI(), "my-table", c(a = "integer", b = "text"))
sqlCreateTable(ANSI(), "my-table", iris)

# By default, character row names are converted to a row_names colum
sqlCreateTable(ANSI(), "mtcars", mtcars[, 1:5])
sqlCreateTable(ANSI(), "mtcars", mtcars[, 1:5], row.names = FALSE)
```

---

sqlData *Convert a data frame into form suitable for upload to an SQL database*

Description

This is a generic method that coerces R objects into vectors suitable for upload to the database. The output will vary a little from method to method depending on whether the main upload device is through a single SQL string or multiple parameterized queries. This method is mostly useful for backend implementers.

Usage

```
sqlData(con, value, row.names = NA, ...)
```

Arguments

con	A database connection.
value	A data frame
row.names	Either TRUE, FALSE, NA or a string. If TRUE, always translate row names to a column called "row_names". If FALSE, never translate row names. If NA, translate rownames only if they're a character vector. A string is equivalent to TRUE, but allows you to override the default name. For backward compatibility, NULL is equivalent to FALSE.
...	Other arguments used by individual methods.

Details

The default method:

- Converts factors to characters
- Quotes all strings with dbQuoteIdentifier()
- Converts all columns to strings with dbQuoteLiteral()
- Replaces NA with NULL

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

sqlData(con, head(iris))
sqlData(con, head(mtcars))

dbDisconnect(con)
```

---

sqlInterpolate *Safely interpolate values into an SQL string*

Description

Accepts a query string with placeholders for values, and returns a string with the values embedded. The function is careful to quote all of its inputs with dbQuoteLiteral() to protect against SQL injection attacks.

Placeholders can be specified with one of two syntaxes:

- ?: each occurrence of a standalone ? is replaced with a value
- ?name1, ?name2, ...: values are given as named arguments or a named list, the names are used to match the values

Mixing ? and ?name syntaxes is an error. The number and names of values supplied must correspond to the placeholders used in the query.

Usage

```
sqlInterpolate(conn, sql, ..., .dots = list())
```

Arguments

conn	A DBI::DBIConnection object, as returned by dbConnect().
sql	A SQL string containing variables to interpolate. Variables must start with a question mark and can be any valid R identifier, i.e. it must start with a letter or ., and be followed by a letter, digit, . or _.
..., .dots	Values (for ...) or a list (for .dots) to interpolate into a string. Names are required if sql uses the ?name syntax for placeholders. All values will be first escaped with dbQuoteLiteral() prior to interpolation to protect against SQL injection attacks. Arguments created by SQL() or dbQuoteIdentifier() remain unchanged.

Value

The sql query with the values from ... and .dots safely embedded.

Backend authors

If you are implementing an SQL backend with non-ANSI quoting rules, you'll need to implement a method for sqlParseVariables(). Failure to do so does not expose you to SQL injection attacks, but will (rarely) result in errors matching supplied and interpolated variables.

Examples

```
sql <- "SELECT * FROM X WHERE name = ?name"
sqlInterpolate(ANSI(), sql, name = "Hadley")

# This is safe because the single quote has been double escaped
sqlInterpolate(ANSI(), sql, name = "H'); DROP TABLE--;")

# Using paste0() could lead to dangerous SQL with carefully crafted inputs
# (SQL injection)
name <- "H'); DROP TABLE--;"
paste0("SELECT * FROM X WHERE name = '", name, "'")

# Use SQL() or dbQuoteIdentifier() to avoid escaping
sql2 <- "SELECT * FROM ?table WHERE name in ?names"
sqlInterpolate(ANSI(), sql2,
  table = dbQuoteIdentifier(ANSI(), "X"),
  names = SQL("('a', 'b')")
)

# Don't use SQL() to escape identifiers to avoid SQL injection
sqlInterpolate(ANSI(), sql2,
  table = SQL("X; DELETE FROM X; SELECT * FROM X"),
  names = SQL("('a', 'b')")
)
```

```
# Use dbGetQuery() or dbExecute() to process these queries:
if (requireNamespace("RSQLite", quietly = TRUE)) {
  con <- dbConnect(RSQLite::SQLite())
  sql <- "SELECT ?value AS value"
  query <- sqlInterpolate(con, sql, value = 3)
  print(dbGetQuery(con, query))
  dbDisconnect(con)
}
```

Index

∗ **DBI classes**
DBIConnection-class, 52
DBIConnector-class, 53
DBIDriver-class, 53
DBIObject-class, 54
DBIResult-class, 55
DBIResultArrow-class, 55
∗ **DBIConnection generics**
dbAppendTable, 5
dbAppendTableArrow, 7
dbCreateTable, 22
dbCreateTableArrow, 24
dbDataType, 26
dbDisconnect, 28
dbExecute, 29
dbExistsTable, 32
dbGetInfo, 40
dbGetQuery, 41
dbGetQueryArrow, 44
DBIConnection-class, 52
dbIsReadOnly, 56
dbIsValid, 57
dbListFields, 58
dbListObjects, 59
dbListTables, 61
dbQuoteIdentifier, 62
dbReadTable, 66
dbReadTableArrow, 68
dbRemoveTable, 70
dbSendQuery, 72
dbSendQueryArrow, 75
dbSendStatement, 78
dbUnquoteIdentifier, 81
dbWriteTable, 84
dbWriteTableArrow, 87
∗ **DBIConnector generics**
dbConnect, 21
dbDataType, 26
dbGetConnectArgs, 39
DBIConnector-class, 53
dbIsReadOnly, 56
∗ **DBIDriver generics**
dbCanConnect, 16
dbConnect, 21
dbDataType, 26
dbGetInfo, 40
DBIDriver-class, 53
dbIsReadOnly, 56
dbIsValid, 57
∗ **DBIResult generics**
dbBind, 11
dbClearResult, 17
dbColumnInfo, 19
dbFetch, 33
dbGetInfo, 40
dbGetRowCount, 46
dbGetRowsAffected, 47
dbGetStatement, 49
dbHasCompleted, 50
DBIResult-class, 55
dbIsReadOnly, 56
dbIsValid, 57
dbQuoteLiteral, 63
dbQuoteString, 65
∗ **DBIResultArrow generics**
dbBind, 11
dbClearResult, 17
dbFetchArrow, 36
dbFetchArrowChunk, 38
dbHasCompleted, 50
DBIResultArrow-class, 55
dbIsValid, 57
∗ **SQL generation**
sqlAppendTable, 93
∗ **command execution generics**
dbBind, 11
dbClearResult, 17
dbExecute, 29

dbGetRowsAffected, 47
dbSendStatement, 78
∗ **data retrieval generics**
dbBind, 11
dbClearResult, 17
dbFetch, 33
dbFetchArrow, 36
dbFetchArrowChunk, 38
dbGetQuery, 41
dbGetQueryArrow, 44
dbHasCompleted, 50
dbSendQuery, 72
dbSendQueryArrow, 75
∗ **datasets**
.SQL92Keywords, 4
.SQL92Keywords, 4

as.character(), *22*
as.data.frame(), *69*
as.Date(), *35*
as.integer(), *22*
as.numeric(), *22*
as.POSIXct(), *35*

blob::blob, *6*, *8*, *14*, *27*, *86*, *89*

character, *14*, *27*, *35*, *62*, *64, 65*

data.frame, *5*, *14*, *34*, *36*, *38*, *42*, *45*, *84*
Date, *14*, *35*
Dates, *27*
DateTimeClasses, *27*
dbAppendTable, 5, *9*, *24*, *26, 27*, *29*, *31*, *33*, *41*, *43*, *46*, *52*, *56, 57*, *59–61*, *63*, *68, 69*, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbAppendTable(), *7*, *85*, *90*
dbAppendTableArrow, *7*, 7, *24*, *26, 27*, *29*, *31*, *33*, *41*, *43*, *46*, *52*, *56, 57*, *59–61*, *63*, *68, 69*, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbAppendTableArrow(), *5*, *24*, *85*, *88*
dbBegin, 9
dbBegin(), *82*
dbBind, 11, *18*, *20*, *31*, *35*, *37*, *39*, *41*, *43*, *46–49*, *51*, *55–57*, *64*, *66*, *74*, *77*, *80*
dbBind(), *12, 13*, *17–20*, *29*, *34*, *37, 38*, *48*, *50*, *72, 73*, *76*, *78*, *93*
dbBindArrow *(*dbBind*)*, 11
dbBindArrow(), *12, 13*, *17–20*, *34*, *37, 38*, *48*, *50*, *72, 73*, *76*, *78*
dbBreak *(*dbWithTransaction*)*, 82
dbCanConnect, 16, *22*, *27*, *41*, *54*, *56, 57*
dbCanConnect(), *21*
dbClearResult, *15*, 17, *20*, *31*, *35*, *37*, *39*, *41*, *43*, *46–49*, *51*, *55–57*, *64*, *66*, *74*, *77*, *80*
dbClearResult(), *12, 13*, *18*, *20*, *29*, *34, 35*, *37–39*, *41*, *44*, *48*, *51*, *72, 73*, *75, 76*, *78, 79*
dbColumnInfo, *15*, *18*, 19, *35*, *41*, *47–49*, *51*, *55–57*, *64*, *66*
dbColumnInfo(), *12*, *17*, *20*, *34*, *51*, *59*, *73*
dbCommit *(*dbBegin*)*, 9
dbCommit(), *82*
dbConnect, *16*, 21, *27, 28*, *40, 41*, *53, 54*, *56, 57*
dbConnect(), *4, 5*, *7*, *9*, *12, 13*, *16–19*, *22*, *25*, *28, 29*, *32*, *34*, *37–39*, *42*, *44*, *48*, *50*, *53*, *58*, *60–62*, *64, 65*, *67*, *69, 70*, *72*, *75, 76*, *78*, *81*, *83, 84*, *88*, *97*
dbCreateTable, *7*, *9*, 22, *26, 27*, *29*, *31*, *33*, *41*, *43*, *46*, *52*, *56, 57*, *59–61*, *63*, *68, 69*, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbCreateTable(), *5*, *24*, *85*, *90*
dbCreateTableArrow, *7*, *9*, *24*, 24, *27*, *29*, *31*, *33*, *41*, *43*, *46*, *52*, *56, 57*, *59–61*, *63*, *68, 69*, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbCreateTableArrow(), *7*, *22*, *85*, *88*
dbDataType, *7*, *9*, *16*, *22*, *24*, *26*, 26, *29*, *31*, *33*, *40, 41*, *43*, *46*, *52–54*, *56, 57*, *59–61*, *63*, *68, 69*, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbDataType(), *23*, *95*
dbDisconnect, *7*, *9*, *24*, *26, 27*, 28, *31*, *33*, *41*, *43*, *46*, *52*, *56, 57*, *59–61*, *63*, *68, 69*, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbDisconnect(), *4*, *22*
dbDriver, *16*, *22*, *27*, *41*, *54*, *56, 57*
dbExecute, *7*, *9*, *15*, *18*, *24*, *26, 27*, *29*, 29, *33*, *41*, *43*, *46*, *48*, *52*, *56, 57*, *59–61*, *63*, *68, 69*, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbExecute(), *5*, *13*, *18*, *22*, *42–44*, *46*, *48*, *74*, *77, 78*
dbExistsTable, *7*, *9*, *24*, *26, 27*, *29*, *31*, 32, *41*, *43*, *46*, *52*, *56, 57*, *59–61*, *63*, *68, 69*, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbExistsTable(), *85*, *90*
dbFetch, *15*, *18*, *20*, 33, *37*, *39*, *41*, *43*, *46–49*,

51, *55–57*, *64*, *66*, *74*, *77*
dbFetch(), *11, 12*, *17*, *20*, *34*, *41*, *46*, *51*, *72, 73*
dbFetchArrow, *15*, *18*, *35*, 36, *39*, *43*, *46*, *51*, *56, 57*, *74*, *77*
dbFetchArrow(), *11*, *13*, *37, 38*, *44*, *75, 76*
dbFetchArrowChunk, *15*, *18*, *35*, *37*, 38, *43*, *46*, *51*, *56, 57*, *74*, *77*
dbFetchArrowChunk(), *36*
dbGetConnectArgs, *22*, *28*, 39, *53*, *56*
dbGetConnectArgs(), *53*
dbGetException, *7*, *9*, *24*, *26, 27*, *29*, *31*, *33*, *41*, *43*, *46*, *52*, *56, 57*, *59–61*, *63*, *68, 69*, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbGetInfo, *7*, *9*, *15, 16*, *18*, *20*, *22*, *24*, *26, 27*, *29*, *31*, *33*, *35*, 40, *43*, *46–49*, *51, 52*, *54–57*, *59–61*, *63, 64*, *66*, *68, 69*, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbGetInfo(), *54*
dbGetQuery, *7*, *9*, *15*, *18*, *24*, *26, 27*, *29*, *31*, *33*, *35*, *37*, *39*, *41*, 41, *46*, *51, 52*, *56, 57*, *59–61*, *63*, *68, 69*, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbGetQuery(), *4*, *12*, *17*, *19*, *30, 31*, *34*, *44*, *50*, *72*, *75*, *80*
dbGetQueryArrow, *7*, *9*, *15*, *18*, *24*, *26, 27*, *29*, *31*, *33*, *35*, *37*, *39*, *41*, *43*, 44, *51, 52*, *56, 57*, *59–61*, *63*, *68, 69*, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbGetQueryArrow(), *12*, *37, 38*, *41*, *72*, *75, 76*
dbGetRowCount, *15*, *18*, *20*, *35*, *41*, 46, *48, 49*, *51*, *55–57*, *64*, *66*
dbGetRowCount(), *41*
dbGetRowsAffected, *15*, *18*, *20*, *31*, *35*, *41*, *47*, 47, *49*, *51*, *55–57*, *64*, *66*, *80*
dbGetRowsAffected(), *13*, *18*, *29*, *41*, *48*, *78*
dbGetStatement, *15*, *18*, *20*, *35*, *41*, *47, 48*, 49, *51*, *55–57*, *64*, *66*
dbGetStatement(), *41*
dbHasCompleted, *15*, *18*, *20*, *35*, *37*, *39*, *41*, *43*, *46–49*, 50, *55–57*, *64*, *66*, *74*, *77*
dbHasCompleted(), *12*, *18*, *20*, *34*, *41*, *51*, *73*
DBI *(*DBI-package*)*, 3
DBI-package, 3
DBI::dbBegin(), *83*
DBI::dbBind(), *14*, *30, 31*, *43*, *45, 46*, *73, 74*, *76, 77*, *79, 80*
DBI::dbBindArrow(), *14*
DBI::dbClearResult(), *13, 14*, *47–49*, *51*, *57*, *72, 73*, *76*, *78, 79*
DBI::dbCommit(), *83*
DBI::dbDataType(), *27*, *86*
DBI::dbDisconnect(), *57*
DBI::dbExecute(), *41*
DBI::dbExistsTable(), *60*, *71*
DBI::dbFetch(), *14*, *19*, *47, 48*, *50*, *72*
DBI::dbFetchArrow(), *76*
DBI::dbGetQuery(), *63*, *67*
DBI::dbGetQueryArrow(), *69*
DBI::dbGetRowCount(), *14*, *41*
DBI::dbGetRowsAffected(), *14*, *41*, *78*
DBI::dbGetStatement(), *41*
DBI::dbHasCompleted(), *14*, *41*
DBI::DBIConnection, *3*, *5*, *7*, *9*, *16*, *21, 22*, *25*, *27–29*, *32*, *40*, *42*, *44*, *57, 58*, *60–62*, *64, 65*, *67*, *69, 70*, *72*, *75*, *78*, *81*, *83, 84*, *88*, *97*
DBI::DBIDriver, *3, 4*, *16*, *21*, *27*, *40*
DBI::DBIResult, *3*, *11*, *13*, *17*, *19*, *33*, *41*, *46*, *48–50*, *57*, *72*, *78*
DBI::DBIResultArrow, *36*, *38*, *76*
DBI::dbIsValid(), *14*
DBI::dbListObjects(), *59*
DBI::dbListTables(), *32*, *60*, *71*
DBI::dbQuoteIdentifier(), *6*, *8*, *23–25*, *32*, *59, 60*, *67–71*, *81*, *85*, *88, 89*
DBI::dbReadTable(), *6*, *8*, *86*, *89*
DBI::dbRollback(), *83*
DBI::dbSendQuery(), *12, 13*, *18*, *41*, *47–50*, *57*
DBI::dbSendQueryArrow(), *12, 13*
DBI::dbSendStatement(), *12, 13*, *18*, *34*, *47–50*, *57*
DBI::dbUnquoteIdentifier(), *60*
DBI::dbWriteTable(), *60, 61*
DBI::Id, *60*, *81*
DBI::SQL, *62*, *64, 65*
DBI::SQL(), *81*
DBI::sqlColumnToRownames(), *67*
DBI::sqlRownamesToColumn(), *86*
DBIConnection, *9*, *40*, *56, 57*
DBIConnection-class, 52
DBIConnector, *39*
DBIConnector-class, 53
DBIDriver, *40*, *53*, *56, 57*
DBIDriver-class, 53

DBIObject, *40*, *56, 57*
DBIObject-class, 54
DBIResult, *12, 13*, *17–19*, *34*, *40*, *48*, *50*, *56, 57*, *73*, *78*
DBIResult-class, 55
DBIResultArrow, *13*, *37, 38*, *76*
DBIResultArrow-class, 55
DBIResultArrowDefault-class *(*DBIResultArrow-class*)*, 55
dbIsReadOnly, *7*, *9*, *15, 16*, *18*, *20*, *22*, *24*, *26–29*, *31*, *33*, *35*, *40, 41*, *43*, *46–49*, *51–55*, 56, *57*, *59–61*, *63, 64*, *66*, *68, 69*, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbIsValid, *7*, *9*, *15, 16*, *18*, *20*, *22*, *24*, *26, 27*, *29*, *31*, *33*, *35*, *37*, *39*, *41*, *43*, *46–49*, *51, 52*, *54–56*, 57, *59–61*, *63, 64*, *66*, *68, 69*, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbListConnections, *16*, *22*, *27*, *41*, *54*, *56, 57*
dbListFields, *7*, *9*, *24*, *26, 27*, *29*, *31*, *33*, *41*, *43*, *46*, *52*, *56, 57*, 58, *60, 61*, *63*, *68, 69*, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbListObjects, *7*, *9*, *24*, *26, 27*, *29*, *31*, *33*, *41*, *43*, *46*, *52*, *56, 57*, *59*, 59, *61*, *63*, *68, 69*, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbListObjects(), *90*
dbListResults, *7*, *9*, *24*, *26, 27*, *29*, *31*, *33*, *41*, *43*, *46*, *52*, *56, 57*, *59–61*, *63*, *68, 69*, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbListTables, *7*, *9*, *24*, *26, 27*, *29*, *31*, *33*, *41*, *43*, *46*, *52*, *56, 57*, *59, 60*, 61, *63*, *68, 69*, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbListTables(), *59*
dbQuoteIdentifier, *7*, *9*, *24*, *26, 27*, *29*, *31*, *33*, *41*, *43*, *46*, *52*, *56, 57*, *59–61*, 62, *68, 69*, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbQuoteIdentifier(), *5*, *7*, *22, 23*, *25*, *32*, *58*, *67*, *69, 70*, *81*, *84*, *88*, *90*, *92–97*
dbQuoteLiteral, *15*, *18*, *20*, *35*, *41*, *47–49*, *51*, *55–57*, 63, *66*
dbQuoteLiteral(), *96, 97*
dbQuoteString, *15*, *18*, *20*, *35*, *41*, *47–49*, *51*, *55–57*, *64*, 65
dbQuoteString(), *92, 93*
dbReadTable, *7*, *9*, *24*, *26*, *28, 29*, *31*, *33*, *41*, *43*, *46*, *52*, *56, 57*, *59–61*, *63*, 66, *69*, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbReadTable(), *4*, *68, 69*, *90*
dbReadTableArrow, *7*, *9*, *24*, *26*, *28, 29*, *31*, *33*, *41*, *43*, *46*, *52*, *56, 57*, *59–61*, *63*, *68*, 68, *71*, *74*, *77*, *80*, *82*, *87*, *90*
dbReadTableArrow(), *66, 67*
dbRemoveTable, *7*, *9*, *24*, *26*, *28, 29*, *31*, *33*, *41*, *43*, *46*, *52*, *56, 57*, *59–61*, *63*, *68, 69*, 70, *74*, *77*, *80*, *82*, *87*, *90*
dbRemoveTable(), *85*, *88*, *90*
dbRollback *(*dbBegin*)*, 9
dbRollback(), *82*
dbSendQuery, *7*, *9*, *15*, *18*, *24*, *26*, *28, 29*, *31*, *33*, *35*, *37*, *39*, *41*, *43*, *46*, *51, 52*, *56, 57*, *59–61*, *63*, *68, 69*, *71*, 72, *77*, *80*, *82*, *87*, *90*
dbSendQuery(), *11, 12*, *17*, *19*, *31*, *33, 34*, *41*, *49, 50*, *73*, *75*, *78*, *80*
dbSendQueryArrow, *7*, *9*, *15*, *18*, *24*, *26*, *28, 29*, *31*, *33*, *35*, *37*, *39*, *41*, *43*, *46*, *51, 52*, *56, 57*, *59–61*, *63*, *68, 69*, *71*, *74*, 75, *80*, *82*, *87*, *90*
dbSendQueryArrow(), *11*, *13*, *36–38*, *44*, *72*, *76*
dbSendStatement, *7*, *9*, *15*, *18*, *24*, *26*, *28, 29*, *31*, *33*, *41*, *43*, *46*, *48*, *52*, *56, 57*, *59–61*, *63*, *68, 69*, *71*, *74*, *77*, 78, *82*, *87*, *90*
dbSendStatement(), *11*, *13*, *17, 18*, *29*, *43*, *46*, *48, 49*, *72*, *74, 75*, *77, 78*
dbUnquoteIdentifier, *7*, *9*, *24*, *26*, *28, 29*, *31*, *33*, *41*, *43*, *46*, *52*, *56, 57*, *59–61*, *63*, *68, 69*, *71*, *74*, *77*, *80*, 81, *87*, *90*
dbUnquoteIdentifier(), *60*, *62*
dbWithTransaction, 82
dbWithTransaction(), *10*
dbWriteTable, *7*, *9*, *24*, *26*, *28, 29*, *31*, *33*, *41*, *43*, *46*, *52*, *56, 57*, *59–61*, *63*, *68, 69*, *71*, *74*, *77*, *80*, *82*, 84, *90*
dbWriteTable(), *4*, *70*, *88*, *90*
dbWriteTableArrow, *7*, *9*, *24*, *26*, *28, 29*, *31*, *33*, *41*, *43*, *46*, *52*, *56, 57*, *59–61*, *63*, *68, 69*, *71*, *74*, *77*, *80*, *82*, *87*, 87
dbWriteTableArrow(), *85*
difftime, *14*, *27*

factor, *14*, *27*
fetch *(*dbFetch*)*, 33
format(), *21*

hms::as_hms(), *35*

I(), 27
Id, 62, 81
Id (Id-class), 90
Id(), 5, 7, 23, 25, 32, 58, 67, 69, 70, 84, 88, 93, 94
Id-class, 90
Inf, 34, 43
integer, 14, 27, 34, 35
is.na(), 64, 66

logical, 14, 27, 35

NA, 14, 35
nanoarrow::as_nanoarrow_array(), 39
nanoarrow::as_nanoarrow_array_stream(), 7, 11, 37, 45, 88
nanoarrow::infer_nanoarrow_schema(), 25
NULL, 35
numeric, 14, 27, 34, 35

on.exit(), 12, 13, 18, 20, 34, 37, 38, 48, 51, 73, 76, 79
ordered, 27

POSIXct, 14, 35
POSIXlt, 14

raw, 14, 27, 35
rbind(), 14
rownames, 67, 86, 91

SQL, 62, 81, 92
SQL(), 5, 7, 23, 25, 32, 58, 67, 69, 70, 84, 88, 93, 94, 97
SQL-class (SQL), 92
sqlAppendTable, 93
sqlAppendTable(), 5
sqlAppendTableTemplate (sqlAppendTable), 93
sqlAppendTableTemplate(), 5
sqlColumnToRownames (rownames), 91
sqlCreateTable, 94
sqlCreateTable(), 22
sqlData, 95
sqlInterpolate, 96
sqlParseVariables(), 97
sqlRownamesToColumn (rownames), 91
sqlRownamesToColumn(), 5, 23
summary(), 54

transactions, 82
transactions (dbBegin), 9

withr::defer(), 12, 13, 18, 20, 34, 37, 38, 48, 51, 73, 76, 79