Package 'DBI'

June 18, 2017

Version 0.7

Date 2017-06-17

Title R Database Interface

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.

Depends R (>= 3.0.0), methods

Suggests blob, covr, hms, knitr, magrittr, rprojroot, rmarkdown, RSQLite (>= 1.1-2), testthat, xml2

Encoding UTF-8

License LGPL (>= 2)

URL http://rstats-db.github.io/DBI

URLNote https://github.com/rstats-db/DBI

BugReports https://github.com/rstats-db/DBI/issues

Collate 'DBObject.R' 'DBDriver.R' 'DBConnection.R' 'ANSI.R' 'DBI-package.R' 'DBResult.R' 'data-types.R' 'data.R' 'deprecated.R' 'hidden.R' 'interpolate.R' 'quote.R' 'rownames.R' 'table-create.R' 'table-insert.R' 'table.R' 'transactions.R' 'util.R'

VignetteBuilder knitr

RoxygenNote 6.0.1

NeedsCompilation no

Author R Special Interest Group on Databases (R-SIG-DB) [aut], Hadley Wickham [aut], Kirill Müller [aut, cre]

Maintainer Kirill Müller <krlmlr+r@mailbox.org>

Repository CRAN

Date/Publication 2017-06-18 15:23:10 UTC

R topics documented:

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 DBIDriver, DBIConnection, and DBIResult. The backend provides implementation for all methods of these base classes that are defined but not implemented by DBI. All methods have an ellipsis `...` in their formals.

Construction of the DBIDriver object

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

Author(s)

Maintainer: Kirill Müller <krlmlr+r@mailbox.org>

Authors:

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

See Also

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

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

Examples

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

dbBind *Bind values to a parameterized/prepared statement*

Description

For parametrized or prepared statements, the dbSendQuery() and dbSendStatement() functions can be called with statements that contain placeholders for values. The dbBind() function binds these placeholders to actual values, and is intended to be called on the result set before calling dbFetch() or dbGetRowsAffected().

Usage

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

Arguments

res	An object inheriting from DBIResult.
params	A list of bindings, named or unnamed.
...	Other arguments passed on to methods.

Details

DBI supports parametrized (or prepared) queries and statements via the dbBind() generic. 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 **RMySQL** 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 dbSendQuery() and also for data manipulation statements issued by dbSendStatement(). Calling dbBind() for a query without parameters raises an error. Binding too many or not enough values, or parameters with wrong names or unequal length, also raises an error. If the placeholders in the query are named, all parameter values must have names (which must not be empty or NA), and vice versa, otherwise an error is

raised. The behavior for mixing placeholders of different types (in particular mixing positional and named placeholders) is not specified.

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

Specification

DBI clients execute parametrized statements as follows:

1. Call dbSendQuery() or dbSendStatement() with a query or statement that contains placeholders, store the returned DBIResult object in a variable. Mixing placeholders (in particular, named and unnamed ones) is not recommended. It is good practice to register a call to dbClearResult() via on.exit() right after calling dbSendQuery() or dbSendStatement() (see the last enumeration item). Until dbBind() has been called, the returned result set object has the following behavior:
   - dbFetch() raises an error (for dbSendQuery())
   - dbGetRowCount() returns zero (for dbSendQuery())
   - dbGetRowsAffected() returns an integer NA (for dbSendStatement())
   - dbIsValid() returns TRUE
   - dbHasCompleted() returns FALSE
2. Construct a list with parameters that specify actual values for the placeholders. The list must be named or unnamed, depending on the kind of placeholders used. Named values are matched to named parameters, unnamed values are matched by position in the list of parameters. All elements in this list must have the same lengths and contain values supported by the backend; a data.frame is internally stored as such a list. The parameter list is passed to a call to dbBind() on the DBIResult object.
3. Retrieve the data or the number of affected rows from the DBIResult object.
   - For queries issued by dbSendQuery(), call dbFetch().
   - For statements issued by dbSendStatements(), call dbGetRowsAffected(). (Execution begins immediately after the dbBind() call, the statement is processed entirely before the function returns.)
4. Repeat 2. and 3. as necessary.
5. Close the result set via dbClearResult().

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

At least the following data types are accepted:

- integer
- numeric
- logical for Boolean values (some backends may return an integer)
- NA

- character
- factor (bound as character, with warning)
- Date
- POSIXct timestamps
- POSIXlt timestamps
- lists of raw for blobs (with NULL entries for SQL NULL values)
- objects of type blob::blob

See Also

Other DBIResult generics: DBIResult-class, dbClearResult, dbColumnInfo, dbFetch, dbGetInfo, dbGetRowCount, dbGetRowsAffected, dbGetStatement, dbHasCompleted, dbIsValid, dbQuoteIdentifier, dbQuoteString

Examples

```
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)
```

---

dbClearResult *Clear a result set*

Description

Frees all resources (local and remote) associated with a result set. In some cases (e.g., very large result sets) this can be a critical step to avoid exhausting resources (memory, file descriptors, etc.)

Usage

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

Arguments

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

Value

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

Specification

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

See Also

Other DBIResult generics: DBIResult-class, dbBind, dbColumnInfo, dbFetch, dbGetInfo, dbGetRowCount, dbGetRowsAffected, dbGetStatement, dbHasCompleted, dbIsValid, dbQuoteIdentifier, dbQuoteString

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 should describe an aspect of the result set field (field name, type, etc.)

Usage

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

Arguments

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

Value

A data.frame with one row per output field in res. Methods MUST include name, field.type (the SQL type), and data.type (the R data type) columns, and MAY contain other database specific information like scale and precision or whether the field can store NULLs.

See Also

Other DBIResult generics: DBIResult-class, dbBind, dbClearResult, dbFetch, dbGetInfo, dbGetRowCount, dbGetRowsAffected, dbGetStatement, dbHasCompleted, dbIsValid, dbQuoteIdentifier, 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.

Usage

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

Arguments

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

Value

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

Specification

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

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

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

See Also

dbDisconnect() to disconnect from a database.

Other DBIDriver generics: DBIDriver-class, dbDataType, dbDriver, dbGetInfo, 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.)
con <- dbConnect(RSQLite::SQLite(), ":memory:")
con

dbListTables(con)

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

Specification

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

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

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`, `dbConnect`, `dbDriver`, `dbGetInfo`, `dbIsValid`, `dbListConnections`

Other DBIConnection generics: `DBIConnection-class`, `dbDisconnect`, `dbExecute`, `dbExistsTable`, `dbGetException`, `dbGetInfo`, `dbGetQuery`, `dbIsValid`, `dbListFields`, `dbListResults`, `dbListTables`, `dbReadTable`, `dbRemoveTable`, `dbSendQuery`, `dbSendStatement`, `dbWriteTable`

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

Value

dbDisconnect() returns TRUE, invisibly.

Specification

A warning is issued on garbage collection when a connection has been released without calling `dbDisconnect()`. A warning is issued immediately when calling `dbDisconnect()` on an already disconnected or invalid connection.

See Also

Other DBIConnection generics: `DBIConnection-class`, `dbDataType`, `dbExecute`, `dbExistsTable`, `dbGetException`, `dbGetInfo`, `dbGetQuery`, `dbIsValid`, `dbListFields`, `dbListResults`, `dbListTables`, `dbReadTable`, `dbRemoveTable`, `dbSendQuery`, `dbSendStatement`, `dbWriteTable`

Examples

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

---

dbDriver *Load and unload database drivers*

Description

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

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

`dbUnloadDriver()` is not implemented for modern backends.

Usage

```
dbDriver(drvName, ...)

dbUnloadDriver(drv, ...)
```

Arguments

`drvName`	character name of the driver to instantiate.
`...`	any other arguments are passed to the driver `drvName`.
`drv`	an object that inherits from `DBIDriver` as created by `dbDriver`.

Details

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

Value

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

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

See Also

Other DBIDriver generics: DBIDriver-class, dbConnect, dbDataType, dbGetInfo, dbIsValid, dbListConnections

Other DBIDriver generics: DBIDriver-class, dbConnect, dbDataType, dbGetInfo, dbIsValid, dbListConnections

Examples

```
# Create a RSQLite driver with a string
d <- dbDriver("SQLite")
d

# But better, access the object directly
RSQLite::SQLite()
```

---

dbExecute *Execute an update statement, query number of rows affected, and then close result set*

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 free-d by dbClearResult().

Usage

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

Arguments

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

Value

dbExecute() always returns a scalar numeric that specifies the number of rows affected by the statement. An error is raised when issuing a statement over a closed or invalid connection, if the syntax of the statement is invalid, or if the statement is not a non-NA string.

Implementation notes

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

See Also

For queries: dbSendQuery() and dbGetQuery().

Other DBIConnection generics: DBIConnection-class, dbDataType, dbDisconnect, dbExistsTable, dbGetException, dbGetInfo, dbGetQuery, dbIsValid, dbListFields, dbListResults, dbListTables, dbReadTable, dbRemoveTable, dbSendQuery, dbSendStatement, dbWriteTable

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

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 DBIConnection object, as returned by dbConnect().
name	A character string specifying a DBMS 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.

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

Additional arguments

TBD: `temporary = NA`

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

Specification

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

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

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

See Also

Other DBIConnection generics: `DBIConnection-class`, `dbDataType`, `dbDisconnect`, `dbExecute`, `dbGetException`, `dbGetInfo`, `dbGetQuery`, `dbIsValid`, `dbListFields`, `dbListResults`, `dbListTables`, `dbReadTable`, `dbRemoveTable`, `dbSendQuery`, `dbSendStatement`, `dbWriteTable`

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 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. An attempt to fetch from a closed result set raises an error. If the n argument is not an atomic whole number greater or equal to -1 or Inf, an error is raised, but a subsequent call to dbFetch() with proper n argument succeeds. Calling dbFetch() on a result set from a data manipulation query created by dbSendStatement() can be fetched and return an empty data frame, with a warning.

Specification

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

- NA for SQL NULL values

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

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

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

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

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, dbIsValid, dbQuoteIdentifier, dbQuoteString

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)
```

dbGetException *Get DBMS exceptions*

Description

Get DBMS exceptions

Usage

```
dbGetException(conn, ...)
```

Arguments

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

Value

a list with elements errorNum (an integer error number) and errorMsg (a character string) describing the last error in the connection conn.

See Also

Other DBIConnection generics: DBIConnection-class, dbDataType, dbDisconnect, dbExecute, dbExistsTable, dbGetInfo, dbGetQuery, dbIsValid, dbListFields, dbListResults, dbListTables, dbReadTable, dbRemoveTable, dbSendQuery, dbSendStatement, dbWriteTable

dbGetInfo *Get DBMS metadata*

Description

Get DBMS metadata

Usage

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

Arguments

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

Value

a named list

Implementation notes

For DBIDriver subclasses, this should include the version of the package (driver.version) and the version of the underlying client library (client.version).

For DBIConnection objects this should report the version of the DBMS engine (db.version), database name (dbname), username, (username), host (host), port (port), etc. It MAY also include any other arguments related to the connection (e.g., thread id, socket or TCP connection type). It MUST NOT include the password.

For DBIResult objects, this should include the statement being executed (statement), how many rows have been fetched so far (in the case of queries, row.count), how many rows were affected (deleted, inserted, changed, (rows.affected), and if the query is complete (has.completed).

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, dbConnect, dbDataType, dbDriver, dbIsValid, dbListConnections

Other DBIConnection generics: DBIConnection-class, dbDataType, dbDisconnect, dbExecute, dbExistsTable, dbGetException, dbGetQuery, dbIsValid, dbListFields, dbListResults, dbListTables, dbReadTable, dbRemoveTable, dbSendQuery, dbSendStatement, dbWriteTable

Other DBIResult generics: DBIResult-class, dbBind, dbClearResult, dbColumnInfo, dbFetch, dbGetRowCount, dbGetRowsAffected, dbGetStatement, dbHasCompleted, dbIsValid, dbQuoteIdentifier, dbQuoteString

---

dbGetQuery *Send query, retrieve results and then clear result set*

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 free-d by dbClearResult().

Usage

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

Arguments

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

Details

This method is for SELECT queries only. 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. 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.

Implementation notes

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

Specification

Fetching multi-row queries with one or more columns be default returns the entire result. A value of Inf for the n argument is supported and also returns the full result. If more rows than available are fetched, the result is returned in full without warning. If zero rows are fetched, the columns of the data frame are still fully typed. Fetching fewer rows than available is permitted, no warning is issued.

A column named row_names is treated like any other column.

See Also

For updates: dbSendStatement() and dbExecute().

Other DBIConnection generics: DBIConnection-class, dbDataType, dbDisconnect, dbExecute, dbExistsTable, dbGetException, dbGetInfo, dbIsValid, dbListFields, dbListResults, dbListTables, dbReadTable, dbRemoveTable, dbSendQuery, dbSendStatement, dbWriteTable

Examples

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

dbWriteTable(con, "mtcars", mtcars)
dbGetQuery(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 DBIResult.
...	Other arguments passed on to methods.

Value

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

See Also

Other DBIResult generics: DBIResult-class, dbBind, dbClearResult, dbColumnInfo, dbFetch, dbGetInfo, dbGetRowsAffected, dbGetStatement, dbHasCompleted, dbIsValid, dbQuoteIdentifier, 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 DBIResult.
...	Other arguments passed on to methods.

Value

dbGetRowsAffected() returns a scalar number (integer or numeric), the number of rows affected by a data manipulation statement issued with dbSendStatement(). The value is available directly after the call and does not change after calling dbFetch(). For queries issued with dbSendQuery(), zero is returned before and after the call to dbFetch(). Attempting to get the rows affected for a result set cleared with dbClearResult() gives an error.

See Also

Other DBIResult generics: DBIResult-class, dbBind, dbClearResult, dbColumnInfo, dbFetch, dbGetInfo, dbGetRowCount, dbGetStatement, dbHasCompleted, dbIsValid, dbQuoteIdentifier, dbQuoteString

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 DBIResult.
...	Other arguments passed on to methods.

Value

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

See Also

Other DBIResult generics: DBIResult-class, dbBind, dbClearResult, dbColumnInfo, dbFetch, dbGetInfo, dbGetRowCount, dbGetRowsAffected, dbHasCompleted, dbIsValid, dbQuoteIdentifier, 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 DBIResult.
...	Other arguments passed on to methods.

Value

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

Specification

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

See Also

Other DBIResult generics: DBIResult-class, dbBind, dbClearResult, dbColumnInfo, dbFetch, dbGetInfo, dbGetRowCount, dbGetRowsAffected, dbGetStatement, dbIsValid, dbQuoteIdentifier, dbQuoteString

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: DBIDriver-class, DBIObject-class, DBIResult-class

Other DBIConnection generics: dbDataType, dbDisconnect, dbExecute, dbExistsTable, dbGetException, dbGetInfo, dbGetQuery, dbIsValid, dbListFields, dbListResults, dbListTables, dbReadTable, dbRemoveTable, dbSendQuery, dbSendStatement, dbWriteTable

Examples

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

## Not run:
con <- dbConnect(RPostgreSQL::PostgreSQL(), "username", "passsword")
con
dbDisconnect(con)

## End(Not run)
```

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 RPgSQL, RMySQL, etc. implement these operations in a DBMS-specific manner.

See Also

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

Other DBIDriver generics: dbConnect, dbDataType, dbDriver, dbGetInfo, 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, DBIDriver-class, DBIResult-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`, `DBIDriver-class`, `DBIObject-class`

Other DBIResult generics: `dbBind`, `dbClearResult`, `dbColumnInfo`, `dbFetch`, `dbGetInfo`, `dbGetRowCount`, `dbGetRowsAffected`, `dbGetStatement`, `dbHasCompleted`, `dbIsValid`, `dbQuoteIdentifier`, `dbQuoteString`

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

See Also

Other DBIDriver generics: DBIDriver-class, dbConnect, dbDataType, dbDriver, dbGetInfo, dbListConnections

Other DBIConnection generics: DBIConnection-class, dbDataType, dbDisconnect, dbExecute, dbExistsTable, dbGetException, dbGetInfo, dbGetQuery, dbListFields, dbListResults, dbListTables, dbReadTable, dbRemoveTable, dbSendQuery, dbSendStatement, dbWriteTable

Other DBIResult generics: DBIResult-class, dbBind, dbClearResult, dbColumnInfo, dbFetch, dbGetInfo, dbGetRowCount, dbGetRowsAffected, dbGetStatement, dbHasCompleted, dbQuoteIdentifier, dbQuoteString

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)
```

dbListConnections *List currently open connections*

Description

Drivers that implement only a single connections MUST return a list containing a single element. If no connection are open, methods MUST return an empty list.

Usage

```
dbListConnections(drv, ...)
```

Arguments

drv	A object inheriting from DBIDriver
...	Other arguments passed on to methods.

Value

a list

See Also

Other DBIDriver generics: DBIDriver-class, dbConnect, dbDataType, dbDriver, dbGetInfo, dbIsValid

dbListFields *List field names of a remote table*

Description

List field names of a remote table

Usage

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

Arguments

conn	A DBIConnection object, as returned by dbConnect().
name	a character string with the name of the remote table.
...	Other parameters passed on to methods.

Value

a character vector

See Also

dbColumnInfo() to get the type of the fields.

Other DBIConnection generics: DBIConnection-class, dbDataType, dbDisconnect, dbExecute, dbExistsTable, dbGetException, dbGetInfo, dbGetQuery, dbIsValid, dbListResults, dbListTables, dbReadTable, dbRemoveTable, dbSendQuery, dbSendStatement, dbWriteTable

Examples

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

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

dbDisconnect(con)
```

---

dbListResults *A list of all pending results*

Description

List of DBIResult objects currently active on the connection.

Usage

```
dbListResults(conn, ...)
```

Arguments

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

Value

a list. If no results are active, an empty list. If only a single result is active, a list with one element.

See Also

Other DBIConnection generics: DBIConnection-class, dbDataType, dbDisconnect, dbExecute, dbExistsTable, dbGetException, dbGetInfo, dbGetQuery, dbIsValid, dbListFields, dbListTables, dbReadTable, dbRemoveTable, dbSendQuery, dbSendStatement, dbWriteTable

dbListTables *List remote tables*

Description

Returns the unquoted names of remote tables accessible through this connection. This should, where possible, include temporary tables, and views.

Usage

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

Arguments

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

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

Additional arguments

TBD: temporary = NA

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

See Also

Other DBIConnection generics: DBIConnection-class, dbDataType, dbDisconnect, dbExecute, dbExistsTable, dbGetException, dbGetInfo, dbGetQuery, dbIsValid, dbListFields, dbListResults, dbReadTable, dbRemoveTable, dbSendQuery, dbSendStatement, dbWriteTable

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 name, to make sure that you generate valid SQL and avoid SQL injection.

Usage

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

Arguments

conn	A subclass of DBIConnection, representing an active connection to an DBMS.
x	A character vector 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. An error is raised if the input contains `NA`, but not for an empty string.

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

Specification

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

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

See Also

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

Examples

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

---

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 avoid SQL injection.

Usage

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

Arguments

conn	A subclass of DBIConnection, representing an active connection to an DBMS.
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 SQL should also return them unchanged. (For backends it may be most convenient to return SQL objects to achieve this behavior, but this is not required.)

Specification

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

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

See Also

Other DBIResult generics: DBIResult-class, dbBind, dbClearResult, dbColumnInfo, dbFetch, dbGetInfo, dbGetRowCount, dbGetRowsAffected, dbGetStatement, dbHasCompleted, dbIsValid, dbQuoteIdentifier

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 *Copy data frames from database tables*

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.

Usage

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

Arguments

conn	A DBIConnection object, as returned by dbConnect().
name	A character string specifying the unquoted DBMS table name, or the result of a call to dbQuoteIdentifier().
...	Other parameters passed on to methods.

Value

dbReadTable() returns a data frame that contains the complete data from the remote table, effectively the result of calling dbGetQuery() with SELECT * FROM <name>. An error is raised if the table does not exist. An empty table is returned as a data frame with zero rows.

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

- If FALSE or NULL, the returned data frame doesn't have row names.
- If TRUE, a column named "row_names" is converted to row names, an error is raised if no such column exists.
- If NA, a column named "row_names" is converted to row names if it exists, otherwise no translation occurs.
- If a string, this specifies the name of the column in the remote table that contains the row names, an error is raised if no such column exists.

The default is `row.names = FALSE`.

If the database supports identifiers with special characters, the columns in the returned data frame are converted to valid R identifiers if the `check.names` argument is TRUE, otherwise non-syntactic column names can be returned unquoted.

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

Additional arguments

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

- `row.names`
- `check.names`

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

Specification

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

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

See Also

Other DBIConnection generics: `DBIConnection-class`, `dbDataType`, `dbDisconnect`, `dbExecute`, `dbExistsTable`, `dbGetException`, `dbGetInfo`, `dbGetQuery`, `dbIsValid`, `dbListFields`, `dbListResults`, `dbListTables`, `dbRemoveTable`, `dbSendQuery`, `dbSendStatement`, `dbWriteTable`

Examples

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

dbWriteTable(con, "mtcars", mtcars[1:10, ])
dbReadTable(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 DBIConnection object, as returned by dbConnect().
name	A character string specifying a DBMS table name.
...	Other parameters passed on to methods.

Value

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

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

Specification

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

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

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

See Also

Other DBIConnection generics: DBIConnection-class, dbDataType, dbDisconnect, dbExecute, dbExistsTable, dbGetException, dbGetInfo, dbGetQuery, dbIsValid, dbListFields, dbListResults, dbListTables, dbReadTable, dbSendQuery, dbSendStatement, dbWriteTable

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().

Usage

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

Arguments

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

Details

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

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

Value

dbSendQuery() returns an S4 object that inherits from DBIResult. The result set can be used with dbFetch() to extract records. Once you have finished using a result, make sure to clear it with dbClearResult(). An error is raised when issuing a query over a closed or invalid connection, if the syntax of the query is invalid, or if the query is not a non-NA string.

Specification

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

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

See Also

For updates: dbSendStatement() and dbExecute().

Other DBIConnection generics: DBIConnection-class, dbDataType, dbDisconnect, dbExecute, dbExistsTable, dbGetException, dbGetInfo, dbGetQuery, dbIsValid, dbListFields, dbListResults, dbListTables, dbReadTable, dbRemoveTable, dbSendStatement, dbWriteTable

Examples

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

dbWriteTable(con, "mtcars", mtcars)
rs <- dbSendQuery(con, "SELECT * FROM mtcars WHERE cyl = 4;")
dbFetch(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 DBIConnection object, as returned by dbConnect().
statement	a character string containing SQL.
...	Other parameters passed on to methods.

Details

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

Value

dbSendStatement() returns an S4 object that inherits from DBIResult. The result set can be used with dbGetRowsAffected() to determine the number of rows affected by the query. Once you have finished using a result, make sure to clear it with dbClearResult(). An error is raised when issuing a statement over a closed or invalid connection, if the syntax of the statement is invalid, or if the statement is not a non-NA string.

Specification

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

See Also

For queries: dbSendQuery() and dbGetQuery().

Other DBIConnection generics: DBIConnection-class, dbDataType, dbDisconnect, dbExecute, dbExistsTable, dbGetException, dbGetInfo, dbGetQuery, dbIsValid, dbListFields, dbListResults, dbListTables, dbReadTable, dbRemoveTable, dbSendQuery, dbWriteTable

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

dbDisconnect(con)
```

---

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

Specification

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

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 curent 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 DBIConnection object, as returned by `dbConnect()`.
`name`	A character string specifying the unquoted DBMS table name, or the result of a call to `dbQuoteIdentifier()`.

value	a data.frame (or coercible to data.frame).
...	Other parameters passed on to methods.

Value

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

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

Additional arguments

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

- row.names (default: NA)
- overwrite (default: FALSE)
- append (default: FALSE)
- field.types (default: NULL)
- temporary (default: FALSE)

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

Specification

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

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

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

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

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

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

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

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

Mixing column types in the same table is supported.

The `field.types` argument must be a named character vector with at most one entry for each column. It indicates the SQL data type to be used for a new column.

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

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

See Also

Other DBIConnection generics: `DBIConnection-class`, `dbDataType`, `dbDisconnect`, `dbExecute`, `dbExistsTable`, `dbGetException`, `dbGetInfo`, `dbGetQuery`, `dbIsValid`, `dbListFields`, `dbListResults`, `dbListTables`, `dbReadTable`, `dbRemoveTable`, `dbSendQuery`, `dbSendStatement`

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")
```

`make.db.names` *Make R identifiers into legal SQL identifiers*

Description

These methods are DEPRECATED. Please use `dbQuoteIdentifier()` (or possibly `dbQuoteString()`) instead.

Usage

```
make.db.names(dbObj, snames, keywords = .SQL92Keywords, unique = TRUE,
  allow.keywords = TRUE, ...)

make.db.names.default(snames, keywords = .SQL92Keywords, unique = TRUE,
  allow.keywords = TRUE)

isSQLKeyword(dbObj, name, keywords = .SQL92Keywords, case = c("lower",
  "upper", "any")[3], ...)

isSQLKeyword.default(name, keywords = .SQL92Keywords, case = c("lower",
  "upper", "any")[3])
```

Arguments

`dbObj`	any DBI object (e.g., `DBIDriver`).
`snames`	a character vector of R identifiers (symbols) from which we need to make SQL identifiers.
`keywords`	a character vector with SQL keywords, by default it's `.SQL92Keywords` defined by the DBI.
`unique`	logical describing whether the resulting set of SQL names should be unique. Its default is `TRUE`. Following the SQL 92 standard, uniqueness of SQL identifiers is determined regardless of whether letters are upper or lower case.
`allow.keywords`	logical describing whether SQL keywords should be allowed in the resulting set of SQL names. Its default is `TRUE`
`...`	any other argument are passed to the driver implementation.
`name`	a character vector with database identifier candidates we need to determine whether they are legal SQL identifiers or not.
`case`	a character string specifying whether to make the comparison as lower case, upper case, or any of the two. it defaults to `any`.

Details

The algorithm in `make.db.names` first invokes `make.names` and then replaces each occurrence of a dot `.` by an underscore `_`. If `allow.keywords` is `FALSE` and identifiers collide with SQL keywords, a small integer is appended to the identifier in the form of `"_n"`.

The set of SQL keywords is stored in the character vector `.SQL92Keywords` and reflects the SQL ANSI/ISO standard as documented in "X/Open SQL and RDA", 1994, ISBN 1-872630-68-8. Users can easily override or update this vector.

Value

`make.db.names` returns a character vector of legal SQL identifiers corresponding to its `snames` argument.

`SQLKeywords` returns a character vector of all known keywords for the database-engine associated with `dbObj`.

`isSQLKeyword` returns a logical vector parallel to `name`.

Bugs

The current mapping is not guaranteed to be fully reversible: some SQL identifiers that get mapped into R identifiers with `make.names` and then back to SQL with `make.db.names()` will not be equal to the original SQL identifiers (e.g., compound SQL identifiers of the form `username.tablename` will loose the dot ".").

References

The set of SQL keywords is stored in the character vector `.SQL92Keywords` and reflects the SQL ANSI/ISO standard as documented in "X/Open SQL and RDA", 1994, ISBN 1-872630-68-8. Users can easily override or update this vector.

---

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)
```

Arguments

x	A character vector to label as being escaped SQL.
...	Other arguments passed on to methods. Not otherwise used.

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 *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()`. These methods are mostly useful for backend implementers.

Usage

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

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

Arguments

`con`	A database connection.
`table`	Name of the table. Escaped with `dbQuoteIdentifier()`.
`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 put in front of column id.

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 *Create a simple table*

Description

Exposes 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	Name of the table. Escaped with dbQuoteIdentifier().
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 statement.
...	Other arguments used by individual methods.

DBI-backends

If you implement one method (i.e. for strings or data frames), you need to implement both, otherwise the S4 dispatch rules will be ambiguous and will generate an error on every call.

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
- Converts all columns to strings
- 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

Safely interpolate values into an SQL string

Usage

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

Arguments

conn	A database connection.
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	Named values (for ...) or a named list (for .dots) to interpolate into a string. All strings will be first escaped with dbQuoteString() prior to interpolation to protect against SQL injection attacks.

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--;")
```

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

Description

Refer to a table nested in a hierarchy (e.g. within a schema)

Usage

```
Table(...)
```

Arguments

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

---

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

Specification

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

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

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

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

The transaction isolation level is not specified by DBI.

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)
```

Index